Scalable Computer Network Emulation Using Node Virtualization and Resource Monitoring

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Steffen Dirk Maier

aus Filderstadt

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr.-Ing. Bernd Freisleben

Tag der mündlichen Prüfung: 16.02.2011

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2011

Acknowledgments

First of all, I would like to thank my PhD advisor Prof. Dr. Kurt Rothermel for providing the opportunity and the support to pursue a PhD within the Distributed Systems research group at the Universität Stuttgart. For kindly serving on my PhD committee and reviewing this thesis, I am grateful to Prof. Dr. Bernd Freisleben at the Philipps-Universität Marburg.

The colleagues and friends of the research group as well as of neighboring groups made this a memorable experience. Thank you for all the interesting discussions on research and everything else. My thanks also go to the students who contributed to the "Network Emulation Testbed" (NET) research project.

Part of this work was funded by the German Research Foundation (DFG) under grant DFG-GZ RO 1086/9-1. Grants of the State of Baden-Württemberg allowed for the acquisition of the NET hardware without which this research would not have been possible. A PPP-USA grant of the German Academic Exchange Service (DAAD) enabled face to face research with our Emulab/Netbed colleagues at the Flux Group, University of Utah, USA. I would like to express my gratitude for the support.

My thanks go to the colleagues at IBM Deutschland Research & Development for coping with my sometimes unusual working time and making it possible to finish my thesis after I had left the research group.

Last but not least, I am deeply grateful to my parents and grandparents for their continuous support and patience.

Contents

List of Abbreviations

ABI application binary interface

AODV ad-hoc on-demand distance vector

API application programming interface

ARP address resolution protocol

BER bit error rate

BSD Berkeley Software Distribution

CPU central processing unit

CSMA/CA carrier sense multiple access with collision avoidance

CSMA/CD carrier sense multiple access with collision detection

CSMA carrier sense multiple access

FER frame error rate

GUI graphical user interface

I/O input/output

ICMP Internet control message protocol

IOCTL input/output control

IP Internet protocol

ISA instruction set architecture

LAN local area network

LLC logical link control

MAC medium access control

MANET mobile ad hoc network

MSS maximum segment size

MTU maximum transfer unit

NET Network Emulation Testbed

NIC network interface card

OS operating system

PCB protocol control block

PDES parallel discrete event simulator

pnode physical node

QoS quality of service

RTT round trip time

SNR signal to noise ratio

TCP transmission control protocol

UDP user datagram protocol

UML User Mode Linux

VLAN virtual local area network

VM virtual machine

VMM virtual machine monitor (hypervisor)

vnode virtual node

VR virtual routing

VRF Virtual Routing and Forwarding

WLAN wireless local area network

Abstract

Ongoing development of computer network technology requires new communication protocols on all layers of the protocol stack to adapt to and to exploit technology specifics. The performance of new protocol implementations has to be evaluated before deployment. Computer network emulation enables the execution of real unmodified protocol implementations within a configurable synthetic environment. Since network properties are reproduced synthetically, emulation supports reproducible measurement results for wired and wireless networks. Meaningful evaluation scenarios typically involve a large number of communicating nodes. Reproducing the network properties of the medium access control layer can be accomplished efficiently on cheap common off the shelf computers and allows to evaluate network protocols, transport protocols, and applications. However, meaningful emulation scenario sizes often require more nodes than affordable computers.

To scale the number of nodes in an emulation scenario beyond the available computers, we discuss approaches to virtualization and operating system partitioning. Focusing on the latter, we argue for virtual protocol stacks, which provide an extremely lightweight node virtualization enabling the execution of multiple instances of software to be evaluated on each physical computer. To connect virtual nodes on the same and on different computers, we design and implement a highly efficient software communication switch. A centralized emulation control component distributes dynamic network property updates which result from node mobility for instance. To handle the large number of nodes and thus increased updates, we propose a hierarchical control where the central component delegates updates to sub-components distributed over the computers of an emulation system. Extensive evaluations show the scalability of our virtualized network emulation system.

Virtual nodes executed on the same computer share its limited resources. Hosting too many virtual nodes on the same computer may lead to resource contention. This can cause unrealistic measurement results and is thus undesirable. Discussing different approaches to handle resource contention, we argue for detection and recovery. We define quality criteria that allow the detection of resource con-

tention. In order to observe those quality criteria during emulation experiments, we propose a highly lightweight monitoring approach. Our monitoring is based on instrumenting an operating system kernel and observing basic resource scheduling events. This enables the detection of even peak resource usage within a split second. Thorough evaluations demonstrate the effectiveness of quality criteria and monitoring as well as the negligible overhead of our monitoring approach.

Deutsche Zusammenfassung

Skalierbare Rechnernetz-Emulation durch Knotenvirtualisierung und Betriebsmittelüberwachung

1 Einführung

Die voranschreitende Entwicklung von Rechnernetz-Technologien und zugehörigen Anwendungsszenarien erfordert verbesserte oder neue Protokolle auf allen Schichten des Kommunikationsstapel inklusive verteilter Anwendungen. Auf der Anwendungsschicht entstehen neue Methoden wie Peer-to-Peer Datenverteilung [ATS04], elastische Overlay-Netze [ABKM01], kontextbezogene Ereignisbenachrichtigung [Bau07] oder energieeffiziente Diensterkennung [Sch07]. Auf der Transportschicht gibt es neue Varianten der Überlaststeuerung [XHR04, JWL04, KHF06] oder Anpassungen an drahtlose Umgebungen [Ela02, EKL05, EKL06]. Auf der Vermittlungsschicht werden neue Ansätze zur Bestimmung von Leitwegen in drahtgebundenen Netzen [KKKM07, LCR+07] sowie in mobilen drahtlosen Netzen [JM96, PR99, LNT02] entwickelt. Drahtlose Mesh-Netze erfordern Anpassungen auf allen Schichten des Protokollstapel [AWW05].

Während der Entwicklung von verbesserten oder neuen Protokollen und verteilten Anwendungen ist es notwendig, ihre Leistung zu untersuchen. Traditionell kommen drei verschiedene Methoden zur Anwendung: Mathematische Analyse, Simulation und Probebetrieb in Realumgebung. Mathematische Modelle [Jai91] können nur in frühen Entwicklungsphasen sinnvoll eingesetzt werden, da die Modelle zwangsläufig von der Realität abstrahieren und daher typischerweise nur gewisse Aspekte des gesamten Systems nachbilden. Als Folgeschritt unterstützt die Simulation komplexere Modelle und erlaubt die Ausführung solcher Modelle zur Leistungsbewertung beispielsweise in Simulatoren wie dem Network Simulator 2 (ns2) [BEF+00] oder GloMoSim [ZBG98]. Der letzte Entwicklungsschritt beinhaltet üblicherweise eine prototypische Implementierung. Prüfstände zur Ausführung solcher Prototypen können für drahtgebundene Netze [ABKM03, CCR+03, BBC+04, APST05] entworfen werden [BL03] wie auch für drahtlose Netze [RKM+05, RSO+05].

Im Anschluss an diesen letzten Entwicklungsschritt gilt es, die Implementierung in Betrieb zu nehmen. Da es schwierig sein kann, den Betrieb direkt in einem bereits laufenden System wie dem Internet vorzunehmen, gibt es spezielle Plattformen zur initialen Ausbringung [BFH+06, GEN06]. Solche Plattformen ermöglichen auch die Leistungsbewertung mit Ergebnissen, die repräsentativ für eine aktuelle Internet-Umgebung sind [PHM06].

Während der Entwicklung verteilter Software ist es oft notwendig, die Leistung bestimmter Aspekte der Implementierung isoliert zu betrachten. Jedoch beeinflussen nicht nur andere gleichzeitige Nutzer einer Ausbringungsplattform Messergebnisse, sondern auch die Internet-Umgebung mit der diese Plattform verbunden ist. Da derartige Plattformen keine reproduzierbaren Messungen erlauben, sind sie für solche Leistungsbewertungen ungeeignet. Isolierte Prüfstände mit exklusivem Benutzerzugriff erlauben reproduzierbare Ergebnisse. Jedoch können notwendige Geldmittel oder verwaltungstechnische Einschränkungen den Aufbau eines solchen Prüfstandes unmöglich machen. Im Gegensatz zu Prüfständen für drahtgebundene Netze, ist es praktisch unmöglich reproduzierbare Messungen in drahtlosen Umgebungen durchzuführen. Simulation würde zwar Reproduzierbarkeit gewährleisten, aber eine existierende Realimplementierung muss typischerweise für ein Simulationsmodell erneut implementiert werden. Darüber hinaus können Messergebnisse sich zwischen Simulation und Realumgebung auf Grund der Modellabstraktion der Simulation unterscheiden.

Offensichtlich besteht in den zur Verfügung stehenden Leistungsbewertungsmethoden eine Lücke zwischen Simulation und Messung in Realumgebung. *Rechnernetz-Emulation* füllt diese Lücke, indem sie die Ausführung unmodifizierter realer Implementierungen in einer konfigurierbaren synthetischen Umgebung unterstützt. Hierzu reproduziert eine Software, genannt *Emulationswerkzeug*, vorgegebene Netzeigenschaften während sie über eine flexible Netz-Hardware mit möglicherweise anderen Eigenschaften kommuniziert. Die Kombination aus flexibler Hardware und geeigneten Emulationswerkzeugen wird *Prüfstand für Rechnernnetz-Emulation* genannt. Ein Kommunikationsprotokoll oder eine verteilte Anwendung ist ein *Software-Prüfling*. Protokolle auf Verbindungsschicht oder darunter werden hier nicht betrachtet, da die Leistungsbewertung von beispielsweise der Medienzugriffssteuerung (MAC) spezielle Hardware-Umgebungen zur Nachbildung der Bitübertragungsschicht [HH02, SBBD03, JS03, VBV+05] erfordert.

Leistungsbewertung in Szenarien mit mobiler drahtloser Kommunikation erfordert eine große Anzahl kommunizierender Knoten. Die Untersuchung neuer Anwendungen in traditionellen infrastruktur-basierten Netzen, wie beispielsweise eines großräumigen Lokationsdienstes, benötigen ebenfalls die Emulation vieler Knoten, da sowohl Endsysteme als auch Vermittlungselemente der Infrastruk-

tur in einem Emulationsmodell berücksichtigt werden müssen. Herkömmliche Emulationssysteme nahmen an, dass ein kommunizierender Knoten eines Emulationsszenarios einem Rechner eines Emulationsprüfstandes entspricht. Viele Prüflinge benötigen nicht die gesamten Betriebsmittel eines Prüfstandrechners, sondern nur einen Bruchteil davon. Neuere Emulationsysteme nutzen dies aus, um einen Prüfstandrechner (*p-Knoten*) in mehrere virtuelle Knoten (*v-Knoten*) zu partitionieren. Jeder Knoten eines Emulationsszenarios wird dann durch einen v-Knoten repräsentiert.

Die Kosten der Virtualisierung bestimmen welcher Anteil der Betriebsmittel den Prüflingen noch zur Verfügung stehen. Um maximale Skalierbarkeit mit einer gegebenen Prüfstand-Hardware zu erreichen, wird Knotenvirtualisierung mit minimalem Overhead benötigt. Auf jedem p-Knoten werden mehrere Prüflinge in jeweils einem eigenen v-Knoten ausgeführt und teilen sich die Betriebsmittel des beherbergenden p-Knoten. Die Beherbergung zu vieler v-Knoten führt zu Wettbewerb um Betriebsmittel. Dies kann wiederum zu unrealistischen Messergebnissen führen und ist daher unterwünscht. Um solchen Wettbewerb zu erkennen, ist es notwendig, Qualitätskriterien für realistische Rechnernetz-Emulation zu definieren und während Messläufen zu überwachen.

Diese Arbeit leistet folgende fünf Beiträge: Architektur einer leichtgewichtigen Knotenvirtualisierung auf Basis virtueller Protokollstapel, Architektur eines effizienten virtuellen Software-Switch, Architektur einer skalierbaren hierarchischen Emulationssteuerung, Definition von Qualitätskriterien zur Erkennung von Wettbewerb um Betriebsmittel, sowie Architektur einer leichtgewichtigen Überwachung der Qualitätskriterien während Emulations-Experimenten.

Die Struktur dieser Arbeit gliedert sich wie folgt. In Kapitel 2 werden Grundlagen der Rechnernetz-Emulation vorgestellt. Zur Ermöglichung skalierbarer Emulation, wird die Virtualisierung von Knoten in Kapitel 3 diskutiert und evaluiert. Um realistische Messergebnisse bei der Leistungsbewertung mit Hilfe eines virtualisierten Emulationssystems zu erhalten, werden in Kapitel 4 Qualitätskriterien zur Erkennung von Wettbewerb um Betriebsmittel definiert und ein Überwachungsverfahren für Qualitätskriterien vorgestellt und evaluiert. Kapitel 5 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick auf mögliche zukünftige Forschungsthemen.

2 Grundlagen der Rechnernetz-Emulation

In Kapitel 2 wird die Architektur eines Systems zur Rechnernetz-Emulation diskutiert sowie darauf aufbauend die prototypische Implementierung eines Prüfstandes für Rechnernetz-Emulation beschrieben. Die zu Grunde liegende Klassifikation der Architekturkomponenten ist aus einer vorhergehenden Arbeit [Her05] übernommen.

Als Einführung in die Architekturdiskussion werden zunächst für alle Protokollschichten die Netzeigenschaften beleuchtet, die für die Nachbildung in einer Emulation relevant sind. Die Bitübertragungsschicht übernimmt die Übertragung eines Bit-Stroms zwischen zwei direkt verbundenen Kommunikationspartnern. Typischerweise in der Hardware eines Adapters implementiert, beinhalten ihre Eigenschaften die Übertragungsrate, Ausbreitungsverzögerung und Bitfehlerrate. Die Sicherungsschicht ist zuständig für die Übertragung von Rahmen mit begrenzter Größe zwischen direkt verbundenen Stationen und besteht aus zwei Teilschichten. Die Medienzugriffssteuerung (MAC) koordiniert den exklusiven Zugriff ein gemeinsames Übertragungsmedium und führt dadurch Rahmenverzögerung und Rahmenfehlerrate ein. Die logische Verbindungssteuerung (LLC) kann optional durch Verbindungsorientiertheit, Zuverlässigkeit oder Flusssteuerung weitere Rahmenverzögerung hinzufügen. Die Vermittlungsschicht nutzt global eindeutige Stationsadressen und verbindet verschiedene Netztechnologien, indem sie Pakete weiterleitet. Leitwegermittlung und andere Funktionen führen Paketverzögerung ein. Weiterhin können bei der Weiterleitung Pakete verworfen oder ihre Reihenfolge vertauscht werden. Die Transportschicht überträgt Nachrichten an verschiedene Dienste auf Rechnern. Verschiedene optionale Funktionen wie Verbindungsorientiertheit, Zuverlässigkeit, Fluss- oder Überlaststeuerung verursachen Nachrichtenverzögerung. Die Medienzugriffssteuerung ist die niedrigste Protokollschicht, die von einem Emulations-Werkzeug in Software nachgebildet werden kann. Nachzubildende Netzeigenschaften lassen sich auf Rahmenverzögerung und Rahmenverlustrate reduzieren. Alle darüberliegenden Schichten kommen als potentielle Prüflinge in Frage.

Der zweite Teil der Architekturdiskussion beschäftigt sich mit drei Software-Modulen, aus denen sich ein Emulationssystem zusammensetzt: Netzmodell, Emulationswerkzeug und Emulationssteuerung. Aus einer vom Benutzer vorgegebenen Szenariospezifikation lässt sich ein Netzmodell ableiten, in dem die Emulationssteuerung statische und dynamische Parameter verwaltet. Das Modell lässt sich entweder zentral verwalten oder in hybrider Weise aufteilen in einen zentralen Anteil sowie Untermodelle verteilt auf die Prüfstandrechner. Für die Architektur von Emulationswerkzeugen gibt es drei Alternativen. Ein zentralisiertes Werkzeug muss den gesamten Netzverkehr eines Szenarios verarbeiten und kann deshalb zum Engpass werden. Eine Verteilungsvariante mit einer Werkzeuginstanz pro Netzsegment verteilt die Verkehrslast auf mehrere Werkzeuge, benötigt jedoch zusätzliche Rechner pro Netzsegment. In der dritten Alternative existiert für jede Netzanbindung eine separate Werkzeuginstanz, die dann jeweils nur den Verkehr von und zu einer Station verarbeiten muss und auf dem jeweiligen Stationsrechner ausgeführt werden kann. Die Emulationssteuerung verwaltet das Netzmodell und konfiguriert hieraus die Emulationswerkzeuge. Mit verteilten Werkzeugen erfor-

dert zentralisierte Steuerung Kommunikation, während bei verteilter Steuerung jedes Werkzeug seinen vorberechneten Parameterablauf selbständig mit Hilfe einer lokalen synchronisierten Uhr während eines Experiments abspielt. Am Geeignetsten erscheinen hybrides Modell, verteilte Werkzeuge pro Netzanbindung und zentralisierte Steuerung.

Der dritte und letzte Teil der Architekturdiskussion behandelt Emulationsfunktionalität bestehend aus Emulationssteuerung, und je einem Emulationswerkzeug für exklusives Kommunikationsmedium sowie gemeinsames Medium. Jedes Emulationsexperiment läuft in drei Phasen ab. In der Vorbereitungsphase werden Betriebsmittel des Prüfstandes zugewiesen und konfiguriert sowie gegebenenfalls eine Aufwärmung der Prüflinge vorgenommen. Während der Messphase bilden die verteilten Emulationswerkzeuge die vorgegebenen Netzeigenschaften nach und werden von der Steuerung über dynamische Parameteränderungen informiert. Schließlich werden in der Aufräumphase Betriebsmittel wieder freigegeben. Das globale Emulationsmodell besteht bei n Knoten aus einer $n \times n$ Matrix mit dem Dreitupel $e = (p_b, b, t_p)$ als Elemente, wobei p_b die Bitfehlerrate, b die Übertragungsrate und t_p die Ausbreitungsverzögerung darstellen. Daraus kann jedes Werkzeug mit Hilfe der Rahmenlänge l die Rahmenfehlerrate $p_f = 1 - (1 - p_b)^l$ und Serialisierungsverzögerung $t_s = l/b$ berechnen.

Ein Emulationswerkzeug für exklusives Kommunikationsmedium auf Knoten k verwendet Zeile $\{e_{ij} | i = k\}$ und Spalte $\{e_{ij} | j = k\}$ des globalen Modells. Ein Rahmen wird genau dann verworfen, wenn eine Zufallszahl R mit $0 \leq R < 1$ kleiner als die Rahmenfehlerrate ist $R < p_f$. Jeder nicht verworfene Rahmen wird mit einer Zeitmarke $t_q = t + t_p + t_s$ versehen, wobei t die aktuelle Zeit ist, und in eine Warteschlange gestellt, aus der er schließlich zum Zeitpunkt t_q genommen und übertragen wird [HR02, Her05].

Ein Emulationswerkzeug für gemeinsames Kommunikationsmedium bildet die Medienzugriffssteuerung nach. Für die weitverbreitete Klasse des Mehrfachzugriffs mit Trägererkennung (CSMA) enthält das lokale Modell dieselbe Zeile und Spalte wie zuvor. Als Erweiterung führt jede Station zusätzlich ein lokales Modell des gemeinsamem Mediums mit Hilfe eines virtuellen Trägers, um hieraus Rahmenverzögerung sowie -verlust durch Kollisionen nachzubilden [Mai02, HMR03, Yan04, Her05].

Das „Network Emulation Testbed“ (NET) setzt die beschriebene Architektur in einem Emulationsprüfstand um. Er besteht aus 64 handelsüblichen Personalcomputern die durch ein Emulationsnetz sowie ein Verwaltungsnetz verbunden sind. Auf dem Emulationsnetz wird die Verbindungstopologie eines Szenarios durch Hardware-Unterstützung mit IEEE 802.1Q VLAN Technologie [IEE03] nachgebildet. Tagged VLANs ermöglichen mehrere logische Netzanbindungen pro physischer Anbindung eines Rechners. Die Emulationswerkzeuge sind dynamisch

ladbare Module für den Linux-Kern in Version 2.4.24 und integrieren sich in Senderichtung über die Gerätetreiberfunktion hard_start_xmit() sowie in Empfangsrichtung entweder über dev_add_pack() bzw. einen neuen Eingriffspunkt in netif_receive_skb(). Konfiguriert werden die Werkzeuge über einen Systemruf zur Ein-/Ausgabesteuerung (IOCTL). Für drahtlose mobile ad-hoc Netze simuliert die Emulationssteuerung Knotenmobilität mit Hilfe von Bewegungsspuren [HMTR04]. Für jedes Knotenpaar aus Sender und Empfänger kann über deren Position auf einer zweidimensionalen Fläche die Funkausbreitungsbedingung berechnet werden [Fri46, Rap01, BMJ+98, FV02]. Da realistische Ausbreitungsmodelle wichtig sind [TMB01, SHR05, Ste09], kann auch die geographische Umgebung mit statischen Hindernissen wie Gebäuden berücksichtigt werden [Com91, Lan99, WHL99]. Für ein diskretisiertes zweidimensionales Raster werden für alle mögliche Knotenpaarpositionen Dämpfungswerte vorberechnet und auf Sekundärspeicher abgelegt für ein späteres Nachschlagen durch die Emulationssteuerung [SHR05, Her05, Ste09].

NET wurde erfolgreich zur Leistungsbewertung in einer Dissertation [Bau07] sowie einer Master's Thesis [Kar04] eingesetzt. Mit den in diesem Kapitel beschriebenen Konzepten wird jeder Knoten eines Emulationsszenarios auf einen Rechner des Prüfstandes abgebildet wodurch die Szenariengröße hier auf 64 Knoten beschränkt ist.

3 Knotenvirtualisierung

Kapitel 3 beschäftigt sich mit Knotenvirtualisierung als Mittel für skalierbare Rechnernetz-Emulation. Hierzu werden verwandte Arbeiten diskutiert, Anforderungen definiert und Grundlagen von Protokollstapeln vorgestellt. Anschließend werden Ansätze zur Knotenvirtualisierung diskutiert und quantitativ verglichen. Für den geeignetsten Ansatz wird dann unsere Architektur beschrieben und Implementierung evaluiert. Teile dieses Kapitels wurden bereits früher veröffentlicht in Tagungsbänden [MHR05, MGWR07] und einem Fachblatt [MHR07].

Verwandte Arbeiten, um Szenariengrößen mit mehr Knoten als Rechnern zu ermöglichen, lassen sich in drei Klassen aufteilen: Parallelisierung, Abstraktion des Emulationsmodells und Knotenvirtualisierung. Simulation mit meist diskreten Ereignissen lässt sich als Emulationswerkzeug einsetzen, indem Ereignisse in Echtzeit abgearbeitet werden und eine Schnittstelle zur Verarbeitung echten Netzverkehrs hinzugefügt wird. Durch Partitionierung der Netztopologie eines Emulationsszenarios lässt sich die Echtzeitsimulation auf mehrere Rechner parallelisieren, um größere Szenarien zu unterstützen [RFA99, Fal99, SU01, LC04]. Die Skalierbarkeit hängt von Netzverkehr zwischen Partitionen und dadurch notwendiger Synchronisation ab. Außerdem werden zusätzliche Rechner benötigt, um Prüflinge auszuführen, deren Netzverkehr durch die parallelisierte Echtzeitsimulation geleitet wird.

Durch Abstraktion des Emulationsmodells wird weniger Rechenleistung für die Emulation benötigt, was die Verarbeitung von mehr Netzverkehr auf derselben Hardware erlaubt und dadurch die Skalierbarkeit steigert. Während die zuvor besprochenen Ansätze auf Paketebene simulieren, lassen sich langlebige Verbindungen auf Transportschicht durch Flüssigkeitsströmung mit Hilfe von Differentialgleichungen modellieren. Die Lösung der Gleichungen ergibt die mittlere Warteschlangenlänge sowie die mittlere Paketverzögerung auf dem Pfad von Quelle zu Senke. Daraus lässt sich der Durchsatz ermitteln. Durch Messung von Statistikwerten des echten Netzverkehrs der Prüflinge und Berücksichtigung in den Differentialgleichungen lässt sich Simulation auf Paketebene mit Flüssigkeitsströmung kombinieren [ZJTB03, ZJTB04, Kid04, KSU05]. Flüssigkeitsströmungsmodelle können Emulation durch effiziente Nachbildung von Hintergrundlast ergänzen. Eine andere Möglichkeit der Modellabstraktion beschreibt ROSENET [GF04, Gu08]. Hier kommen ein Emulator mit niederer Genauigkeit und ein Simulator mit hoher Genauigkeit zum Einsatz. Der Emulator verzögert und verwirft Pakete zwischen zwei Prüflingen auf zwei verbundenen Rechnern nach einem statistischen Modell. Dabei zeichnet er Zwischenankunftszeiten von Paketen der Prüflinge auf und leitet diese an den Simulator weiter, der ein zukünftiges Emulationsmodell berechnet, das Prüflingsverkehr so beeinflusst wie das simulierte Netzwerk, durch das die Pakete geschickt werden. Ein solcher Ansatz ist vorwiegend für Transportschichtverbindungen in Internet-ähnlichen Netzen geeignet und stützt sich auf die Beständigkeit von Internet-Pfadeigenschaften.

Ansätze zur Knotenvirtualisierung nutzen aus, dass Prüflinge oft nur einen Bruchteil der Betriebsmittel eines Prüfstandrechners benötigen und daher mehrere Prüflinge auf einem Rechner ausgeführt werden können. Die Virtualisierung von Protokollstapel [HSK99, ESW01] oder ganzen Maschinen [JX03] im Benutzeradressraum sind aufgrund des Overhead eher für Testzwecke als für Leistungsbewertung geeignet. ModelNet [VYW+02, YED+03] bildet ein Netz von Punkt-zu-Punkt-Verbindungen auf Vermittlungsschicht nach. Erweiterungen für drahtlose Netze [MYV02, MRBV04, MI04, MI05, ESHF04] sind zentralisiert und daher wenig skalierbar. Bisherige Ansätze zur Virtualisierung von Protokollstapel im Kernadressraum [ZN02a, ZN02b, Kuc03, Ara03, GSHL03, HRS+04, Zec03, ZM04] haben verschiedene Einschränkungen. Ein Ansatz mit virtuellen Maschinen [AH06] benötigt für drahtlose Netze zwei physische Netzadapter pro Knoten. Außerdem beschränkt erhöhter Speicherverbrauch durch strikte Trennung der Adressräume virtueller Maschinen die Skalierbarkeit.

Drei Anforderungen werden an Knotenvirtualisierung gestellt. Um unveränderte Prüflinge verwenden zu können, sollte sie *transparent* sein. Um verschiedene Ausführungsumgebungen anzubieten, sollte Knotenvirtualisierung *flexibel* sein. Weiterhin sollte sie *effizient* sein, um für eine gegebene Prüfstand-Hardware die

größtmögliche Anzahl Knoten zu ermöglichen.

Folgende Funktionseinheiten eines Protokollstapel sind zu virtualisieren: Rechnernetz-Geräte auf Sicherungsschicht, Leitwegtabelle auf Vermittlungsschicht, Kommunikationsendpunkte auf Transportschicht und Prozesse auf Anwendungsschicht.

Es gibt zwei grundsätzliche Ansätze, die für Knotenvirtualisierung in Frage kommen: Virtuelle Maschine (VM) [SN05] oder Betriebssystempartitionierung. Virtuelle Maschinen auf Systemebene, die ihren Gästen eine von der darunterliegenden Hardware unterschiedliche Befehlssatzarchitektur anbieten [Law96], sind zwar transparent und flexibel, aufgrund der notwendigen Emulation jedoch ineffizient. Virtuelle Maschinen auf Basis eines Wirtsbetriebssystems („hosted“ oder auch Typ II [Gol73]) [Law00, SVL01, KKL+07] stellen Gästen dieselbe Befehlssatzarchitektur wie die der Hardware zur Verfügung und sind transparent sowie flexibel jedoch eingeschränkt effizient aufgrund des beteiligten gastgebenden Host-Betriebssystem. Eine Sonderform solcher virtueller Maschinen, die nicht Hardware sondern beispielsweise die Systemrufschnittstelle des gastgebenden Betriebssystems virtualisiert [Dik00, SB02], ist transparent jedoch aufgrund der notwendigen Portierung des Gastbetriebssystems eingeschränkt flexibel und durch teure Kontextwechsel eingeschränkt effizient. Klassische VMs auf Systemebene („unhosted“ oder auch Typ I) [MS70, Wal02] kombiniert mit Paravirtualisierung [WSG02, BDF+03, Rus08] sind transparent und effizient jedoch eingeschränkt flexibel infolge der notwendigen Portierung von Gastbetriebssystemen auf die paravirtualisierte Systemschnittstelle. Als Form der Betriebssystempartitionierung bestehen virtuelle Protokollstapel aus partitioniertem Protokollstapel und Mehrprogrammbetrieb [DC01, SR02, Zec03, KHS+03, Leu04, ELK07, BBHL08]. Sie unterstützen unveränderte Anwendungen und Dienstprogramme zur Leitwegermittlung, erfordern jedoch Anpassungen an Prüflingen auf Transport- oder Vermittlungsschicht und sind daher nur teilweise transparent. Da virtuelle Protokollstapel nur ein Betriebssystem pro Prüfstandrechner unterstützen sind sie nur teilweise flexibel. Anders als bei virtuellen Maschinen sind keine zusätzlichen Kontextwechsel oder Kopiervorgänge im Speicher notwendig, was virtuelle Protokollstapel zum effizientesten Knotenvirtualisierungsansatz macht.

Unsere Architektur für ein skalierbares Emulationssystem besteht aus Software-Switch, virtuellem Protokollstapel und hierarchischer Emulationssteuerung. Der Software-Switch verbindet transparent sowohl Protokollstapel auf demselben Prüfstandrechner untereinander als auch Stapel auf verschiedenen Rechnern und trifft die Weiterleitungsentscheidung in sehr kurzer konstanter Zeit. Die Protokollstapelvirtualisierung basiert auf „Virtual Routing and Forwarding“ (VRF) in Version 0.100 von James R. Leu [Leu04], das erweitert wurde um anwendungstransparenten Zugriff auf Leitwegtabellen, Paketduplizierung an Benutzerprozesse

sowie ein Loopback-Gerät pro virtuellem Knoten. Hierarchische Emulationssteuerung verwendet zwei Zwischenprozesse auf jedem Prüfstandrechner, um hierüber schließlich auf den virtuellen Knoten Befehle auszuführen beziehungsweise Parameteraktualisierungen an Emulationswerkzeuge zu übermitteln.

Ein experimenteller Vergleich je eines Vertreters von Hosted VM (User Mode Linux [Dik00]), klassischer VM auf Systemebene mit Paravirtualisierung (Xen [BDF+03]) und unseres Prototyps virtueller Protokollstapel ergibt, dass Hosted VMs praktisch nicht skalieren und der virtuelle Protokollstapel zwei- bis viermal besser skaliert als die klassische VM [GWS06, MGWR07]. Unser Software-Switch trifft seine Weiterleitungsentscheidung in durchschnittlich 98 ns und erreicht einen Durchsatz von bis zu 3 GBit/s auf einem Intel Pentium 4 2,4 GHz Prozessor. Das Emulationswerkzeug [HR02] bildet Übertragungsratenlimitierung, Rahmenverzögerung und -verlustrate in einem weiten Wertebereich auch in virtualisierter Umgebung nach. Für drahtgebundene Szenarien mit 100 MBit/s Übertragungsrate ermöglicht unser Ansatz 11 mal größere Emulationsszenarien, für drahtlose Szenarien mit 11 MBit/s Übertragungsrate ohne Emulation der Medienzugriffssteuerung 28 mal größere Szenarien als ohne Knotenvirtualisierung.

Außerhalb des NET-Forschungsprojekts wurde das in diesem Kapitel beschriebene skalierbare Emulationssystem zu Evaluationszwecken in zwei Dissertationen erfolgreich eingesetzt [Sch07, Wac08].

4 Wettbewerb um Betriebsmittel in virtualisierter Emulation

Kapitel 4 untersucht die Erkennung von Wettbewerb um Betriebsmittel sobald zu viele virtuelle Knoten einen Prüfstandrechner überlasten und damit zu unerwünschten verfälschten Messergebnissen führen können. Es werden verwandte Arbeiten diskutiert, Qualitätskriterien zur Erkennung von Wettbewerb um Betriebsmittel hergeleitet und ein Ansatz zur Überwachung von Qualitätskriterien während Emulationsexperimenten vorgestellt. Sowohl die Wirksamkeit der Qualitätskriterien und ihrer Überwachung als auch die Effizienz der Überwachung werden anschließend evaluiert.

Verwandte Arbeiten zur Überwachung von Leistungsmessung in verteilten Systemen lassen sich in drei Kategorien klassifizieren: Rechnernetz-Überwachung, skalierbare Rechnernetz-Emulation und Instrumentierung von Programmtext. Rechnernetz-Überwachung kann zur Erkennung von Wettbewerb um Netzbetriebsmittel verwendet werden, indem auf potentielle Engpässe überwacht wird. Eine solche Überwachung kann entweder aktiv durch Senden von Testverkehr oder passiv durch reine Beobachtung stattfinden. Ansätze zur Ermittlung des Durchsatzes zwischen Endstationen basieren darauf, dass ein gesendetes Paketpaar

auf dem Pfad zum Empfänger am Engpass zeitlich auseinandergezogen wird und nach einer Reflexion zum Sender dort die Kapazität [Jac97] bzw. die verfügbare Übertragungsrate [HS03] abgeleitet werden kann. Da solche Ansätze aktiv vorgehen, könnte Netzverkehr der Prüflinge unerwünscht beeinflusst werden. Passive Netzüberwachung beobachtet Nutzlast, die meist auf Transportschicht eingeschränkt ist [SSK97, SKS00, ZL04, GD04, GZS+06], wodurch solche Ansätze nicht direkt auf die Emulation der Medienzugriffssteuerung übertragbar sind. Ansätze zur Abrechnung von Netzverkehr arbeiten ebenfalls passiv und basieren auf der Abtastung von Rahmen [JPP92, PPM01] im Gegensatz zu periodischer Abtastung. Solche Ansätze dienen vornehmlich der Erkennung von Verbindungen mit höchstem Durchsatz und können Auslastungsspitzen Aufgrund der Abtastung nur schwer erkennen. Eine weitere passive Klasse ist die Überwachung von Dienstgüte, die durch Abtastung oder ereignisgesteuert erfolgt [SB96], und im Gegensatz zur Erkennung stattdessen Wettbewerb um Betriebsmittel verhindert oder vermeidet. Rechnernetz-Überwachung kann als Prüfling oder zur Überprüfung der Konfiguration von Emulationsszenarien eingesetzt werden. Sie ist für die Überwachung virtualisierter Emulation nicht ausreichend, da beispielsweise der verfügbare Durchsatz durch den Software-Switch auch von CPU-Betriebsmitteln abhängt.

Skalierbare Rechnernetz-Emulation, die sich auf die Nachbildung von Netztopologien aus Punkt-zu-Punkt-Verbindungen beschränkt und die interne Verarbeitung von Verkehr gegenüber dem Annehmen von Verkehr außenliegender Prüflinge bevorzugt, kann Wettbewerb um CPU-Betriebsmittel einfach erkennen, sobald von Prüflingen eingehende Rahmen verworfen werden [VYW+02]. Separate sogenannte Eckknoten führen hierbei mehrere Instanzen von Prüflingen aus, wobei die Anzahl der Instanzen durch das Verhältnis aus Rechenleistung und Netzdurchsatz gesteuert wird. Dies erfordert jedoch die Vorabbestimmung einer Kennzahl für jeden Prüflingstyp und Prüfstandrechner. Netbed [HRS+04, HRS+08] basiert vergleichbar mit unserem Ansatz auf Protokollstapelvirtualisierung und überwacht mit Hilfe eines Benutzerprozesses CPU, Unterbrechungslast, Sekundärspeicher, Rechnernetz und Hauptspeicher. Da die Überwachung auf Abtastung basiert, können entweder kurze Lastspitzen kaum erkannt werden oder eine hohe Abtastrate stellt eine signifikante Zusatzlast dar, die Skalierbarkeit einschränkt.

Instrumentierung von Programmtext ermöglicht ereignisbasierte Überwachung mit geringem Overhead. Einfaches Profiling durch Zählen oder Zeitnehmen von Blockausführungen ist nicht feingranular genug für die Überwachung von Qualitätskriterien in virtualisierter Emulation. Detaillierte statische Instrumentierung von Programmtext kann manuell im Quelltext oder mit programmunterstützter Modifikation von Maschinenprogrammteilen [SE94, LS95] erreicht werden. Dynamische Instrumentierung von Betriebssystemen [TM99] wurde über die Zeit

weiterentwickelt, um sie durch Einsatz von Skriptsprachen [Moo01, CSL04] sicherer und schließlich durch Übersetzung der Skripte effizient [PCE+05] zu machen. Dynamische Instrumentierung liefert einen Mechanismus zur Überwachung. Das tatsächliche Überwachungsprogramm muss jedoch nach wie vor manuell erstellt werden.

Die Erkennung von Wettbewerb um Betriebsmittel soll *minimalen Aufwand* benötigen und *schnell* sein, sowie den *Anteil jedes virtuellen Knoten* am Betriebsmittelverbrauch bereitstellen. Es lassen sich drei Arten unterscheiden, um Qualitätskriterien zur Erkennung von Wettbewerb herzuleiten: Mathematisch, systematisch und empirisch. Aufgrund der Komplexität eines Emulationssystems ist die mathematische Modellierung aller relevanten Systemaspekte kaum möglich. Auch ein systematisches Verständnis des Systems kann nicht verhindern, dass wichtige Aspekte übersehen werden [Hoh05]. Daher liegt es nahe, empirisch vorzugehen und Spuren dynamischen Systemverhaltens mit Hilfe von Instrumentierung aufzuzeichnen und hieraus anschließend unabhängig von der Ausführung Qualitätskriterien abzuleiten. Netzbetriebsmittel sind überlastet, wenn der mittlere Durchsatz in einem sehr kurzen Zeitintervall einen Schwellwert überschreitet. Die CPU ist überlastet, wenn der Leerlaufprozess für eine zu lange Zeit nicht mehr aktiv wurde [Lep05]. Wettbewerb um Hauptspeicher besteht, wenn die Ein-/Auslagerungsrate in einem sehr kurzen Zeitintervall einen Schwellwert übersteigt. Sekundärspeicher ist überlastet, wenn die Warteschlange mit Ein-/Ausgabeaufträgen für eine zu lange Zeit nicht leer läuft.

Zur Überwachung der Qualitätskriterien während Emulationsexperimenten werden Ereignisse zur Einplanung von Betriebsmitteln im Betriebssystemkern instrumentiert. Dieser Ansatz kommt ohne zusätzliche Kontextwechsel oder Unterbrechungen aus und erkennt kürzeste Lastspitzen. Weiterhin ist die relativ grobe Standarduhrzeit im Kern ausreichend und es kommen nur schnelle und billige Berechnungsoperationen auf Basis von Addition und Vergleich von Ganzzahlen zum Einsatz. Für die vier Betriebsmittelklassen Netz, CPU, Hauptspeicher und Sekundärspeicher werden die Überwachungsalgorithmen in Pseudo-Code vorgestellt.

Um die Wirksamkeit der Qualitätskriterien experimentell zu evaluieren, wird für jede Betriebsmittelklasse eine gezielt ansteigende Last bis hin zur Überlastung generiert. Der Vergleich von Soll- und Istzustand zeigt, dass die Kriterien genau dann alarmieren, wenn Überlast vorliegt. Die Parameter und Schwellwerte für die Überwachung hängen ausschließlich von der Prüfstand-Hardware ab und sind unabhängig von den jeweiligen Prüflingen. Derselbe Experimentaufbau wird verwendet, um die Wirksamkeit der Überwachung zu evaluieren. Hierbei zeigt die Überwachung wie erwartet bei derselben Last Betriebsmittelwettbewerb an wie zuvor bei der Wirksamkeit der Kriterien. Zusätzlich kommen dieselben

Emulationsszenarien wie im vorhergehenden Kapitel 3 zu Knotenvirtualisierung zum Einsatz. Dadurch wird gezeigt, dass Wettbewerb auch für typische Szenarien zuverlässig erkannt wird. Die Messung des Overhead der Überwachung zeigt schließlich, dass dieser mit höchstens $\approx 0.0005\,\%$ vernachlässigbar ist und damit die Überwachung effizient arbeitet.

5 Schlussfolgerung

Kapitel 5 schließt diese Arbeit mit einer Zusammenfassung und einem Ausblick auf mögliche zukünftige Forschungsthemen.

Da Emulations-Werkzeuge Netzeigenschaften der Medienzugriffssteuerung nachbilden und Virtualisierung die Knotenanzahl in einem Emulationsszenario um Größenordnungen erhöhen kann, erhöht sich auch die Frequenz, mit der die Emulationssteuerung Wellenausbreitungsparameter für drahtlose Netze in Echtzeit auf Sekundärspeicher nachschlagen muss. Um vergleichsweise teures Lesen von Sekundärspeicher zu minimieren, können die vorberechneten Daten in einem ersten Schritt durch Selektion und Komprimierung reduziert werden. Weiterhin können die Daten für effizienten sequentiellen Lesezugriff angeordnet und indexiert werden. Ein Cache mit anwendungsspezifischer Ersetzungsstrategie kann dann alle möglichen Ausbreitungsdaten für ein bestimmtes Zeitintervall vorhalten. Erste Untersuchungen hierzu wurden in zwei studentischen Arbeiten durchgeführt [Dic07, Häu07].

Während die Prüfstandrechner in dieser Arbeit Einprozessorsysteme sind, geht der Trend in Richtung Mehrprozessorsysteme mit mehreren Kernen je Prozessor. Dies führt eine Ungleichmäßigkeit beim Zugriff auf gemeinsamen Hauptspeicher ein. Eine entsprechende Abbildung von virtuellen Knoten auf Prüfstandrechner kann dies ausnutzen, um virtuelle Knoten mit hohem Netzverkehrsaufkommen auf dasselbe System oder denselben Prozessor zu legen. Hauptspeicher kann zwischen virtuellen Knoten partitioniert werden, um teure Sperroperationen beim Zugriff auf den gemeinsamen Speicher zu vermeiden.

Skalierbare Emulation in Echtzeit erfordert einen Kompromiss zwischen Transparenz und Flexibilität der Knotenvirtualisierung auf der einen Seite und ihrer Effizienz auf der anderen Seite. Virtuelle Maschinenmonitore, die ihren Gästen virtuelle Zeit zur Verfügung stellen, ermöglichen skalierbare Emulation, die nicht in Echtzeit abläuft [GYM+06]. Durch Multiplizieren der realen Uhrzeit mit einem Zeitdehnungsfaktor erscheinen den Prüflingen Betriebsmittel wie CPU und Netz der Emulationsumgebung entsprechend leistungsfähiger. Dadurch können mehr Prüflinge auf Kosten der Gesamtexperimentdauer Arbeit in ihrer virtuellen Zeit verrichten. Als Ansatz zwischen konstantem Zeitdehnungsfaktor und paralleler Simulation auf Basis diskreter Ereignisse kann der Dehnungsfaktor epochenweise an den aktuellen Ressourcenverbrauch angepasst werden. Ein anderes Teilprojekt

des Forschungsprojekts „Network Emulation Testbed“ untersucht derartige neue Ansätze [GMHR08, GHR09].

Chapter 1

Introduction

1.1 Motivation

Continuing development of computer network technologies and corresponding application scenarios requires improved or new protocols on all layers of the protocol stack including distributed applications. This is true for classic wired networks such as the Internet as well as for recent wireless networks such as mobile ad hoc networks (MANETs).

On application layer, the new communication paradigm of peer-to-peer content distribution [ATS04] has become responsible for a significant share of the overall Internet traffic. The concept of overlay networks to build virtual network topologies on top of the existing Internet infrastructure is used in different fields of application, e.g. to improve wide-area routing using Resilient Overlay Networks (RONs) [ABKM01]. New services emerge with the proliferation of mobile communication such as a distributed event service [Bau07] to notify the user of interesting events that depend on the user's context, e.g. his location. Since mobile communication devices are usually powered by batteries, which represent a scarce power resource, it is essential to design energy aware protocols. In order to find new services in the network, an energy efficient service discovery [Sch07] is necessary.

On transport layer, new variants of congestion control for the Transmission Control Protocol (TCP) such as Binary Increase Congestion Control TCP (BIC-TCP) [XHR04] or FAST TCP [JWL04] support the large bandwidth delay product of fast large-distance communication with data rates of tens of gigabits per second. The increasing use of multimedia streaming in the Internet has given rise to new transport protocols such as the Datagram Congestion Control Protocol (DCCP) [KHF06]. Since TCP flavors designed for wired networks often perform suboptimal in wireless networks, different variants have been developed for wireless communication [Ela02]. Recent advancements include TCP with Adaptive Pacing

(TCP-AP) [EKL05] for pure wireless networks and TCP with Gateway Adaptive Pacing (TCP-GAP) [EKL06] for hybrid networks consisting of wired and wireless parts.

On network layer, advances in routing algorithms such as the Resilient Border Gateway Protocol (R-BGP) [KKKM07] improve the reliability of the Internet backbone. In addition to the well known routing paradigms link-state, distance-vector, and path-vector, a new technique called Failure-Carrying Packets (FCP) [LCR+07] supports delay and loss-sensitive applications such as telephony over the Internet using the Voice over Internet Protocol (VoIP). In contrast to the relatively static wired infrastructure of the Internet, wireless mobile ad hoc networks (MANETs) exhibit a very dynamic network topology due to the mobility of communicating nodes. This requires new routing protocols such as Dynamic Source Routing (DSR) [JM96] or Ad-hoc On-demand Distance Vector (AODV) routing [PR99] and corresponding implementations such as AODV from the Uppsala University in Sweden (AODV-UU) [LNT02].

The emerging technology of wireless mesh networks (WMNs) supports the idea of being connected to the Internet everywhere and requires protocol improvements on all layers of the protocol stack [AWW05].

The performance of improved or new protocols and distributed applications has to be evaluated during their development. Traditionally, three different methods have been used for performance evaluation during the course of development: mathematical analysis, simulation, and live testing. In early design stages, mathematical models can be used to predict the performance of the software to be designed [Jai91]. However, such models need to abstract from the reality and can usually only model certain aspects of the entire software system. It is hardly possible to develop models that reflect realistic properties of more complex software systems. In order to cope with such systems, simulation models allow for the algorithmic representation of more complex properties and environments. The execution of those models in simulator engines such as the network simulator 2 (ns2) [BEF+00] or GloMoSim [ZBG98] allows performance predictions in the next step of software development. The last step in the development process usually involves the creation of a prototypical implementation. This allows the execution of the actual software but requires an appropriate environment to obtain realistic performance evaluation results. Testbeds for live testing provide such environments. They can either be set up specifically for a certain isolated experiment or there are testbeds publicly available for research purposes. Testbeds can be designed for wired networks [BL03] and already exist in the form of RON [ABKM03] or PlanetLab [CCR+03, BBC+04, APST05]. There are also testbeds specialized on wireless networks such as the Open Access Research Testbed for Next-Generation Wireless Networks (ORBIT) [RKM+05, RSO+05].

After this last step of the software development, implementations need to be deployed. Since this can be difficult to accomplish in an already running environment such as the Internet, special platforms ease deployment of new services and protocols. Based on the experiences in building and running the PlanetLab testbed, a Virtual Network Infrastructure (VINI) [BFH+06] has been proposed as a step towards the long-term goal of a Global Environment for Network Innovations (GENI) [GEN06]. Such platforms can also be used to obtain performance evaluation results, that are representative for the current Internet environment [PHM06].

During the development of distributed software, it may be necessary to investigate certain isolated aspects of the implementation. However, not only other parallel users of a deployment platform influence measurement results but also the Internet environment to which the platform is connected. Hence, the lack of reproducible measurement results prohibits the use of public deployment platforms for such evaluation purposes. Isolated testbeds with exclusive user access allow reproducible results. Yet, extraordinary efforts for setup or financial restrictions for acquisition can prevent the assembly of such a testbed. If an isolated testbed requires geographically distributed nodes, administrative restrictions can also prevent a successful setup. In contrast to testbeds for wired networks, it is virtually impossible to obtain reproducible results in wireless testbeds due to the nature of the involved transmission of electromagnetic waves. Simulation supports reproducible results for all kinds of networks. However, the necessary simulation model usually has to be implemented for the specific environment of the simulator engine. Since such engines abstract from the corresponding real programming environment, the simulation model differs from a real implementation. Therefore, evaluation results can also differ between simulation and live testing.

Obviously there is a gap in the chain of evaluation methods between simulation and live testing. Computer *network emulation* fills this gap by supporting the execution of real implementations within a synthetic, configurable environment. For that purpose, a piece of software called *network emulation tool* reproduces specified network properties while communicating over flexible networking hardware possibly having different properties. A facility consisting of a combination of flexible networking hardware and suitable emulation tools is called *network emulation testbed*. Communication protocols or distributed applications that are subject to performance measurements in a network emulation testbed are called *software under test*. This includes software on application, transport, and network layer. We do not consider data link layer protocols since testing and performance evaluation of e.g. medium access control (MAC) algorithms requires very specialized evaluation environments, that emulate physical layer properties [HH02, SBBD03, JS03, VBV+05] and are out of the scope of this thesis.

Since network emulation reproduces the properties of a network environment synthetically, it supports reproducible measurement results for wired and wireless networks. Thus, emulation enables comparative performance measurements, i.e. comparing the performance of one implementation in different network environments or comparing the performance of different implementations in the same network environment.

Comparative performance measurements for mobile computing scenarios, e.g. the evaluation of an ad hoc routing protocol, typically require large scenarios with hundreds of nodes. The evaluation of new applications for traditional infrastructure-based networks, e.g. a large-scale location service, may also require a high number of nodes, since both the end systems and the intermediate systems of the underlying infrastructure have to be considered in a scenario model. Common network emulation systems used to assume that one communicating node in an emulation scenario corresponds to one physical computer in an emulation testbed. This severely limits the scalability, since testbeds with the required number of hundreds of computers are typically not available. However, a number of applications, e.g. in mobile computing scenarios, only need a fraction of the resources that a testbed computer can provide. Therefore, recent approaches employ node virtualization to partition each available testbed computer ("physical node," *pnode*) into multiple virtual nodes (*vnodes* [GSHL03]). Each communicating node in a scenario model is then mapped to one of the potentially many vnodes. The overhead of node virtualization determines how much resources of a pnode remain for the execution of the software under test. In order to achieve maximum scalability with a given testbed hardware, we need node virtualization with minimal overhead.

On each pnode, the software under test is executed in vnodes and shares the limited resources of their hosting pnode. Hosting too many vnodes leads to resource contention, which can cause unrealistic measurement results and is thus undesirable. In order to detect such resource contention, quality criteria for realistic network emulation need to be defined and monitored during an emulation run.

1.2 Contributions

The contributions of this thesis can be classified in two main topics: Node virtualization and resource contention in virtualized emulation. The following elementary contributions derive from these categories:

1. We discuss the entire range of resource virtualization approaches for computer systems and their suitability to scalable network emulation. Based on virtual protocol stacks we propose an architecture for *node virtualization* that

enables emulation of wired and wireless networks that scale in the number of communicating nodes.

2. In order for all the nodes to communicate with each other efficiently, we design and implement a *software communication switch* transparently connecting any virtual nodes on the same testbed computer as well as on different testbed computers. This switch makes its forwarding decision in extremely short constant time and forwards frames without any expensive copying of frame payload.

3. Some parameters, such as connectivity deriving from mobility of nodes with wireless network connection, are best maintained in a global network emulation model. To be able to cope with the potentially high volume of network parameter updates with many virtual nodes, we propose a *hierarchical emulation control*, that delegates distribution of dynamic parameter updates for emulation tools to all involved testbed computers.

4. Since virtual nodes on the same testbed computer share its limited resources, hosting too many virtual nodes may lead to resource contention and thus unrealistic emulation experiment results. In order to enable detection of resource contention, we define a set of *quality criteria* for all important resource classes comprising network, CPU (central processing unit), memory, and disk.

5. To observe our quality criteria during emulation experiments, we design and implement a highly *lightweight monitoring* approach. Monitoring works inside the operating system kernel sparing additional expensive context switches as opposed to user space approaches. Since we trigger monitoring by basic resource scheduling events, we can detect contention within a split second and with minimal overhead as opposed to sampling based monitoring which would require periodic timer interrupts causing context switches.

In combination, these contributions enable scalable network emulation of wired and wireless networks as well as arbitrary hybrid combinations such as mobile ad hoc networks with gateway connection to the Internet.

1.3 Structure

The remainder of this thesis is structured as follows. In the following Chapter 2, we describe the foundations of network emulation which form the basis to build our novel concepts and improvements on. The foundations comprise a discussion of

the architecture of a network emulation system and a description of our concrete implementation of a network emulation testbed.

As a major step towards scalable network emulation, we present node virtualization in Chapter 3. After an introduction, we discuss related work, define requirements for node virtualization, and present foundations of protocol stacks to show the entities, that need to be virtualized. Based on the discussion of different virtualization approaches, we develop our proposed node virtualization architecture. By means of extensive evaluations, we demonstrate the scalability of our virtualized network emulation system.

In order to support realistic measurement results with virtualized emulation, we discuss resource contention in virtualized emulation in Chapter 4. After an introduction, we review related work. We define quality criteria, that our presented lightweight monitoring approach can observe during emulation experiments to detect resource contention. Thorough evaluations show the effectiveness of quality criteria and monitoring as well as a negligible monitoring overhead.

We conclude this thesis in Chapter 5 with a summary including our main evaluation results and an outlook on possible future work arising from the insight of this thesis.

Chapter 2

Foundations of Computer Network Emulation

In this chapter, we discuss the general architecture of a network emulation system first. Based on the gained insight, we describe our concrete implementation of a network emulation testbed, which forms the basis for evaluating the novel concepts and improvements of this thesis.

2.1 Architecture of a Network Emulation System

Since network emulation is about the reproduction of network properties, we provide an overview of all protocol layers. For each layer, we derive its network properties from its specific tasks. An emulation system comprises a combination of three different software modules, for each of which we discuss different architectural alternatives. Based on the most promising architecture, we describe the functional parts of an emulation system. The classifications used in this section are adopted from a previous thesis [Her05] within the same research project.

2.1.1 Protocol Layers

Computers communicate over a network by means of protocols specifying message formats and sequences of control messages among others. In order to be able to manage the wealth of protocol functionality, computer networks are architecturally divided into protocol layers. There are two widely accepted layering standards. The International Organization for Standardization (ISO) defined the Open Systems Interconnection (OSI) model comprising seven layers. The nowadays common Internet reference model consists of four layers. It neglects two lesser important layers of the OSI model and also combines two important other layers. Therefore, a

hybrid layer model [Tan96] comprising five layers usually serves as a compromise for describing the concepts of computer networks. We also rely on this hybrid five layer model shown in Tab. 2.1.

tasks	**layers**	**network properties**
application specific	L5: application layer	—
addressing of services, connection handling, reliability, congestion control, flow control	L4: transport layer	message delay
globally unique node addresses, routing, forwarding, fragmentation, reassembly, connection handling, reliability, congestion control	L3: network layer	packet delay incl. reordering, packet drop
transmit frames between two directly connected nodes	L2: data link layer	
connection handling, reliability, flow control	(logical link control)	frame delay
coordinate exclusive access to shared medium	medium access control	medium access delay, frame drop
transmit stream of bits between two directly connected nodes	L1: physical layer	bit rate, propagation delay, bit error

Table 2.1: Five layer model according to [Tan96].

Each layer is responsible for carrying out specific tasks. To fulfill its duty, a layer builds on the services provided by the next lower layer and extends them. A layer provides its resulting service to the next upper layer. The behavior of a layer results in certain properties, that the next upper layer perceives through the service interface and either accepts or has to deal with. Network emulation can generally be done on any protocol layer. Emulating a certain layer means reproducing the properties implied by the emulated layer. This may include additional properties of each underlying layer whose effects are still perceivable through the emulated layer. Software under test can then be part of any of the layers building on the service interface of the emulated layer.

In the following, we discuss all layers from bottom to top. For each of them, we briefly introduce its tasks and the resulting network properties that would have to be emulated.

2.1.1.1 Physical Layer

The physical layer (L1) is responsible for transmitting a stream of bits from a sender to a receiver both of which are directly connected. This layer is usually implemented in hardware as part of a network adapter. Its properties include the transmission speed noted as bit rate, the delay until one sent bit reaches the receiver noted as propagation delay, and the quality of the communication channel noted as bit error rate (BER). The bit rate can be modeled as a delay queue and hence reduces to serialization delay and queuing delay. Additional delay may be caused by repeaters in the network. The communication channel can be described by a transmission function. In its simplest form it is a function of frequency and its value is a pair of signal amplitude and signal phase. As an approximation, the amplitude can be used to describe the signal attenuation, which is the ratio of transmission power to reception power. However, the receiver not only receives the attenuated signal of the transmitter but also noise, which makes detection of the original signal more difficult. It is the signal to noise ratio (SNR) that determines if the receiver is able to understand the transmission. Taking into account the coding of transmitted symbols allows to determine the bit error rate. That is sufficient for wired communication, where the transmission function is determined by the wire, and both the transmission function and the noise can be considered constant over time.

For wireless communication much more components influence the transmission function, since the signal propagation is not guided and possibly shielded as in the wired case. The wireless channel between sender and receiver is characterized by the propagation of radio waves through the environment. For the simplest propagation in free space, the distance between sender and receiver and the antenna gains of sender and receiver determine the signal attenuation. More realistic models also have to consider parameters such as the height of antennas above ground level and wave propagation obstacles that can cause diffraction, refraction, reflection and thus multipath propagation. If sender or receiver are mobile such as in a MANET, the parameters influencing channel quality obviously vary over time. The current position of sender and receiver within their surrounding environment becomes an influencing factor. For faster mobility speeds, the velocity also affects channel quality and thus the bit error rate of the physical layer.

An additional property of the physical layer, we do not focus on, are the possible communication directions: simplex, half-duplex, or full-duplex. In the following, we do not consider the nowadays uncommon half-duplex mode. Instead, we assume simplex mode or full-duplex mode, which can easily be modeled as two simplex channels in opposite directions.

2.1.1.2 Data Link Layer

The data link layer (L2) is responsible for the transmission of frames between two directly connected communication peers. Frames have limited size usually measured in units of octets or bytes respectively. The layer consists of two sublayers: medium access control (MAC) and logical link control (LLC).

The MAC layer only exists for networks with shared medium which is often the case for local area networks (LANs). All stations connected to such a network can hear each other's transmissions due to the broadcast property of the shared medium. Transmissions can only be received correctly, if at most one station transmits at any time. Since all stations share the same communication channel, they need to coordinate exclusive access to the channel. This is accomplished by a MAC algorithm, that can run in a distributed fashion on each station. It is usually implemented in hardware. More strictly speaking, the algorithm is executed on the network adapter independently from the surrounding computer. While it may often be implemented as software in the device firmware, we still consider it to be hardware to distinguish it from the software running on the computer's CPU (central processing unit). By coordinating exclusive medium access, the MAC layer introduces a medium access delay as its property. Additionally, there might be a switching delay, if a bridge or switch causes additional delays for frame delivery. MAC algorithms, that do not prevent frame collisions, may involve frame drops after excessive collisions. Network properties from the underlying physical layer, that still need to be emulated, are bit rate, propagation delay, and BER. Frame length together with BER can be converted into a frame error rate (FER). Since the data link layer usually uses a checksum to detect and ignore erroneous frames, the FER can be represented by a frame drop rate. The MAC service interface provides a connectionless and unreliable datagram service handling frames.

The LLC layer builds on the MAC service in order to extend it with additional features such as connection handling, reliability, or flow control. In currently used network technologies, LLC can hardly be found any more. It can always be implemented in software as part of the protocol stack and thus of the operating system in most cases. As such, it is the lowest protocol layer that can be considered as software under test on inexpensive standard computers. If present in a system, the important network property of LLC is additional frame delay in comparison to the MAC layer. This happens for the first frames to be sent if there is connection handling. Flow control delays frames to throttle transmission speed to the maximum processing speed of the receiver. Reliability is implemented by means of frame retransmissions, which result in delay. Apart from the possibly better quality of service, the service interface is comparable to the one of the MAC layer and handles frames.

2.1.1.3 Network Layer

The network layer (L3) introduces globally unique addresses for the nodes in a network. In contrast to end systems or hosts respectively, intermediate systems or routers respectively help interconnecting different network technologies. The units of communication are called packets on this layer. A router has two or more direct connections to neighboring networks. It uses routing to decide to which network it should forward packets it receives on any of the connections. The decision itself introduces packet delay as one of the properties of the network layer. Forwarding is usually implemented by enqueuing packets into a waiting queue which exists per outgoing network connection. The fill level and service rate of a queue cause additional packet delay. Since those queues are of limited size, packets have to be dropped as soon as queues are filled. This state is also known as congestion and leads to packet loss. Optionally, the network layer can take countermeasures by congestion control, which in turn leads to packet delay. If the destination network cannot handle packets of the size they were received by a router, they need to be fragmented and later reassembled both of which introduces packet delay due to some computation time. The routing decision is conducted independently for each packet and different packets may be sent along different paths through the network. Consequently, the travel duration of successive packets may be different leading to packet reordering, which is another kind of packet delay. The service interface of the network layer handles packets and may have different qualities of service: connection-less or connection-oriented, reliable or unreliable. The Internet protocol (IP) for instance provides a connection-less and unreliable service. Since it is widespread, we often focus on its properties in the remainder of this thesis without loss of generality.

2.1.1.4 Transport Layer

The transport layer (L4), enables the addressing of different services on a computer in a network. The units of communication are called messages on this layer. A transport protocol may provide either a connection-less or a connection-oriented service. Connection-less transport protocols usually build on a connection-less network layer and an emulation would mainly need to consider properties of the underlying layer such as packet delay including reordering and packet drops. An example for such a protocol would be the user datagram protocol (UDP) which is used on top of IP. Connection-oriented transport protocols are usually also reliable and include flow control. They may also include congestion control, especially if the network layer does not handle congestion itself. Connection handling causes delay for the first messages sent. Reliability takes care of transmission errors and packet drops by retransmitting messages and it takes care of packet reordering by

resorting messages before delivery. Both measures for reliability result in variable message delay. Flow control and congestion control add additional variable message delay. An example for a connection-oriented transport protocol implementing all of the above-mentioned services would be the transmission control protocol (TCP) which is used on top of IP.

2.1.1.5 Application Layer

The application layer (L5) is part of the user space. In contrast to this, the LLC layer, network layer, and transport layer are usually part of the kernel space or operating system respectively. The application layer makes use of operating system services through the system call interface. With respect to networking, the BSD Socket application programming interface (API) is nowadays commonly used to mainly access services of the transport layer. There is no point in emulating network properties of the application layer, since it is the highest layer in the protocol stack without any upper layers using its service and thus perceiving its properties. At least this layer is to be considered as software under test.

2.1.1.6 Summary

From the viewpoint of somebody evaluating the performance of distributed applications or communication protocols, it is desirable to consider as many layers of the protocol stack as possible as software under test. This allows for greatest flexibility of a network emulation system. Emulating a low layer of the protocol stack also has the advantage that only few and simple network properties have to be reproduced. The higher the emulated layer, the more complicated emulation models are required to reproduce all properties that are perceived at this layer's service interface. Emulating the physical layer requires special testbeds [HH02, SBBD03, JS03, VBV+05] that consist of specific customized hardware in contrast to inexpensive commodity personal computers. The MAC layer is the lowest layer that can be emulated in software on inexpensive standard computers as has been shown by previous works [HMR03, MRBV04]. Implementing emulation as software on standard computers is important to support large evaluation scenarios by enabling the purchase of maximum hardware resources at an affordable price. As already mentioned in the previous section, the application layer is the highest layer that can be considered as software under test. Therefore, we consider the following layers as possible software under test: LLC, network, transport, and application layer. If only higher layers need to be evaluated, existing standard implementations of the remaining lower layers can simply be used to build a complete protocol stack.

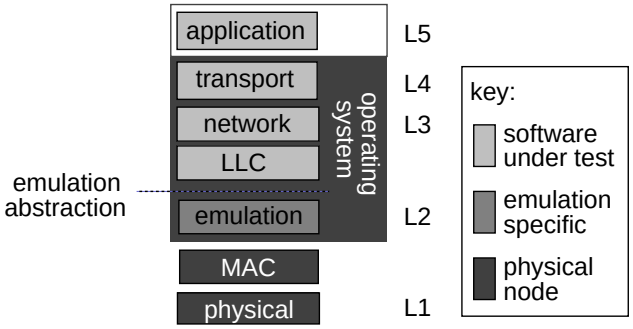

Figure 2.1: Emulation layer in the protocol stack.

In order to support performance evaluation of the above-mentioned software under test, we have to reproduce the network properties of the data link layer—or MAC layer in particular—and provide the corresponding service interface. There are two basic properties to emulate: frame delay and frame drop. All other possible properties can be reduced to those two basic properties. The service interface handles frames and is connection-less and unreliable.

The interface is implemented by a network emulation tool (Fig. 2.1). In order to communicate with tool instances on other computers, it makes use of a reliable high-speed low-latency network, that is part of the network emulation testbed hardware. The tool reproduces network properties by adding additional frame delay and dropping frames accordingly. Modeling of e.g. the frame propagation delay can be implemented by means of a delay queue. For each frame, the tool calculates the transmission time in the future by adding the propagation delay to the current time. Each frame is enqueued together with its transmission time. A timer is set to the transmission time of the frame at the delay queue head. On triggering of the timer, the frame gets dequeued and transmitted, and the timer is set to the transmission time of the frame that is now at the queue head. More details on the modeling of emulated network properties follow in Section 2.1.3.2 and 2.1.3.3 describing two classes of emulation tools.

2.1.2 Emulation Modules

After having decided on the optimal protocol layer to emulate in the previous section, the interface for emulation is sufficiently specified. The abstraction of the real network, that is to be emulated, is represented by a network model. The model serves for maintaining the configuration and parameters of a network emulation scenario. Based on the model's content, network emulation tools are configured to reproduce the necessary network properties. This configuration process is done by a controlling instance. An emulation controller is especially necessary

for configuring dynamic network parameters, that change over the course of an experiment. The emulation of mobile wireless networks is a notable example involving high dynamics of network parameters.

Hence, software for a network emulation system consists of three software modules: network model, emulation tool, and emulation control. We discuss the architectural alternatives for each of those modules and then present sensible combinations. For the discussion, we assume that there are as much testbed computers available as there are nodes in a network to emulate. Additionally, each computer has as much physical network links as necessary to configure the topology of the network scenario. Each pair of nodes is connected by at most one network link.

2.1.2.1 Network Model

The network model represents an abstraction from a real network. It is the basis for emulating the relevant properties of the corresponding real network. A user of an emulation system configures the model by means of a scenario specification. Such a specification contains static as well as dynamic network parameters. An emulation control instance evaluates derived parameter settings and maintains them in the model. For dynamic parameters, the controller continually computes them during the course of an experiment.

Two alternatives exist for the architecture of the network model. First, a global network model contains all information in one *centralized* place. Secondly, the model can be split into one centralized global model part and one self-contained model part on each testbed computer. We name this second architecture a *hybrid* network model since it consists of one centralized and multiple distributed model parts. Emulation control maintains the global part and updates the distributed model parts on each testbed computer. Each of the distributed model parts is used to directly configure the corresponding local emulation tool instance.

2.1.2.2 Emulation Tool

An emulation tool is responsible for reproducing network properties. The network model specifies the necessary parameters for fulfilling this task.

Three alternatives exist for the architecture of emulation tools. A single global network model can configure a single *centralized* emulation tool. Distributed tool proxies in all the protocol stacks redirect the entire network traffic through the central emulation tool. This redirection considerably increases the minimal frame delay that can be emulated, because each frame has to be sent from the source to the central tool and then again from the tool to the final destination. Since the central tool has to process all traffic, it may also constitute a bottleneck limiting

the scalability for the emulation of larger network scenarios. Therefore it stands to reason to distribute multiple emulation tools among the testbed computers. We distinguish two variants of distribution. First, as an intermediate step, a separate tool instance can be responsible for each *network segment* in the scenario. The tools are *distributed* such that each tool is executed on its own testbed computer. On the one hand, each tool only has to process the traffic of the collision domain it is responsible for. On the other hand, an additional exclusive testbed computer is required for the execution of each tool instance. Secondly, a separate tool instance can be responsible for each network link in the scenario. This way, each tool only has to process the traffic relevant to one computer. For networks with shared medium and more than two connected computers, this reduces the amount of traffic to process further. The tool instances can be *distributed* such that each tool instance can be executed on the same computer that the *network link* belongs to, so no additional testbed computers are required.

2.1.2.3 Emulation Control

Emulation control maintains current parameters in the network model and configures emulation tools with those parameters. For dynamic parameters this is carried out continually.

Control can be designed either centralized or distributed. *Centralized* emulation control has access to the global part of the network model. Based on the model and possibly the actual emulation time, control configures all emulation tools with current parameters. This requires network communication, if the tools are distributed. For highly dynamic scenarios this communication might become a bottleneck. *Distributed* emulation control avoids such communication completely. Therefore, the temporal sequence of parameter changes including timestamps gets computed in advance and distributed. Then each distributed control instance independently reproduces the parameter changes by means of its synchronized local clock during the course of an experiment. Computing the sequence in advance, however, allows less flexibility during an experiment since all dynamics are predetermined.

2.1.2.4 Alternatives of Distribution

Not all arbitrary combinations of the aforementioned architecture alternatives are sensible. A centralized emulation tool obviously implies a centralized network model and a centralized emulation control. A hybrid network model can be combined with any of the two distribution variants of tools and either central or distributed control. This results in five different alternatives for distribution enumerated in Tab. 2.2.

alternative	model	tool	control
1	central	central	central
2	hybrid	distributed, per network segment	central
3	hybrid	distributed, per network segment	distributed
4	hybrid	distributed, per network link	central
5	hybrid	distributed, per network link	distributed

Table 2.2: Comparison of distribution alternatives.

The completely centralized architecture of alternative 1 constitutes a bottleneck limiting scalability for larger network scenarios. Alternatives 2 and 3 require one additional testbed computer for each network segment in the scenario to execute one emulation tool instance per computer. This also limits scalability for a given testbed. While alternative 5 may be the most scalable solution, it has been shown that centralized control is sufficiently fast even for the emulation of mobile wireless networks [Her05]. Also, centralized control allows dynamic changes to emulation model parameters during an experiment without predetermined calculation in advance. Hence, we focus on alternative 4 in the following discussions. As depicted in Fig. 2.2, it comprises hybrid model, distributed tools with one instance per network link, and centralized control.

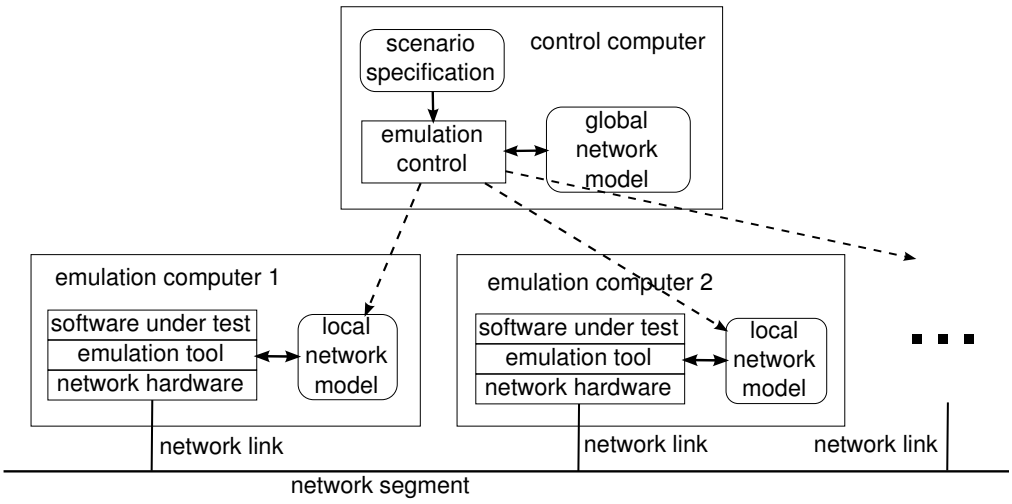

Figure 2.2: Chosen distribution of emulation modules [Her05].

2.1.3 Emulation Functionality

For the MAC layer identified as emulation interface in Section 2.1.1 and the most suitable emulation system architecture from Section 2.1.2, we present the

functionality of such an emulation system in this section. First, we describe the centralized emulation control with the global network model. Secondly, we give an overview of the functionality of network emulation tools. We distinguish between tools for non-shared communication media such as a point-to-point link and tools for shared media such as a wireless LAN.

2.1.3.1 Emulation Control

Input to the centralized emulation control is a scenario description. The description contains static information about the properties of the networks to emulate. It can also contain compute directives for dynamic information. Emulation control guides an emulation experiment through its three phases consisting of preparation, measurement, and cleanup. It sets up the global network model in the first phase and maintains the model during the second phase based on the scenario description and the advancement of experiment time. More details can be found for scenario specification in a previous publication [HLR02], for experiment phases and the global model in a thesis [Her05], and for maintaining dynamic model parameters in [HR02, HMTR04, Her05].

Experiment Phases Each emulation experiment involves a sequence of three phases: preparation, measurement, and cleanup.

Preparation is used to allocate testbed resources. This includes one testbed computer for each node in the scenario specification. The testbed hardware is configured with the requested network topology. The emulation tools are initialized with their first parameter set. Then, it is possible to start and warm up software, that is required for the experiment but is not directly related to the actual measurement. All involved distributed testbed computers synchronize on a barrier at the end of their preparation phase, in order to start the measurement at the same time.

During the measurement phase, the distributed emulation tools reproduce the specified network properties independent of the emulation control. Whenever parameters change in the global network model, emulation control reconfigures the corresponding tools by means of update messages.

After the measurement is finished, emulation control activates the clean up phase. This happens when either a specified measurement duration has elapsed or some algorithm has detected a suitable termination of the software under test. If not done already, control terminates all started software in opposite order and undoes the configuration of the testbed. Subsequently, measurement results can be collected from the testbed computers, possibly merged, and evaluated.

Global Model As mentioned in Section 2.1.1.6, network emulation tools only need to reproduce the two basic network properties which are frame drop and frame delay. A scenario description may also contain higher level parameters. The emulation control has to compute the basic properties from higher level parameters. It is reasonable to separate computation in the control instance from the reproduction of basic properties in the emulation tools. However, in our chosen architecture some input parameters for certain computations are only available at the distributed emulation tools. Therefore, we use a hybrid approach where computation is distributed among control and tools.

Frame drops are determined by the FER and caused by the MAC layer in case of excessive collisions or in case of transmission failures detected by a frame checksum. The FER p_f can be computed from the frame length l and the BER p_b as follows: $p_f = 1-(1-p_b)^l$. Only an emulation tool knows about the current frame length or excessive collisions, so those parameters cannot be part of the global model. However, the BER is either constant for wired networks or can be computed by the emulation control for wireless networks and hence becomes part of the global model.

Frame delays consist of serialization delay, propagation delay, and medium access delay. Serialization delay t_s for a single frame can be computed from frame length l and bit rate b as follows: $t_s = l/b$. Since only a tool knows about the current frame length, serialization delay cannot be part of the global model. However, the bit rate is usually specified by the scenario specification and hence becomes part of the global model. Propagation delay t_p is either specified in the scenario specification for wired networks or control can compute it for wireless networks. Thus, propagation delay is also part of the global model. Medium access delay is dynamically determined by the MAC algorithm and depends on the network load among other things. Since only a tool knows about those influential parameters, medium access delay cannot be part of the global model.

BER p_b, bit rate b, and propagation delay t_p are part of the global network model. Each of those parameters can be different for each connection direction or each pair of nodes in the scenario description respectively. With n nodes, the global model is represented by an $n \times n$ matrix having the 3-tuple $e = (p_b, b, t_p)$ as elements.

2.1.3.2 Emulation Tool for Non-shared Communication Medium

On a testbed computer, one instance of an emulation tool is executed for each local network link in the scenario. The tool can modify local network traffic in both transmit direction and receive direction. Modification is done according to the basic properties frame drop and frame delay. The tool computes the basic properties from the parameters in the global model. The local network model for all emulation tool

instances on one testbed computer contains a relevant subset of the global model. Since a tool reproduces basic properties in transmit and receive direction, only the row $\{e_{ij}|i = k\}$ and the column $\{e_{ij}|j = k\}$ of the global model matrix are relevant to the tool instance on node k.

We focus on the emulation of networks with non-shared communication medium such as wired networks first and describe tool functionality for other types of networks in the next section.

The tool emulates frame drops based on the BER and the individual frame length. Unicast transmissions to any destination node d can be efficiently dropped at the source s taking the BER p_b of the matrix element e_{sd} into account. Drops for broadcast or multicast transmissions have to be decided for each pair of source s and destination d_j with $j = [1..n]$ separately. In order to take advantage of possibly efficient broadcast delivery mechanisms of the testbed networking hardware, the tool on each destination node has to come to a drop decision for the incoming transmission. In either case, the tool generates a new random number R with $0 \leq R < 1$ and drops the frame iff $R < p_f$.

Frame delay consists of medium access delay, propagation delay, and serialization delay. Since we assume an non-shared communication medium in this section, there is no medium access delay. The tool can take the propagation delay t_p directly from the local model and add it to the serialization delay. Frame length and bit rate determine the serialization delay for a single frame. However, the bit rate and propagation delay determine the capacity of the communication channel. The capacity is also known as the "bandwidth-delay-product." Since the emulation tool models the communication channel, it also has to model the channel capacity. A delay queue is used to hold frames up to a total size corresponding to the channel capacity. The tool instance also maintains an internal queuing time t_q. If the queuing time is behind the actual time t, the delay queue has become empty and the queuing time is set to the actual time: $t_q = t$. Otherwise, propagation delay and serialization delay are added to the queuing time to allow for queuing delay: $t_q = t_q + t_p + t_s$. The frame then gets stored in the queue. A timer is set to trigger the removal and final transmission of the frame at time t_q.

Frame delay can be implemented on the source or the destination side. Delaying at the sender is easier to implement and valid if the differences in propagation delay to different receivers can be neglected. The inherent small frame delay of the low-latency testbed network can be considered in the calculations for emulation delay. We refer the reader to a previous publication [HR02] and a thesis [Her05] for more details.

2.1.3.3 Emulation Tool for Shared Communication Medium

We now describe the functionality of an emulation tool for networks with a shared communication medium such as WLAN (wireless local area network). The presence of a MAC algorithm influences the medium access delay. This needs to be considered in addition to the tool functionality for non-shared medium presented in the previous section. Our approach is to reproduce the MAC algorithm in software as part of an emulation tool. A single tool reproducing all classes of algorithms is hardly possible. We distinguish the following three classes of MAC algorithms: collision-free protocols, limited-contention protocols, and random access protocols. The class of collision-free protocols includes static channel multiplexing such as time-division (TDMA), frequency-division (FDMA), or code-division multiple access (CDMA). Such static multiplexing can already be emulated sufficiently with the previously described emulation tool for non-shared communication medium. From the two remaining classes, random access protocols include the widespread concept of carrier sense multiple access (CSMA). With the proliferation of WLANs according to the IEEE standard 802.11 [IEE99a, IEE99b, IEE01], the subclass CSMA with collision avoidance (CSMA/CA) has become of great interest for performance evaluations, e.g. in MANET scenarios. Therefore, we focus on the basic functionality of a tool for emulating the class of CSMA algorithms.

Our local network model contains the same row and column of the global model as described in the previous section. Each element e_{sd} of this sparse submatrix also consists of BER p_b, bit rate b, and propagation delay t_p. We extend this basis by the concept of a virtual carrier.

The high-speed low-latency network of an emulation testbed is usually switched. This implies that only addressed stations receive transmissions. In order to establish an emulated broadcast domain typical for a shared media, our emulation tool encapsulates each frame into a broadcast frame before transmission. This way, each station is able to overhear all transmissions.

Since we use a low latency emulation network, its transmission delays are much less than the delays to be emulated. We thus neglect transmission delays on the emulation network and simplify the discussion here. This allows us to time stamp incoming frames with an un-synchronized local clock t and assume that this is the point of time when a frame was sent.

We give an overview of the virtual carrier functionality depicted in Fig. 2.3. A station (S1), whose MAC algorithm has decided to transmit a frame, sends that frame out immediately. Since a carrier obviously started, the sending station remembers this state by setting a local variable t_b for the beginning of a virtual carrier to the actual time: $t_b = t$. The prospected emulated end t_e of the virtual carrier is calculated as the time of carrier start plus the serialization delay: $t_e = t_b + t_s$. All other stations (S2 and S3) receive the frame almost immediately and

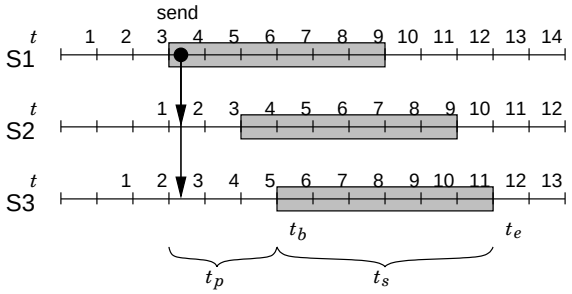

Figure 2.3: Space time diagram of virtual carrier sensing method.

calculate the beginning of the frame's virtual carrier by adding the corresponding propagation delay to their local time: $t_b = t + t_p$. The end of the virtual carrier is obtained in the same way as for the sending station. Having this information, each station is able to sense the virtual carrier, if for its local time $t_b \leq t < t_e$ holds true. The overlap of virtual carriers connotes a collision. A receiving station delivers the frame at time t_e, if the frame has not been involved in a collision.

Based on the local clock and the time stamps of all overheard frames, each station is able to autonomously maintain a model of the shared medium state. The state consists of the two points of time for the beginning t_b and the ending t_e of a virtually sensed carrier. We add those two state variables to the local network model. This provides our basis for the emulation of MAC algorithms using carrier sensing.

On top of the virtual carrier model, a MAC algorithm can be implemented in software. Such an implementation is usually based on a finite state machine. State transitions are triggered by timers, transmission of frames, or reception of frames. The reproduction of the MAC algorithm allows for emulated medium access delay, which is dynamically dependent on the network load. Existing algorithms such as IEEE 802.11 [IEE99a] using CSMA/CA or IEEE 802.3 [IEE05] using CSMA with collision detection (CSMA/CD) can be implemented this way. For more details, we refer the reader to a previous publication [HMR03], two diploma theses [Mai02, Yan04], and a PhD thesis [Her05].

Since we transmit each frame encapsulated in a broadcast frame to enable virtual carrier sensing, an emulation tool has to drop frames at the destination. This works as described in the previous section. During the emulation of WLAN, the distinct BER for each pair of source and destination reflects the current connectivity between wireless stations. Central emulation control determines the individual BERs considering a suitable physical layer model and node mobility. Updated BER values are distributed to the emulation tools. More details can be found in a previous publication [HMTR04], a diploma thesis [Dud02], and a PhD thesis [Her05].

2.2 The "Network Emulation Testbed"

Having discussed different approaches to the architecture of a network emulation system in the previous section, we now describe our implementation of the chosen architecture. Our Network Emulation Testbed (NET) uses a hybrid network model, distributed emulation tools with one instance per link, and a centralized emulation control. It provides the basis for evaluating the novel concepts and improvements of this thesis. In the following, we describe the involved hardware, the implementation of our emulation tools, and the implementation of the central emulation control with its global network model.

2.2.1 Hardware

2.2.1.1 Computers and Networks

The Network Emulation Testbed consists of 64 inexpensive common off the shelf (COTS) personal computers (PCs) (see Fig. 2.4). Those computers are connected by a Fast Ethernet network used for administration and emulation control messages. A dedicated high-performance server computer executes the centralized emulation control software, which uses the control network to send parameter updates of the global network model to the distributed emulation tools on the PCs. A separate network connection to the control server makes the testbed accessible from other networks for administration and control purposes. For actual network traffic of an emulation experiment, the PCs are connected by a monolithic, programmable Gigabit Ethernet switch. Having a separate exclusive network for emulation traffic prevents interference with administration or control traffic. A connection between control server and emulation network switch is exclusively used for configuration purposes during the preparation experiment phase. The control server neither processes nor directly influences any network traffic of an emulation experiment.

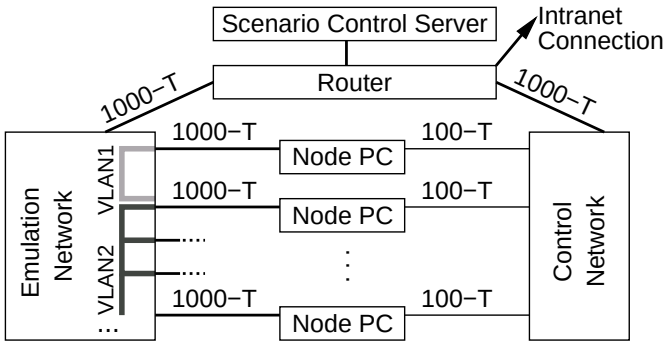

Figure 2.4: The Network Emulation Testbed.

2.2.1.2 Connection Topology

As discussed in Section 2.1.3, the BER values p_b in the global network model are used to emulate the communication channel quality between each pair of nodes in an emulation scenario. Emulation efficiency can be optimized, if the emulation network hardware is able to filter traffic. This relieves emulation tools from processing frames, that would be dropped anyway. Since configuring networking hardware takes a considerable amount of time [Her05], it is hardly usable for dynamic aspects of communication channels between nodes. Therefore, we leave the reproduction of dynamic aspects to the emulation tools and concentrate on static aspects here.

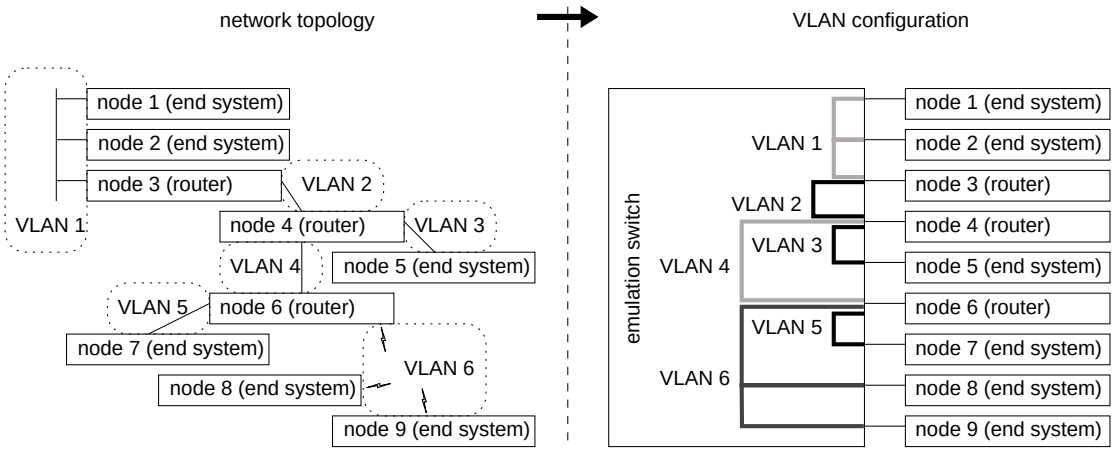

Figure 2.5: Network topology emulation with tagged VLANs.

A node s is potentially able to successfully transmit to another node d, if $p_b < 1$ where p_b is from the model tuple e_{sd}. The set of such channels forms the network topology represented by a directed graph. Using IEEE 802.1Q VLAN (virtual LAN) technology [IEE03], the emulation switch is able to create a connection topology represented by an undirected graph (see Fig. 2.5). All edges of the given directed topology graph can be mapped to edges of the undirected graph supported by the hardware. If a channel between two nodes is not symmetric, the emulation tool on the node, where no edge originates, can efficiently drop frames before transmission and thus ensure the asymmetry. A switch only forwards frames to ports, and thus the connected PCs, that are part of the same VLAN identified by a tag. We configure the switch during the preparation experiment phase. Each point-to-point link or shared media network segment in an emulation scenario, e.g. a WLAN channel, is mapped to a uniquely tagged VLAN.

2.2.2 Emulation Tools

2.2.2.1 Virtual Network Devices

In scenarios where a node has multiple network links, it is necessary to assign multiple VLAN tags to the same emulation switch port. On a testbed node, each link to a tagged VLAN is represented by a virtual network device. On transmission, each virtual device extends the frame header with the corresponding VLAN identifier before actually transmitting over the physical interface. On reception, testbed nodes use this information to demultiplex frames to the responsible virtual network device. Virtual network devices for tagged VLANs can be dynamically configured in software. This flexible mechanism allows multiplexing of individual virtual network links over a single physical link to the emulation network. Thereby, we are able to fulfill our earlier assumption from Section 2.1.2, that each testbed computer has as much network links as necessary to configure the topology of the network scenario.

2.2.2.2 Integration into Protocol Stack

In Section 2.1.1.6, we identified the MAC layer as the layer of the protocol stack a network emulation tool should implement (see also Fig. 2.6). The service interface handles frames and is connection-less and unreliable. Network device drivers in common operating systems implement exactly this interface. We use Linux as an example open source software allowing modifications at the kernel. For the discussion of details such as function names, we assume version 2.4.24 of the Linux kernel. Although being strictly speaking a monolithic kernel, Linux is modular especially with respect to device drivers. Therefore, we decided to implement our emulation tools as dynamically loadable kernel modules. Since they do not actually drive any physical network adapter hardware, they are called virtual network

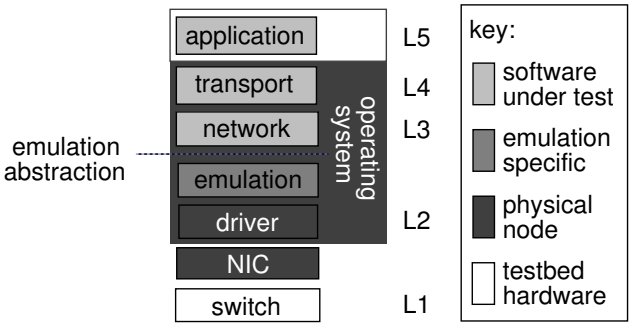

Figure 2.6: Software under test and network emulation tools on a physical node in NET.

device drivers. Our tools are entirely transparent to implementations on the LLC or network layer respectively and thus to our intended software under test. In order to implement its emulation functionality, a tool has to be integrated into the protocol stack's data path in both transmit and receive direction.

2.2.2.3 Integration in Transmit Direction

The network device driver interface specifies a function for transmitting frames. In Linux, the function is called hard_start_xmit(). Our emulation tools implement this function and register their implementation on kernel module load time. If a frame has been delayed appropriately and should not be dropped, our tools transmit the frame by handing it over to a real network device using the function dev_queue_xmit(), which will finally call hard_start_xmit() of that real device. Our tools also inform the protocol stack whether the emulated network is saturated, if the tool is busy transmitting a frame, or not. This is accomplished using the functions netif_stop_queue() and netif_wake_queue() of the network driver subsystem in Linux.

2.2.2.4 Integration in Receive Direction

Receiving frames happens asynchronously. Therefore, a real physical network adapter signals frame reception by means of an interrupt request (IRQ). The CPU then executes an interrupt service routine (ISR) for handling such an asynchronous event in software. The device driver for the adapter implements an appropriate ISR, in which it appends the received frame to the input queue of the protocol stack. In Linux, network drivers use the function netif_rx() for enqueuing.

Actual processing of the received frame happens later on in the receive software interrupt, which is scheduled before the return from system calls or at the next operating system timer interrupt. The receive software interrupt takes received frames from the input queue and demultiplexes the frames to the responsible network layer protocol entity for processing. The protocol identifier of the frame header is used for demultiplexing.

Network protocol demultiplexing is one possible place to hook an emulation tool into the receive path of frames. Therefore, the tool registers with the protocol stack for an otherwise unused network layer protocol identifier in order to exclusively receive all matching frames. Registration is done with dev_add_pack() in Linux. With this approach, a tool appears to the protocol stack in disguise as a pseudo network layer protocol entity. This only works, if all relevant frames are sent with the chosen protocol identifier, which is generally not the case. However, such an approach is suitable for an emulation tool for networks with shared medium. As described in Section 2.1.3.3, frames get encapsulated into a broadcast frame before

transmission. This can be used to assign the correct protocol identifier. Since the encapsulation prefixes the frame with an additional frame header carrying the broadcast address and chosen protocol identifier, the original destination address and protocol identifier remain unmodified. On receiving such a frame, the responsible emulation tool strips the encapsulation header, delays the frame accordingly, and can then reinsert the original frame into the protocol stack by calling netif_rx(), if the frame should not be dropped on the receive side.

In Linux, the function netif_receive_skb() takes frames from the input queue for further processing inside the receive software interrupt. The load of receive IRQs for high-speed networks, such as the Gigabit Ethernet used in NET, can prevent the CPU from processing other useful things including software interrupts. The effect is the so-called receive livelock [MR96]. In order to mitigate the IRQ load, different approaches reduce the number of receive interrupts. Linux uses adaptive polling [SOK01], which is also executed by the function netif_receive_skb().

Since all received frames pass this function, it is another place to hook an emulation tool into the receive path. However, no suitable hooking interface is provided by Linux. Therefore, we patch netif_receive_skb() with our own hook mechanism placed before the network layer protocol demultiplexing. Our emulation tool for networks with non-shared communication medium uses this hook to process incoming frames.

2.2.2.5 Configuration Interface

The reception of dynamic parameter updates from the central emulation control requires blocking network I/O. Blocking is usually restricted to process context and hardly feasible to implement directly in the emulation tool. Therefore, we implement a user space daemon receiving parameter updates over the network using either UDP or TCP on each testbed computer. On reception, the daemon passes the updates to the local emulation tool instances. There are various ways of passing information between user space and kernel space. In Linux, there is a choice [RC01] among IOCTL (I/O control), custom character devices, virtual file systems—such as system file system (sysfs) or proc file system (procfs)—, or netlink sockets [SKKK03]. For our prototypical implementation, we choose the classic IOCTL, which has historically already been used for configuring different networking aspects, for which no other specific configuration interface has been designed.

2.2.3 Emulation Control and Global Network Model

Centralized emulation control maintains the global network model and it distributes parameter updates to the emulation tools as described in the previous

section. The global network model contains static and dynamic parameters. Static parameters do not need any maintenance and can easily be distributed during the preparation experiment phase. Diverse dynamic parameters can be part of the global model. Exemplary, we discuss the maintenance of dynamic parameters for the emulation of a MANET.

We describe the scenario of a MANET by means of a trace of node movements, a geographical model of the scenario environment, and technical information on the wireless hardware. For simplicity, we assume that the bit rate b is constant and the propagation delay t_p is negligible due to the relatively short communication range of 450 m maximum with WLAN. Hence, only the BER p_b needs to be maintained as a dynamic parameter.

Using the trace of node movements, the mobility of nodes can be simulated to obtain the node positions at any time during an emulation experiment [HMTR04]. Considering actual node positions, the geographical environment, and technical information about the wireless hardware the propagation of radio waves can be simulated. The different possible propagation models can be classified into those ignoring the geographical environment and those considering the environment. The free space model [Fri46] or the two-ray ground model [Rap01] ignore the environment. A combination of both models is typically used in MANET simulation [BMJ+98, FV02]. However, a realistic propagation model has impact on the performance evaluation and is thus very important [TMB01, SHR05, Ste09]. For considering the geographical environment, there are two subclasses of propagation models: empirical models [Com91] or deterministic models using ray tracing [Lan99, WHL99]. For either model, the calculation of attenuation values is not possible in real-time during an experiment. Consequently, we compute the values for all pairs of sender position and receiver position on a discrete two-dimensional grid of $5 \times 5\ \mathrm{m}^2$ once in advance [SHR05, Her05, Ste09]. It stands to reason to use an algorithm such as "Intelligent Ray Tracing" [WHL99] since it computes the attenuation to all possible receiver positions for a fixed sender position in one step. Due to its considerable size, the resulting data is stored on an external RAID (redundant array of inexpensive disks) connected to the control server. Emulation control periodically reads the necessary attenuation values to obtain the BER between all pairs of nodes. For our initial implementation, we use a signal attenuation threshold for deciding if a frame gets dropped ($p_b = 1$) or received correctly ($p_b = 0$). Of course, more sophisticated models for obtaining the BER as described in [TMB01, ABCG04] can easily be integrated.

2.2.4 Application Case Studies

A previous thesis [Her05] within the same research project contains more details on the implementation of the testbed and its software. For additional details,

especially concerning the implementation, we refer the reader to diploma theses [Dud02, Mai02, Yan04] and a study thesis [Cas05]. For a practical description of how to use the Network Emulation Testbed, we refer to the NET manual [HM04]. Outside the NET research project, our emulation system as described in Section 2.2 has been successfully used for performance evaluation in a PhD thesis [Bau07] and a master's thesis [Kar04].

2.3 Summary

In this chapter, we identified the MAC layer as the optimal layer of the protocol stack whose network properties should be reproduced by an emulation tool. An emulation system consists of three modules: network model, emulation tool, and emulation control. We described each of those modules and discussed the possible alternatives of distributing those modules in an emulation system. The discussion showed, that an approach with hybrid model, distributed tools with one instance per network link, and centralized control is favorable. For this architecture, we outlined the functionality of emulation control and both an emulation tool for non-shared communication medium and a tool for shared communication medium. The concrete design for our prototypical implementation of an emulation system called Network Emulation Testbed (NET) was presented. With the concepts discussed in this chapter, each node in an emulation scenario is mapped to a PC of the testbed, which limits the scenario size to 64 nodes.

Chapter 3

Node Virtualization

In this chapter, we present node virtualization to enable scalable network emulation. After an introduction, we discuss related work in the area of scalable network emulation. A set of requirements serves our evaluations in later sections. As a basis for understanding, we give an overview of the foundations of protocol stack implementations before discussing possible approaches to node virtualization. Our discussion includes a qualitative comparison of the presented approaches. This leads us to our chosen approach for which we introduce the architecture of our virtualized emulation system. Finally, we present an extensive evaluation of our prototype validating that the approach we chose actually performs best. Parts of this chapter have been previously published in a conference paper [MHR05] and a journal article [MHR07].

3.1 Introduction

Often comparative performance evaluation is required in scenarios with more nodes than computers available in an emulation testbed. For instance, the evaluation of a large scale location service in a wired network requires not only the modeling of all collaborating location servers but also the modeling of the underlying network infrastructure including its routers to obtain realistic measurement results. Another example for the evaluation of a large scale system would be a peer to peer network. Even though the peer nodes seem to communicate directly with each other by means of an overlay network, the performance is strongly influenced by the underlying network infrastructure. Hence, this infrastructure has to be modeled in an emulation scenario along with the peer nodes (Fig. 3.1).

In wireless networks such as MANETs, each mobile node is itself both an end node and a router. Due to their mobility, each node dynamically influences the network topology and thus the performance of software under test such as

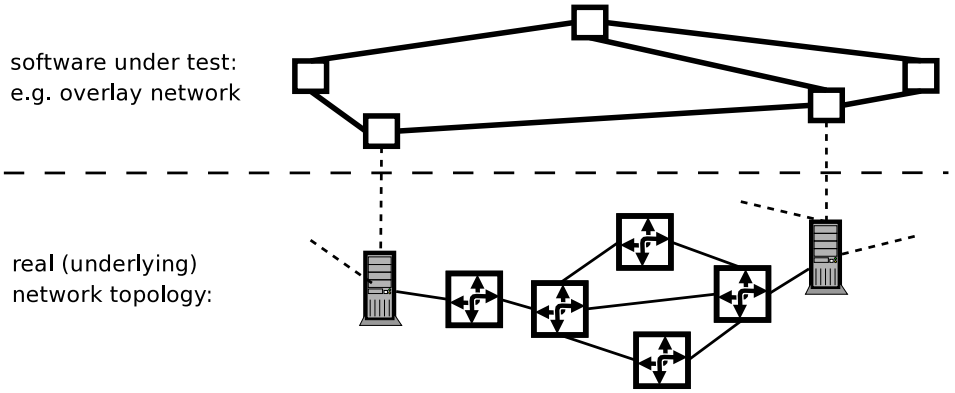

Figure 3.1: Part of an emulation scenario involving an overlay network.

a MANET routing protocol. As in the wired network cases, each node has to be modeled in an emulation scenario to obtain realistic measurement results. Consequently, there is need for scaling the number of nodes that can be emulated on a given testbed beyond the number of available testbed computers.

There are two complementary general approaches to scalability of network emulation. First, the existing resources of a testbed can be virtually increased by executing the software under test slower than real-time and thus trading experiment duration for scalability. However, we do not consider recent advances in network emulation using time virtualization to virtually increase network and compute resources of an emulation testbed. Such approaches are part of future work as discussed in Section 5.2. In this work, we focus on network emulation in real-time. The second general approach to scalability is to make optimal use of the existing testbed resources.

We consider three different approaches to making optimal use of existing testbed resources. First, emulators with centralized tools—such as simulators extended with real-time scheduling and interface for processing real network traffic—can be parallelized to increase scalability. Since our emulation approach is already distributed (except for the uncritical emulation control), parallelization is already implicitly included. Secondly, abstractions to the emulation model cause the emulation of network properties to require less computational power. This leaves more resources for processing network traffic. While such an approach increases scalability in terms of network traffic, it does not increase scalability in terms of node count. Hence, we consider abstractions to the emulation model a complement to scaling scenario size in terms of nodes.

Thirdly, software under test often only needs a fraction of the resources provided by a testbed computer. In order to make optimal use of existing resources, more than one node of an emulation scenario should be executed on a testbed computer.

This can be achieved by partitioning each testbed computer ("physical node," *pnode*) into multiple virtual nodes (*vnodes* [GSHL03]). Each of those vnodes provides an execution environment for one instance of software under test. The nodes of an emulation scenario correspond to vnodes and are mapped to the available pnodes. Such node virtualization should imply minimal virtualization overhead in order to leave as much of the pnodes' resources available for the execution of as many instances of software under test as possible. The node virtualization approach enables scalability of real-time network emulation in terms of the maximum number of nodes for a given testbed hardware.

3.2 Related Work

In this section, we review existing approaches to the scalable emulation of computer networks. All approaches have in common that they emulate certain network properties and support the transparent execution of unmodified software under test on certain layers of the protocol stack. The software under test perceives the properties of the emulated synthetic network as if it was running in a corresponding real network.

Approaches to scalable network emulation make use of different methods to increase the maximum possible number of nodes in an emulation scenario beyond the number of physically available testbed computers. We classify scalable network emulation approaches into three classes: parallelization, emulation model abstraction, and node virtualization.

3.2.1 Parallelization

The first class of scalable network emulation approaches builds on simulation—usually discrete event—and employs parallelization. Potentially large network topologies of an emulation scenario get partitioned and each partition is emulated on its own computer or CPU respectively. As long as the synchronization overhead between partitions is not too high, parallelization gains speedup with an increasing number of computers or CPUs. In addition, main memory requirements can be reduced on each computer or CPU, since only a fraction of the network topology has to be emulated.

Ns-e [Fal99] is an emulation extension of the well-known network simulator ns-2 [BEF+00]. The extension consists of a real-time event scheduler and an interface for processing frames of real network traffic. An emulation setup consists of one centralized simulator instance and a number of computers executing the software under test. The latter computers are connected to the central instance by a fast low latency network. On those computers, a proxy layer in the protocol stack intercepts

network traffic and passes it to the central instance and vice versa. Since the entire traffic of the emulated network is passed to the centralized simulation instance, the scalability is limited by the amount of traffic, that can be processed by the centralized simulator. For a typical mobile ad hoc network (MANET) experiment, a total scenario size of about 50 nodes is possible [KMJ00]. To some extent, this can be alleviated by extending the discrete event simulation into a parallel engine [RFA99]. However, an additional testbed computer is required for each instance of unmodified software under test to be executed.

IP-TNE [SU01] (Internet protocol – traffic and network emulator) is a parallel discrete event simulator (PDES) for computer networks consisting of point-to-point links. The simulator is extended with an efficient real-time scheduling of events and an interface for processing real network traffic. Those two extensions make IP-TNE a parallel network emulator. For the execution of software under test, additional computers are required. Traffic is passed through the parallel emulator instance. As with the above-mentioned ns-e, the possible number of nodes executing unmodified software under test is limited by the number of available computers.

Liu et al. [LC04] propose a real-time parallel discrete event simulator for networks consisting of point-to-point links. An interface for processing real network traffic supports emulation. The authors also describe a feedback based mechanism for mapping the network topology of an emulation scenario to the computers running the parallel emulator. Such an automatic mapping is essential for parallel discrete event simulation since its speedup through parallelization depends on the quality of partitioning the network topology. The less expensive events between the partitions emulated on different computers, the better. The authors report realistic emulation of large Internet-like topologies with 100 autonomous systems employing the routing protocols BGP4 (border gateway protocol 4) and OSPF (open shortest path first). A cluster of computers executing the PDES emulates such a network. The execution of unmodified software under test requires additional computers. Only applications are supported as software under test. A wrapper library for the socket API redirects messages to the emulation instance and vice versa.

3.2.2 Emulation Model Abstraction

The second class of scalable network emulation approaches abstracts from detailed emulation models. Using less detailed models requires less computation power for emulation. This allows more network traffic to be processed on the same hardware and thus increases scalability.

MAYA [ZJTB03, ZJTB04] integrates three different paradigms for modeling networks: simulation on packet-level, emulation, and simulation of fluid flows.

Simulation on packet-level means that the simulator needs to process an event for each packet to be sent on an outgoing link. This is similar to all the approaches discussed in Section 3.2.1 with the difference that MAYA is a centralized simulator. The emulation part is implemented similar to the extensions of other simulators with real-time event scheduling and an interface to process real network traffic. Fluid flows simulate long running flows on transport layer, e.g. TCP flows. Instead of simulating each packet on each hop of the flow's path, a simulation model based on queuing theory is used. Solving the resulting differential equations allows to determine average queue lengths of routers on the path and average packet delay of each flow from source to sink through the whole network. Based on this, the simulator is able to calculate the effective throughput of each flow. MAYA integrates the simulation of fluid flows and the emulation on packet-level by measuring statistics of the real network traffic and taking them into account for solving the differential equations of the fluid flows. Queue lengths of routers are now influenced both by fluid flows and real traffic. Hence, packet events of real traffic can be processed and account for the cross traffic of fluid flows and vice versa. The integration of these different paradigms allows parts of an emulation scenario consisting of links and routers to be efficiently simulated and thus increases scalability. However, an additional computer is required for each instance of unmodified software under test to be executed. The evolving real network traffic is then processed by the simulator, which may constitute a bottleneck.

Kiddle [Kid04, KSU05] presents three improvements to IP-TNE: novel parallel discrete event simulation algorithms to improve performance on multi-processor systems with shared memory, selective usage of shared memory communication between processors and message passing between computers for parallel simulation in mixed shared/distributed environments, and finally integration of packet-based and flow-based simulation. While the first two improvements are specific for parallel simulation, the latter extension is similar to the approach of MAYA with the difference, that both packet-based and flow-based simulation is tightly integrated inside IP-TNE. Since the flow extension does not support TCP fluid flows, traffic flows based on a Pareto on/off model are used. Unfortunately, the evaluations for large scale emulation including fluid flows for background traffic do not employ real unmodified software under test and its real network traffic. Instead, the simulator components, that would interface to real traffic, synthetically generate ingress IP packets or simply drop egress packets, respectively. Therefore, it remains unclear how scalable the system is with regard to the inevitable overhead of interfacing with real traffic.

ROSENET [GF04, Gu08] proposes an architecture, that comprises a low fidelity emulator and a high fidelity simulator. As example entities, the authors use an intermediate router node with NIST Net [CS03] as emulator and GTNetS [Ril03]

as parallel discrete event simulator. The emulator connects two computers each running an instance of unmodified real software under test on application layer. It emulates delay and packet drop based on a parametrized model and at the same time collects packet interarrival times of application traffic. The collected data is sent to the simulator, which performs packet-based simulation for the past application packets within a simulated potentially large network environment including cross and background traffic. Based on the simulated delay and loss of application packets, it determines a model from a library plus parameters for the model. A model is chosen with the system identification method. As example model, the authors use auto-regression with external signal as black box. If the model or its parameters have changed, they are sent to the emulator, which uses the parametrized model to predict the near future. This closed loop is executed periodically. The application of models in the emulator relies on the constancy of Internet path properties. The focus is on transport layer connections in wired Internet-like networks. The low fidelity emulation of delay and loss results in network properties, that are statistically comparable to those of the high fidelity packet-based simulation.

3.2.3 Node Virtualization

The third and last class of scalable network emulation approaches utilizes the fact that software under test usually deployed on resource-poor devices only needs a fraction of the resources provided by a testbed computer. By partitioning a testbed computer into virtual nodes, each testbed computer can emulate multiple communicating nodes of an emulation scenario. This optimizes the utilization of testbed hardware and increases scalability.

Entrapid [HSK99] and Alpine [ESW01] virtualize the protocol stack in user space and thus provide multiple execution environments for software under test on a single computer. In combination with network emulation tools connecting such virtualized stacks, the emulation of network scenarios is possible. However, software under test on the application layer has to be adapted in order to interact with the user space protocol stacks. The packet processing in user space also introduces considerable timing inaccuracies, compared to real protocol stacks. Thus, such approaches are rather suitable for testing than for performance evaluation.

vBET [JX03] is an approach designed for emulating a network scenario on a single computer. It makes use of User Mode Linux (UML) [Dik00] in order to provide virtual machines as execution environment for multiple virtual nodes on one computer. In combination with additional network emulation tools, it is possible to emulate a network scenario. Connecting multiple of such vBET computers could allow larger scenarios. However, the use of UML's virtual machine concept introduces considerable overhead and thus limits the number of virtual nodes per

computer. The authors report a maximum throughput for their software switch between vnodes of 128 MBit/s, which is more than an order of magnitude below our approach. vBET is more suitable for qualitative analysis than comparative performance analysis. Our results in Section 3.7.1 confirm that, since UML hardly supports more than one vnode per pnode in our scenarios even with applied optimizations for hosted VMs similar to [KDC03].

ModelNet [YWV+02, VYW+02, YED+03] is a parallel network emulator. It is primarily designed to emulate a given network topology of point-to-point links. The topology is partitioned among a cluster of emulation computers. Each cluster node processes network packets through internal arbitrarily connected links and routing instances. Existing implementations of e.g. routing protocols cannot be analyzed but have to be specifically re-implemented for the cluster nodes. Computers running software under test on transport or application layer have to be externally connected to the central emulation cluster. Several instances of the software under test can run on each such edge node, which makes the approach scalable. However, the interface to the emulated network is based on the socket API, which restricts the software under test to the application layer.

Mahadevan et al. [MYV02] proposed an early extension of ModelNet for the emulation of wireless networks. It emulates fair bandwidth sharing of the wireless shared medium by consuming bandwidth on all receiver links within communication range for each frame sent. This is a first step towards the emulation of wireless networks which are based on a shared medium. Yet, the effects of a MAC protocol such as frame drop due to collision are not emulated. Just as with ModelNet, the software under test is limited to the application layer. Performance evaluations in wireless networks such as MANETs often need to consider specific routing protocols. However, this extension for ModelNet does not support the execution of unmodified software under test on e.g. the network layer. Instead, a routing protocol would have to be reimplemented for the core of the parallel emulator.

MobiNet [MRBV04, MRBV05, MRBV06] descends from the same research project as ModelNet but is an independent approach for the emulation of MANETs. The presented emulation of IEEE 802.11 is completely centralized and not parallelized. Similar to our approach for the emulation of networks with shared medium, MobiNet makes use of a virtual carrier in its single global network model and implements the 802.11 MAC protocol on top. As with ModelNet, protocols on network layer have to be reimplemented for the environment of the centralized emulator. As an example for a MANET routing protocol, DSR (dynamic source routing) was especially implemented for MobiNet. Additional computers are required for running unmodified software under test on application layer. Those computers are connected to the central emulator. The authors report successful emulation of 100 MANETs of which each MANET has only 2 nodes. In such a

scenario, there is hardly concurrence between nodes on the same wireless channel and the network load can hardly be as high as to enforce many collisions. Since this scenario effectively resembles 100 completely separate tiny MANETs without any interaction, it is difficult to deduce the scalability of the proposed system from the reported evaluation results.

Mahrenholz and Ivanov [MI04] improve the real-time scheduler of ns-e by reducing system call overhead and enabling more fine-granular sleep timers with busy waiting which again introduces some overhead. Additionally, they introduce the use of virtual machines to expose unmodified software under test to network properties emulated by ns-e. Both the virtual machines and ns-e run on a single computer. While the virtual machines are planned to be distributed to multiple computers [MI05], the centralized ns-e still constitutes a bottleneck since it has to process the network traffic of the entire emulation scenario. The authors report faithful emulation of a WLAN scenario with up to 6 nodes represented by virtual machines. Similar to MobiNet multiple separate WLANs with only two nodes each are emulated in the evaluations which makes deduction of scalability difficult.

Empower or Emwin respectively [ZN02a, ZN02b, ZN02c, ZN03a, ZN03b, ZN04] partly virtualizes the protocol stack to allow the execution of multiple virtual routing instances on each emulation computer. Those instances are used to build a network topology consisting of wired point-to-point links or wireless links. For wireless emulation, each virtual router processes a list of node mobility events according to its local clock and determines dynamic connectivity based on node positions. Global coordination for processing the list is done by means of time synchronization between testbed computers. However, filtering of network traffic to emulate the current wireless connectivity happens on the sending side which prevents the emulation of the hidden terminal situation common in WLANs. This would require filtering on the receiving side. An algorithm for automatically mapping router nodes of an emulation scenario to the testbed hardware is provided and helps to prevent overloading the testbed computers with too many virtual router instances. Background traffic can be generated directly inside the emulator. Other traffic generators or software under test on transport or application layer need to be executed on additional computers connected with the computers emulating the network topology. This is similar to ModelNet discussed above. Each end of an emulated network link is mapped to a physical link of an existing hardware network interface. For wireless routers, even two physical links are required: one link for ingress traffic and one link for egress traffic which would in reality both be handled by one single network adapter transmitting or receiving on the wireless channel in turn. The authors equip each testbed node with several network cards to increase scalability. However, the scalability, i.e. the number of vnodes per pnode, is limited by the number of physical network interfaces per pnode.

Kucheria [Kuc03] proposes a scalable emulation system for IP networks consisting of wired point-to-point links by introducing partial virtualization of the protocol stack. In order to require minimal modifications to an existing protocol stack, the system only virtualizes the data link layer by introducing virtual network devices. The network layer and layers above are not virtualized. To still obtain a separate virtual routing table per virtual node, the virtual network device contains its own routing table along with its own look-up mechanism. Fake entries in the routing table of the existing IP implementation force packets to enter a special virtual device and trigger another route look-up on the respective virtual table. Obviously, such an approach is not entirely transparent to the layers above the data link layer. This results in some constraints for the configuration of the system. It only supports links between different testbed computers, i.e. virtual nodes on the same computer must not have a link between each other. This makes the mapping of an emulation scenario to the existing testbed hardware more difficult. In cases, where the longest linear path of the topology contains more routers than testbed computers, it makes the mapping impossible. Hence, the approach limits scalability for certain emulation scenarios.

Virtual IP machines [Ara03] virtualize the protocol stack to obtain virtual nodes for scalable network emulation. On network layer, each vnode has its own routing table and ARP (address resolution protocol) cache. On transport layer, each vnode has its own lists of sockets. On application layer, a tool exists to execute unmodified software under test and its child processes on a certain vnode. The implementation only supports point-to-point links. Links between vnodes on the same testbed computer are established internally and make use of efficient communication without requiring external network connections. However, there is no support for the emulation of wireless networks. Unfortunately, the author does not report any evaluation results, so there is no statement on the scalability of the system.

Netbed [WLS+02] is an emulation testbed, that supports scalable network emulation by employing protocol stack virtualization [GSHL03, HRS+04, HRS+08]. On data link and network layer, Netbed uses virtual routing [SR02] providing multiple routing tables and a mapping between packets received by certain devices and the corresponding routing table. To extend the virtualization also to transport and application layer, additional extensions have been made and BSD jails [KW00] have been integrated to extend the virtualization also to other non-network namespaces of the operating system such as file systems and processes. The testbed supports the emulation of scenarios with wired links. While it is possible to link real wireless nodes to an emulated scenario [WLG02, JSF+06], there is no support for the reproducible emulation of wireless networks.

Vimage or Imunes respectively [Zec03, ZM03, ZM04] virtualizes the entire protocol stack in the kernel. While common operating systems support one single

instance of a protocol stack, vimage supports multiple independent instances. To accomplish this, the stack is considerably modified to have all formerly global instance variables independently available for each stack instance. Unmodified software under test on application layer can be executed within a certain protocol stack instance. In combination with the network emulation tool dummynet, link-based scenarios can be emulated in a scalable way. In [ZM04], the authors report TCP throughput of 420 MBytes/s over 15 routing hops on a single machine with a slightly faster processor than used in our evaluation. Though scaling significantly better than a VMware based virtualization implementation, the throughput was measured in a best case without any introduction of emulated network properties such as bandwidth limitation and is thus hardly comparable to our results. Emulated network properties are however essential for network emulation and imply emulation overhead due to timer management reducing the accumulated throughput that can be realistically emulated.

Engel et al. [ESHF04] focus on the emulation of MANETs. On a single computer, a centralized emulator component connects virtual machines with each other reproducing wireless network connectivity. The authors report the emulation of eight to ten virtual machines provided by L4Linux guests running on an L4 microkernel hypervisor. In contrast, virtual machines with User Mode Linux employ higher overhead and can only provide three to four virtual machines on the same hardware. While it is possible to have one virtual machine configured as router to connect to a real network outside of the emulation, a distribution of multiple emulation computers is not supported. This limits the scalability of the approach.

V-eM [AC06, AH06] makes use of virtual machines to provide execution environments for unmodified software under test on network, transport, and application layer. Using the Xen virtual machine monitor [BDF+03, FHN+04], V-eM provides strict resource isolation between virtual nodes at the cost of increased virtualization overhead compared to protocol stack virtualization. To alleviate some overhead of network I/O, the emulation tool NISTnet [CS03] is integrated into the virtual network device backend driver of Xen's management virtual machine (dom0/xen0), which provides access to the physical network devices of a testbed computer. This allows to drop frames early without processing them further in vain. In order to make all communication between any vnodes uniform, V-eM does not allow efficient internal communication between vnodes on the same pnode. Instead, traffic from the source vnode is sent through a physical network interface and received by another physical network interface for delivering to the destination vnode. In wireless scenarios, nodes communicate over a shared medium. Since the dynamic connection topology is not known in advance, one has to assume the worst case of all nodes being within communication range of each other, i.e. one

link between each pair of nodes. When mapping such a scenario to the testbed, two physical network interfaces per wireless node are required. Similar to Empower, the scalability is limited by the number of physical network interfaces per pnode. Additionally, due to the strict separation of virtual machines' address spaces the approach has increased memory requirements when compared to protocol stack virtualization. Hence, main memory may become a bottleneck. The authors report successful emulation of a scenario with 10 routers connected by 100 MBit/s links. Unfortunately these results are not comparable to ours, since they used two processors each of which is faster than our hardware.

3.2.4 Conclusion of Related Work

Parallel network emulation is able to provide scalability as long as the network topology can be partitioned such that network traffic between partitions and thus synchronization overhead is minimized. Even though the parallel emulator instance is executed on multiple CPUs or core computers, the instance has to process all network traffic. Traffic from the attached edge computers executing software under test may be balanced among the emulator core computers. However, in parallel emulation setups with less core computers than edge computers, one core computer has to process traffic from multiple edge computers which might result in an unwanted bottleneck. Our distributed approach to network emulation does not require synchronization and also does not contain a potential bottleneck. Hence, it does not have the limitation of parallel emulation and scales with increased hardware resources.

Emulation model abstractions, such as fluid flow models, usually emulate aggregate network properties of lower layers and operate themselves on layers above the MAC layer, which we identified as optimal emulation layer in Section 2.1.1.6. Therefore, emulation model abstraction cannot generally be used to increase scalability. It is useful to efficiently emulate a network consisting of links and routers through which traffic of software under test should be sent and interact with synthetic cross traffic or to generate and inject background network load efficiently. Hence, we consider it a complement to node virtualization.

Scalable emulation approaches employing node virtualization make optimal use of the existing testbed hardware without introducing bottlenecks and without giving up flexibility. Therefore, we consider node virtualization the most suitable approach for scalable network emulation. Existing approaches all have their own limitations. Some are more suitable for developing or testing than for performance measurement. Others are limited in scalability by either hardware or software. We would like to minimize modifications to potential software under test and be able to obtain reproducible measurement results even for emulated wireless networks. Consequently, there is need for a scalable emulation approach based on

lightweight node virtualization with low overhead which fulfills all requirements of comparative performance evaluation.

3.3 Requirements

In general, node virtualization provides a way to schedule access of a number of consumers to otherwise exclusively used hardware resources. With respect to network emulation, each consumer is an instance of software under test inside its execution environment—the vnode. In this section, we define our three major requirements for node virtualization in the context of network emulation.

Node virtualization should be *transparent* to the software under test, to allow the execution of unmodified code of an implementation. Heterogeneous scenarios involve different implementations of the software under test which require different execution environments, such as a TCP implementation in Linux and another one in Windows. Since more execution environments than pnodes may be necessary, node virtualization should be *flexible* to support different execution environments on the same pnode. The main reason for the introduction of node virtualization to network emulation is scalability with respect to the maximum size of emulation scenarios. The overhead of node virtualization strongly impacts the possible degree of virtualization per pnode and hence the size of an emulation scenario. Thus, node virtualization should be *efficient* to maximize the scenario size in terms of vnodes for a given number of pnodes.

The criteria transparency and flexibility can only be discussed on a qualitative basis whereas efficiency can be evaluated quantitatively by means of experiments. Therefore, we split the evaluation discussion over two sections. We treat the qualitative criteria first in Section 3.5. The results are used to choose an optimally suited architecture presented in Section 3.6. For the prototype implementing this architecture, we discuss the quantitative evaluation in Section 3.7.1.

3.4 Foundations of Protocol Stacks

We are interested in lightweight node virtualization for the emulation of computer networks. In order to show the entities of a networked computer system, that need to be virtualized, we give a brief overview of the implementation of a typical protocol stack before discussing possible approaches to node virtualization in Section 3.5. Our presentation is based on standard books discussing the design and implementation of TCP/IP protocol stacks on Unix variants [WS95, CS99, WPR+02] and the socket API [SFR04], the source code of the Linux kernel in version 2.4.24, and our own lecture on concepts of computer network programming [HML07]. The

concepts only rely on a layered protocol architecture according to the hybrid 5 layer model already used throughout Section 2.1.1 and can easily be transferred to any other protocols.

Since node virtualization basically has to provide execution environments for instances of software under test, it is sufficient to focus on those protocol layers, which we identified as possible software under test in Section 2.1.1.6: logical link control, network, transport, and application layer. Entities in each of those layers store state that represents exactly one node. Hence, the state variables of the relevant entities need to be replicated exclusively for each virtual node instance to obtain node virtualization. In the following, we discuss the layers shown in Tab. 3.1 from bottom to top.

layers	**entities**
L5: application layer	*processes* having network file descriptors (inherited by child processes)
L4: transport layer	*communication endpoints* (each one including state of connection / reliability / congestion control / flow control)
L3: network layer	*routing table*, neighbor cache, fragment lists for reassembly, state of connections / reliability / congestion control
L2: data link layer	
(logical link control)	state of connection / reliability / flow control
device drivers	*network devices*

Table 3.1: Entities of a protocol stack that need to be virtualized.

Software under test requires one or more connections to the emulated network. Operating systems provide connections by means of network devices. Hence, each vnode should have its own *set of network devices*. In Linux, network devices are represented by a given data structure, which can be extended for specific device subtypes. For each device, an instance of this structure and possibly its extension is kept in main memory. The network device structure stores among other things: the configured network layer address(es) and possibly the device's MAC layer unicast address, a list of joined network layer multicast groups and possibly a list of joined MAC layer multicast addresses, and the transmission queue holding frames to be transmitted.

Depending on the existence of the logical link control layer and its implemented services, different state should be kept separately for each vnode instance. For a connection oriented service, the connection state is relevant. For a reliable service, the state of outstanding frames is of importance. If LLC implements flow control,

e.g. using the concept of a sliding window, the state consisting of the current window size and the window pointers matters. For our prototype based on the Linux TCP/IP protocol stack, there is no LLC layer.

Probably the most important functionality of the network layer entity is the forwarding of packets. This requires a routing table to determine which network device to forward a packet to. Especially in MANETs, where nodes of the same wireless channel belong to the same subnet, multiple nodes may have routing table entries pointing to exactly the same destination, namely some other node within reachability. Such entries would unavoidably collide, if multiple vnodes shared a single routing table. Hence, each vnode needs its own exclusive *routing table*. Implementations of routing protocols may be designed to run as processes outside the operating system kernel and update the routing table in the kernel by means of a defined programming interface. Each instance of such a routing protocol running on its own vnode should transparently get access to the corresponding routing table when using the defined programming interface. If the network layer protocol uses its own globally unique addresses and a shared medium network is used for communication, a neighbor table is required for translating network layer addresses into MAC addresses. We assume network layer addresses of nodes, that can communicate with each other—potentially over multiple hops—, to be unique within an emulation scenario. Under this assumption, neighbor table entries are always unique and a separate table per vnode is not necessary. Under the same assumption, separate sets of fragment lists for the reassembly of fragmented packets are not necessary. Depending on additionally implemented services of the network layer, the state of connections, of outstanding frames for reliability, or of congestion control should be kept separately for each vnode instance. For our prototype based on the TCP/IP protocol stack, the network layer entity IP implements a connectionless and unreliable service without congestion control. Hence, only routing tables and their update interface have to be virtualized in our case.

Among other things, the transport layer cares for the addressing of services on a computer. Software under test should be able to bind to the same service address, e.g. port number, in each vnode instance. Therefore, each vnode instance should keep its own exclusive *set of communication endpoints* on transport layer. For TCP/IP implementations, one end of a communication link is typically a socket and contains a protocol control block (PCB) storing the protocol state. With UDP or TCP, this includes the 4-tuple of IP address and port for the local and remote side respectively, as well as the state of the send buffer and receive buffer. With TCP, additional state comprises the connection, outstanding frames for reliability, congestion control and the sliding window for flow control. Since all state is already encapsulated in sockets, it is sufficient to have an own exclusive set of sockets for

each vnode instance.

Software under test on application layer uses the services of lower protocol stack layers by means of a defined programming interface. Processes comprising an instance of software under test should automatically access the respective communication endpoints of the vnode instance they are running in. In other words, the defined programming interface should make sure, that the state variables of the corresponding vnode instance are accessed. Since file descriptors including communication endpoints usually are inherited on spawning new processes, each new processes should also be associated with the vnode instance of its creator. Hence, each vnode instance should maintain its own exclusive *set of processes*. On Unix and thus also on our prototype based on Linux, a child process created with the fork system call inherits open file descriptors including sockets from its parent process. A new child process should be associated with the same vnode instance as his generating parent process such that new sockets created in the child belong to the same vnode instance. This enables the transparent use of unmodified software under test consisting of a process hierarchy.

3.5 Approaches and Qualitative Comparison

There are different approaches to virtualize the entities identified in Section 3.4 in order to obtain lightweight node virtualization for scalable network emulation. For our discussion, we assume that the protocol stack is part of the kernel space, as is prevalent in commodity operating systems. The presented approaches can be classified into two main categories: virtual machines and operating system partitioning.

3.5.1 Virtual Machines

A straightforward way to introduce node virtualization is using a virtual machine (VM) approach. There are different classes of virtual machines. We use the taxonomy of Smith and Nair [SN05] as shown in Fig. 3.2. The taxonomy distinguishes two major classes of VMs based on the specific programming interface of the computer system architecture that they virtualize: process VMs and system VMs. Each of those two classes is further divided into two subclasses. The subclasses differ in the instruction set architecture (ISA) provided by the virtualization software to VM instances. It can either be the same ISA as the one of the underlying hardware or a different ISA.

Process VMs virtualize the application binary interface (ABI) which is used by processes on the application layer (see Fig. 3.3). The ABI consists of two parts: system call interface and user ISA. Processes use the system call interface to

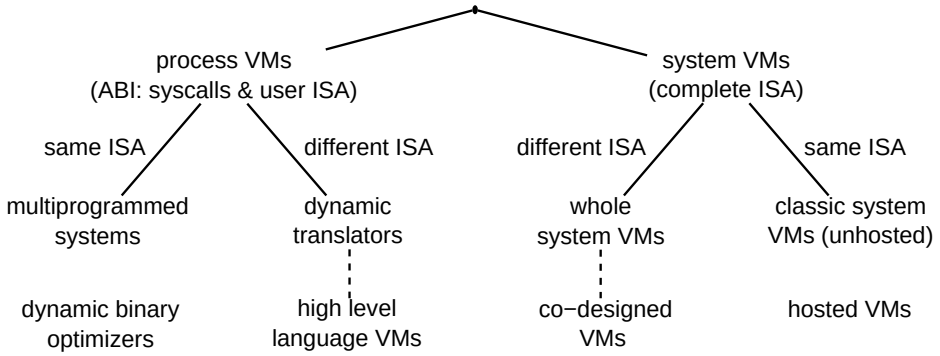

Figure 3.2: Taxonomy of virtual machines according to Smith and Nair [SN05]

invoke services of the operating system. The binary code of processes adheres to the user ISA and is executed by the processor hardware.

Apparently, part of the software under test—namely the network and transport layer as part of the host operating system (host OS)—does not get virtualized by a process VM. With extra porting effort it would be possible to let the protocol stack run as part of the virtualizing software and thus virtualize all necessary entities belonging to the software under test. However, as discussed earlier in the related work (Section 3.2.3), such an approach like Entrapid [HSK99] is more suitable for testing than for performance evaluation due to its implied performance overhead. Therefore, we do not consider process VMs a viable approach to node virtualization. Assuming that an instance of software under test may include multiple processes on application layer, multiprogrammed systems are in fact our point of origin.

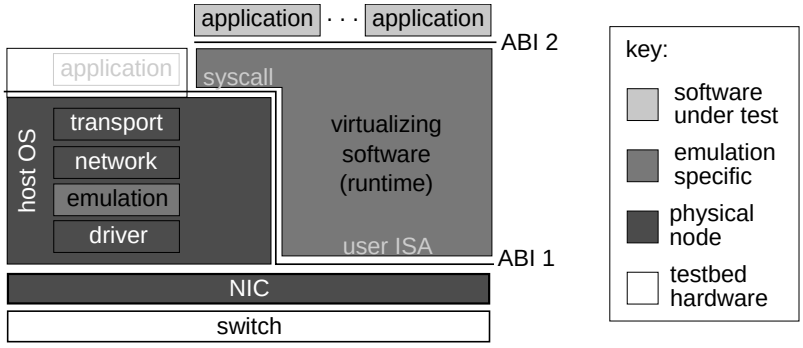

Figure 3.3: Process virtual machine adapted from [SN05].

3.5.1.1 System Virtual Machine

System VMs virtualize the entire instruction set architecture (ISA) consisting of system ISA and user ISA as they are used by both OS and applications (see Fig. 3.4). Instead of running an OS (and applications) directly on the bare hardware, a shim of software is inserted in between. This virtualization software—the virtual machine monitor (VMM) or hypervisor—schedules access of multiple guest operating systems to exclusive hardware resources managed by the VMM.

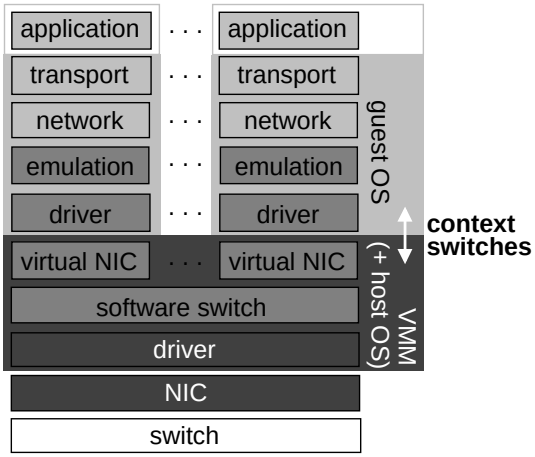

Figure 3.4: System virtual machine.

System VMs, that provide the guest with a different ISA than the one of the hardware, emulate a certain hardware platform. One example is bochs [Law96]. Depending on how detailed the system emulation is, any guest, that runs on the corresponding real system hardware, would run inside the virtual machine. This makes a system VM with different ISA *transparent* and *flexible* with respect to software under test. However, since the system emulation has to be performed all in software, such an approach implies huge overhead. This is *inefficient* and severely limits scalability. Therefore, we do not consider such an approach suitable for scalable network emulation. Also, we assume that potential software under test runs on common off the shelf hardware such as PC-compatible notebooks as an example for mobile devices. This means that testbed computers, which are also common off the shelf PC hardware, have the same ISA and can directly execute the software under test. Hence, we do not have a requirement to support ISAs that are different from the underlying hardware.

System VMs with the same ISA for the guests as the hardware can alleviate the virtualization overhead since the hardware directly executes machine code of the guests. The VMM only traps privileged instructions and access to privileged address spaces.

The granularity of scheduling in system VMs is by guest OS instance. Whenever a guest would like to work something, while another guest is currently scheduled, it has to wait until its own turn. For instance, on receiving a frame from the network, the guest has to wait until it gets scheduled by the VMM in order to actually process the frame. Apparently, this serialization of CPU time introduces latencies for the guests. The more guests are provided by a VMM, the longer such latencies can get. The worst case happens when each guest makes use of its full time-slice and does not yield the CPU voluntarily. Such latencies may influence emulation experiments and lead to unrealistic measurement results.

In order to support emulated network properties, an emulation tool needs to be integrated into such a virtualized system. There are different solutions for an integration. For our discussion here and in Section 3.7.1, we choose to insert our unmodified existing emulation tool on top of network interface drivers inside each guest OS. This allows for focusing on the comparison of different virtualization approaches by keeping the surrounding environment fixed. For communication with other vnodes on the same and on other pnodes, a software switch in the VMM forwards frames correspondingly.

System VMs with same ISA for guest and hardware can be further distinguished into hosted VMs and classic system VMs.

Hosted VM The VMM of hosted VMs makes use of the services of a host OS instead of accessing hardware such as devices itself directly. Hosted VMMs are also called Type II VMMs [Gol73]. Examples for hosted VMs are Plex86 [Law00], VMWare Workstation [SVL01], or Linux Kernel-based Virtual Machine (KVM) [KKL+07]. They support unmodified operating systems, and thus protocol stacks, in each guest instance. Therefore, they are *transparent* with respect to software under test. Since each guest may run its own different OS, hosted VMs are *flexible*. However, context switches between guest OS, and VMM or host OS are triggered by privileged commands or page faults. Network communication also causes such context switches, which imply virtualization overhead due to the necessary emulation of virtual devices to be used by a guest OS. This property of hosted VMs *restricts efficiency* and thus limits scalability.

There are also hosted VMs that do not try to virtualize the ISA of a certain kind of hardware but, e.g. the system call interface of the host OS. User-Mode Linux (UML) [Dik00] or UMLinux / FAUmachine [BS01, SB02] are for instance ports of the Linux operating system to its own system call interface. They combine properties of both a process VM and a system VM. The lower interface of the virtualization software is the ABI of the host OS including its system call interface and user ISA just as in a process VM. The upper interface to the guest is an artificial ISA comparable to a system VM. The hardware abstraction layer (HAL)

of the modified guest OS represents the virtualization software and translates between upper and lower interface. The porting process usually involves the modification of only certain low level hardware access functions often organized in a HAL module. The module seldom overlaps with the software under test making such a virtualization approach *transparent*. In contrast to the above-mentioned typical hosted VM, there is no single VMM that is able to support arbitrary guest operating systems. In fact, each different guest OS has to be ported to the host OS. Therefore, such an approach *restricts flexibility*. The way privileged commands—which are system calls to the host OS in this case—are trapped implies significant overhead. Since network emulation with frequent communication causes a lot of such system calls, such an approach is subject to substantial virtualization overhead. Simple modifications to the host OS improve performance at least comparable to that of typical hosted VMs as mentioned above [KDC03]. Therefore, we consider this special case of hosted VMs also having *restricted efficiency* and thus limiting scalability.

Classic System VM (Unhosted) In comparison to hosted VMs, the VMM of a classic system VM can alleviate virtualization overhead by accessing the hardware directly and thus reducing context switches. Unhosted VMMs are also called Type I VMMs [Gol73]. IBM CP [MS70] or VMWare ESX [Wal02] are examples for classic system VMs. Such a VM supports unmodified guest operating systems and is thus transparent with respect to software under test.

In addition to the unhosted property of classic system VMs, another system VM concept called para-virtualization can be added to reduce virtualization overhead further. Para-virtualization introduces a system ISA and/or an interface to I/O devices that are based on the original hardware interfaces but contain slight differences to achieve virtualization with low overhead.

The new system ISA typically tries to eliminate unprivileged sensitive instructions, which otherwise prevent a processor to be (hardware) virtualizable. Such instructions may influence state of the VMM but do not trap into the VMM to emulate the necessary behavior for its guests. A processor is only (hardware) virtualizable if the sensitive instructions are a subset of the privileged instructions [PG74]. This was, e.g., not the case for the Intel IA-32 (x86) processors [RI00] until the introduction of hardware support for virtualization [Str05, NSL+06]. Therefore, dynamic binary translation was previously used to replace unprivileged sensitive instructions by traps to the VMM [Law00]. With para-virtualization such instructions are statically replaced in the source of a guest OS. Apparently, this requires modifications in the HAL of the guest operating systems.

The new interface to I/O devices typically tries to eliminate expensive interrupt handling and numerous writes to device I/O ports which cause unnecessary context

switches and thus overhead. The basic mechanism is a ring data structure to exchange blocks of data between VMM and guest OS [Rus08]. The guest operating systems need at least custom device drivers supporting this artificial interface.

Para-virtualization is a general VM concept and by no means tied to unhosted VMs. E.g. VMWare Workstation using bundled network device drivers for the guest OS [SVL01] is a hosted VM partially para-virtualized with respect to network resources. Yet, we only discuss para-virtualization along with unhosted VMs since the combination of VMs having low virtualization overhead with a concept reducing overhead even further promises to imply lowest overhead for VM approaches.

Denali [WSG02] and Xen [BDF+03, FHN+04] are examples for unhosted VMs with para-virtualization. Xen offers two different ways of virtualizing the hardware resources. On modern processors with hardware virtualization support, Xen supports the execution of unmodified guest operating systems. However, the necessary faithful emulation of hardware devices implies significant virtualization overhead. Therefore, we focus on the other, original method of Xen incorporating para-virtualization.

Usually, modifications for porting the guest OS to a para-virtualized interface do not overlap with the software under test, making such an approach *transparent*. Each different guest OS has to be modified to support the para-virtualized interface. Therefore, such an approach *restricts flexibility*. Classic system VMs with para-virtualization imply the lowest virtualization overhead and are the most *efficient* virtualization approach so far.

3.5.2 Operating System Partitioning

The virtual machine approaches described in Section 3.5.1 actually virtualize more than is necessary for network emulation. It would be sufficient to provide virtual execution environments for just the software under test as identified in Section 2.1.1.6: logical link control, network, transport, and application layer. Assuming multiprogramming, multiple instances of software under test can already be executed on the application layer. The remaining parts of potential software under test resemble a subset of an OS. Partitioning this subset enables the execution of multiple instances of software under test on all required layers.

Existing approaches partition different namespaces of an OS. Partitioning of processes and file systems, such as with BSD jails [KW00], Solaris Zones [PT04], or Linux-VServer [SPF+07], can be used to provide root access to multiple users and yet separate them from each other and the base system. This has enabled the spread of so-called virtual root servers at Internet service providers (ISPs). Such virtual root servers are often used to, e.g. serve content in the Internet such as dynamic web sites. The virtualization enables consolidation of multiple customers on a single computer at an ISP. Advanced approaches include partitioning of the

protocol stack, e.g. Trusted Linux [DC01], OpenVZ (Virtuozzo) [ELK07], or Linux resource containers [BBHL08]. This further improves flexibility and security by separating root users of the same computer even more. Partitioning of the protocol stack in combination with multiprogramming virtualizes exactly our required layers of software under test.

3.5.2.1 Virtual Protocol Stack

In the following, we call the combination of partitioned protocol stack and multiprogramming simply a virtual protocol stack. With virtual protocol stacks (Fig. 3.5), a vnode consists of the following sets: a set of network devices on the data link layer, a routing table on network layer, a set of sockets on transport layer, and a set of processes on application layer. In contrast to virtual machines, there is no more need for separate virtual network devices and their drivers. The emulation tool itself can appear as several instances of a virtual network device. In tight cooperation, a software switch forwards frames appropriately in order to allow communication between vnodes on the same and on different pnodes.

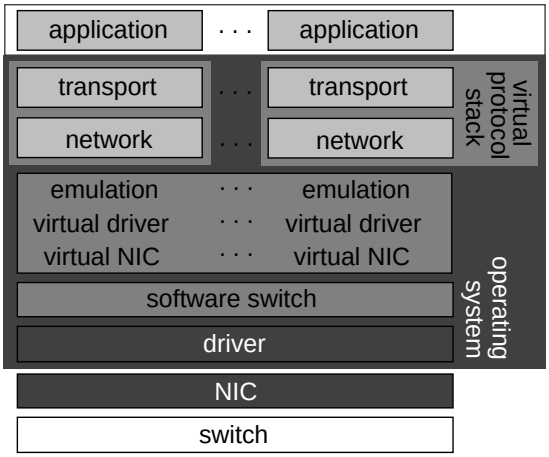

Figure 3.5: Virtual protocol stack approach.

The granularity of scheduling with virtual protocol stacks is by process and by interrupts in the OS. As long as a process of one vnode is scheduled, processes of other vnodes have to wait until their turn. In contrast to VMs, the reception of frames and processing in the protocol stack up to the socket receive buffers happens promptly. Therefore, the timely behavior of protocols in emulation scenarios is lesser affected by the number of vnodes on a computer than with VMs. Virtual protocol stacks promise scalable network emulation with realistic measurement results. Of course, latencies until received data can be handled in a currently

unscheduled process of a vnode are unavoidable. However, even without virtualization, applications might have to deal with latencies due to the decoupling of protocol stack and applications by socket buffers.

Examples for virtual protocol stacks are Trusted Linux [DC01], vimage/imunes [Zec03, ZM03, ZM04], OpenVZ (Virtuozzo) [ELK07], Linux resource containers [BBHL08], or different virtual routing implementations [SR02, KHS+03, Leu04]. While virtualization can be transparent to routing daemons implementing a routing protocol in user space, it requires modifications to other parts of the software under test, in order to provide separate state variables for each stack instance. We therefore consider virtual protocol stacks *partly transparent* to the software under test. Since state variables are replicated but not the code of protocol entities, virtual protocol stacks only support multiple instances of the same stack implementation on one pnode (an exception being vimage, which makes increased effort to slice the stack from the OS and thus supports different stack implementations inside one OS). Yet, using different implementations on different pnodes remedies this *partial flexibility*. Compared to virtual machine approaches, there are no redundant context switches and copy operations. The virtualization overhead for virtual protocol stacks is as low as possible, making them the *most efficient* virtualization approach.

Additionally, virtual protocol stacks support scalability in terms of low memory footprint per vnode. In contrast to VM approaches, which would need special support to share pages between guests [Wal02, Int04b], all vnodes on the same pnode share code segments. Since instances of software under test in vnodes often use the same binaries and especially the same libraries, this results in significantly reduced memory requirements allowing more vnodes for a given memory configuration.

It stands to reason to complement virtual protocol stacks with partitioning of additional OS namespaces such as processes or file systems. On the one hand, it helps establishing a view where vnodes look more closely like VMs from the viewpoint of applications and users. On the other hand, having all vnodes on a pnode in the same root file system might also be considered more convenient since there is no need to make common files such as executables and libraries accessible in each vnode. The only rule an experimenter has to obey is to provide different file names to the file descriptors of different instances of software under test. Processes in different vnodes must not write to the same file, in order to avoid data corruption.

3.5.3 Summary

Tab. 3.2 shows a summary of the discussion in Section 3.5.1 and 3.5.2. We rate each approach on a scale of three levels with plus denoting fulfillment, a circle

virtualization approach	transparency	flexibility	efficiency
hosted VM	+	+ / ∘	∘ / −
classic system VM	+	∘	∘
virtual protocol stack	∘	∘	+

Table 3.2: Comparison of candidate virtualization approaches.

denoting partial fulfillment, and minus denoting restrictions with respect to our requirements from Section 3.3.

While virtual machine approaches can be fully transparent and flexible, they do not fully comply with our paramount goal efficiency. Virtual protocol stacks provide low virtualization overhead and efficient intra-pnode communication. We thus consider virtual protocol stacks best suited for scalable network emulation.

3.6 Architecture

Based on our conclusion in the previous section, we devise an architecture employing virtual protocol stacks. Our architecture supports scalable emulation not only of networks consisting of point-to-point links but also of shared media based networks such as mobile ad hoc networks and even arbitrary combinations for the emulation of hybrid networks. Linux 2.4 serves as operating system for our implementation. As background reading material, we used standard books [BBD+01, RC01, WPR+02], a paper [SOK01], an RFC [SKKK03], various magazine articles [Cox96, kl99, Rub00], material from the Linux documentation project [Rus99, Pom99, Aiv01, Hen02], Linux source code documentation [Smo00, TW+01], as well as the content of our own lecture on design, implementation, and usage of protocol stacks [HML07].

In the following, we describe the three main blocks of our architecture traversing the protocol stack layers from bottom to top: software communication switch, virtual routing, using the two previous blocks for configuring virtual nodes, and finally hierarchical emulation control.

3.6.1 Software Communication Switch

In the context of our network emulation testbed, each software switch introduces a “stacked” sub-switch using the emulation network connection as an uplink to the emulation switch (cf. Fig. 2.4 in Section 2.2.1.1). A software switch resembles the functionality of a hardware Ethernet switch. It mediates both between vnodes located on the same pnode as well as between vnodes located on different pnodes. This provides transparent switching between any vnodes in a scenario.

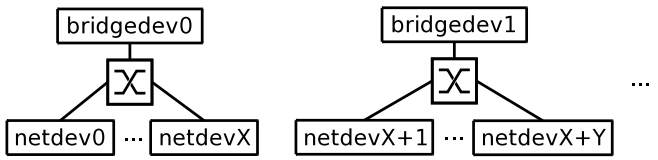

Figure 3.6: Linux bridges and attached network devices.

A software bridge for frame switching already exists in the Linux kernel. However, it primarily supports switching between a number of real Ethernet devices. Only one virtual bridge network device per bridge instance is available for local communication with the local network stack and applications and thus our software under test (Fig. 3.6).

In contrast to the software bridge already existing in Linux, we need one uplink to a real device and multiple local ends. Therefore, we design a custom Linux kernel module providing instances of a virtual switch network device "vnmux" (virtual node multiplexer). Our design supports transparent communication between any local and remote vnodes as well as unvirtualized pnodes. Additionally, link and shared media based configurations can be mixed arbitrarily even on the same pnode.

Fig. 3.7 shows a configuration example involving point-to-point links (vnode1 and vnode2 on pnode1), a hybrid node with both point-to-point link and connection to shared medium (vnode3 on pnode1), and shared medium such as a MANET (vnodes on pnode2). Inside pnode1, vnode1 has a connection to a different pnode via netsh0, vnmux0, and vlan2. Vnode1 and vnode2 are connected through netsh1 and netsh2 respectively with a virtual point-to-point link represented by the virtual switch instance vnmux1. Similarly, vnode2 and vnode3 are connected through the virtual point-to-point link vnmux2. Vnode3 of pnode1 as well as vnode1 through vnode3 of pnode2 participate together in one shared media network represented by vlan3 on the emulation switch as well as on the network virtualization layer by vnmux3 on pnode1 and by vnmux0 on pnode2. More details on virtual node configuration follow in Section 3.6.3.

To get an uplink, this device can be bound internally to the driver of a real network device (NIC). The latter can also be a tagged VLAN device which is in turn bound to a real network device. Doing so enables the creation of an efficient, hardware-assisted connection topology as discussed in Section 2.2.1.2. The maximum available number of non-reserved VLAN tags is limited to $4096 - 3$. However, this does not constitute a scalability limitation, since as much network connections as possible should be mapped to intra-pnode connections not involving a VLAN. So only the remaining inter-pnode connections require a unique VLAN tag. Promiscuous mode for the bound device enables the reception of frames

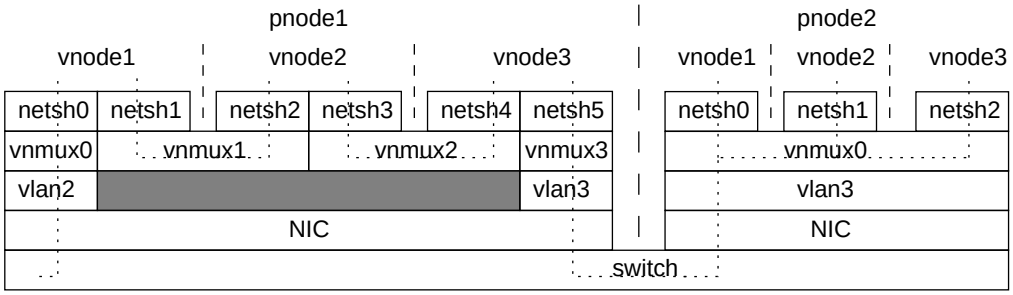

Figure 3.7: Pnode configuration examples: link based (left), hybrid (center), shared media based, e.g. MANET, (right).

destined for local vnode devices. The emulation hardware switch takes care of filtering. It delivers only locally targeted frames after its learning phase, so that the software switch only has to process frames it really is responsible for.

Our software switch processes the relevant frames without copying of payload data. This is essential to fulfill our requirement for efficiency. Non-local outgoing frames are handed over by reference to the uplink device for transmission. For locally destined frames, all references to any existing kernel objects except for payload data are released from the frame's administrative data structure sk_buff. The administrative data structure of frames with a broadcast destination address has to be cloned on delivery for each local recipient. This is necessary since the receive path assumes exclusive and unshared administrative frame data structures. Clones share the same payload without copying it. Subsequently, protocol type, packet class (unicast/broadcast), and local receiving destination device are set for the frame. Finally, the frame is reinserted immediately on the receive path.

At the upper interface of the switch, virtual network devices provided by emulation tool instances ("netsh*X*") register themselves to generate local switch "ports." For those local ports, we potentially need an arbitrary number of MAC addresses which do not conflict with officially registered Ethernet hardware addresses. To create them, we use locally administered unicast addresses having '01' as common bit prefix. Our software switch uses a fixed size table, that contains pointers to registered emulation tool instances representing local ports. The table index is

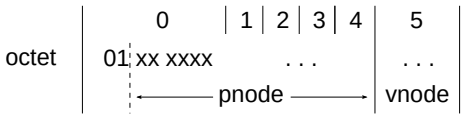

Figure 3.8: MAC address format for software switch ports (in Ethernet wire bit order).

directly determined by the destination MAC address. This results in a low constant look-up time for unicast switching decisions with a complexity of $O(1)$. To this end, we require a special pattern for MAC addresses of vnode devices (Fig. 3.8). Addresses are split into two partitions similar to subnetting of network addresses. The more significant bits identify a pnode. The less significant bits identify vnodes on that particular pnode and serve as vector index for look-up. For the time being, we choose a mask of 5 to 1 bytes respectively, allowing 2^{38}, i.e. over 274 billion, pnodes and 256 vnodes per pnode.

3.6.2 Virtual Protocol Stack

On top of our emulation tool's virtual network devices, that are connected to the software switch, multiple virtual routing instances represent possible software under test. There are two methods for implementing virtual routing in the Linux protocol stack. First, Linux supports having more than one routing table to enable policy based routing. A rule list of policies determines which routing table to consult for a forwarding decision. We refrain from exploiting this functionality to support virtual routing, as this would modify the existing semantics and thus it would violate our goal for transparency. The second method for implementing virtual routing is duplicating the rule list of policy based routing for each virtual protocol stack instance. Since this method retains existing semantics and functionality, we choose this second method. For simplicity, we assume in the following discussion that an unmodified Linux provides exactly one routing table, while in fact there would be one policy rule list with multiple tables attached.

We base our implementation on kernel patches for "Virtual Routing and Forwarding" (VRF) version 0.100 by James R. Leu [Leu04]. VRF provides multiple instances of forwarding information bases (routing tables) as well as mechanisms to associate network devices, and IPv4 UDP/TCP sockets with instances. User space tools exist for instance maintenance and for associating network devices with instances. Missing in the main distribution is the possibility to associate processes with instances. Additional patches by Yon Uriarte provide this feature in part by adding a VRF instance identifier to the process control block. A user space tool can set this ID and thus associate a process with a certain VRF instance.

Towards full transparency for software under test, we extend VRF in four areas: inheritance of VRF ID between processes, access to routing tables from processes, queuing packets to user space processes, and using the loopback device.

Applications consisting of a process tree having multiple processes usually start with one parent process that can be associated with a certain VRF ID using the provided tools of VRF. However, on spawning child processes, they should remain in the same VRF ID as their parent. Therefore, we extend VRF to inherit the VRF ID in the process control block on forking a child process and to keep the ID

on replacing the process image with the exec system call. Newly created sockets inherit the VRF ID of their process. Thereby we lock a process to its new VRF instance as soon as it leaves the default system VRF with ID zero. This is similar to what BSD jails [KW00] implement for other types of operating system namespaces.

For transparency with routing daemons, we extend system interfaces that access routing tables to transparently work on the specific table of the VRF instance the calling process is associated with. The extended system interfaces consist of IOCTLs and netlink sockets for maintaining the routing table as well as a queuing feature for overhearing packets in user space applications. With regard to routing tables, IOCTLs for modification (SIOCADDRT, SIOCDELRT) and the representing file /proc/net/route in the virtual proc file system now operate on the table the calling process is associated with. This way, virtual routing is completely transparent, e.g. for the user space tool route or a routing protocol daemon even in its unmodified binary form. Further extensions involve sockets of the Linux specific protocol family netlink. The protocol route-netlink serves for modifying kernel routing tables. Its message types RTM_NEWROUTE and RTM_DELROUTE are replacements for the aforementioned IOCTLs. Netlink sockets now inherit the VRF ID of their creating process.

Ad hoc routing protocol daemons often need to overhear certain packets, in order to decide on routing policy decisions. A mechanism for overhearing is queuing packets for a user space process. The second adaptation of the netlink protocol family concerns virtualizing the feature ip_queue of the protocol netlink-firewall. Originally, it provides queuing of IPv4 packets to exactly one process. We extend it to provide queuing of packets to one user space process within each VRF instance. The kernel packet filter implementation of iptables selects packets to be queued.

Finally, there is exactly one global instance of the loopback network device in Linux. In order to make a loopback device available for each vnode, we share the single global instance for each vnode. A frame's VRF ID is preserved so that loopback traffic stays inside the respective vnode.

With the described extensions, we gain full transparency for unmodified network applications including routing daemons. To implement all of the above mentioned functionality, only limited modifications to a standard Linux 2.4.24 source tree are necessary. The modifications comprise 1409 lines of code, which consist of 416 additions, 980 changes, and 13 deletions.

3.6.3 Virtual Node Configuration

Software communication switch (Section 3.6.1) and virtual protocol stack (Section 3.6.2) enable the setup of arbitrary network topologies for large emulation scenarios. Since they introduce a virtual layer in addition to the physical testbed hardware described in Section 2.2.1, we describe bottom up how to configure soft-

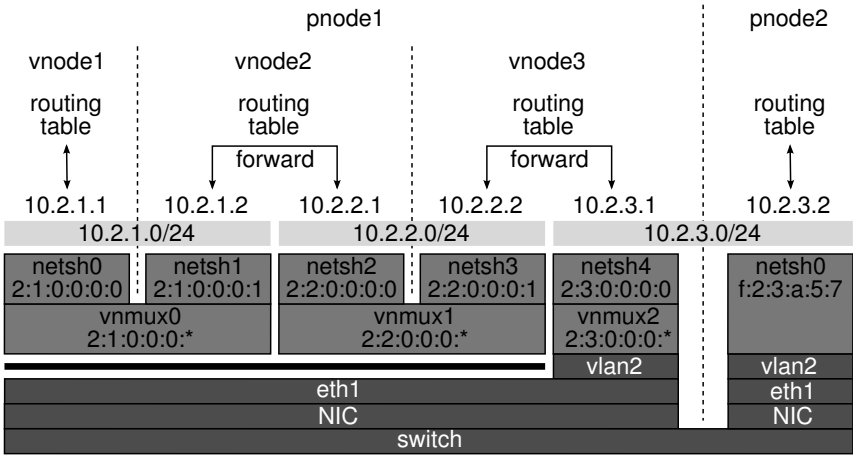

Figure 3.9: Configuration example of link based scenario.

ware switch and virtual protocol stack to integrate seamlessly with a network emulation testbed. Our examples in Fig. 3.9 and 3.10 show two pnodes each. Pnode1 on the left hand side is virtualized into three vnodes vnode1 to vnode3. Pnode2 on the right represents a pnode without any virtualization.

The hardware switch of the emulation network (cf. Fig. 2.4) connects the network interface card (NIC) in each pnode with each other pnode. A NIC is driven by its network device driver shown as eth1. Each network segment (point-to-point link or shared media), that spans multiple pnodes, is represented by a uniquely tagged VLAN. In order to share one NIC for different VLANs, the OS provides one virtual network device for each unique VLAN tag, e.g. vlan2. Even though our architecture supports arbitrary combinations of connection types, we treat link based scenarios separately from shared media based scenarios for simplicity of discussion.

For each virtual point-to-point link within a pnode, we allocate an instance of the software switch, e.g. vnmux0 and vnmux1 in Fig. 3.9. Such pure virtual links do not have any uplink to the hardware switch. Links to another pnode, such as the one over vlan2, have an uplink to the hardware switch utilizing a tagged VLAN. Such physical links do not need a software switch. Vnmux2 could be left out without replacement and netsh4 could use vlan2 directly. Configuring an intermediate software switch anyway might be useful though to simplify scripted configuration by unifying all configuration cases without having to handle the special case of stacking one emulation tool instance onto one physical VLAN uplink differently. Each vnmux owns a unique range of MAC addresses. The first 5 octets of the address identify a vnmux. We use administrator local MAC addresses [IEE05] in order to prevent any collision with the addresses of the hardware NICs. Endpoints of a virtual link are represented by two instances of our network emulation tool providing virtual network devices, e.g. netsh0 and netsh1

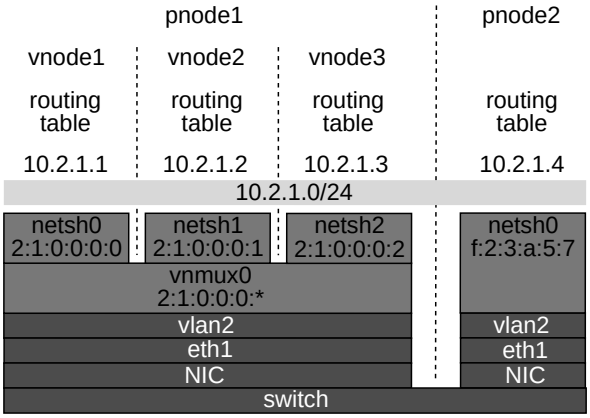

Figure 3.10: Configuration example of shared media based scenario.

as local ports of vnmux0. Without loss of generality, our implementation requires the tool instances connected with a vnmux to share the same first 5 octets of the vnmux' MAC address. The last octet of the MAC address is uniquely chosen for each netsh starting with 0 and strictly monotonically increasing. Tool instances on pnodes without virtualization, such as on pnode2, inherit their MAC address from the underlying NIC. Each tool instance is associated with a certain vnode, e.g. netsh0 with vnode1 but netsh1 and netsh2 with vnode2. On pnodes without virtualization, tool instances are automatically associated with the implicit default vnode comprising the whole pnode. Two endpoints of a link make up their own unique subnet. We show each subnet in classless inter-domain routing (CIDR) notation [FL06], e.g. 10.2.1.0/24. To prevent any accidental routing outside the emulation testbed, we use IP addresses from the private address space [RMK+96]. However, the configuration is by no means limited to those address spaces. Each emulation tool instance gets assigned a unique IP address from the range defined by the corresponding subnet. Reaching targets on subnets other than the own one, requires corresponding routing table entries specifying gateways. Such entries can either be established by dynamic routing using routing daemons or by static routing entries. In our example, vnode2 and vnode3 are pure routers which only forward packets between the two subnets they are connected to. Vnode1 and pnode2 represent end systems.

For each shared media segment, we allocate an instance of the software switch on each pnode hosting connected vnodes, e.g. vnmux0 on pnode1 in Fig. 3.10. The example uses exactly one WLAN channel so there is only one switch instance necessary per pnode. In order to connect vnodes on different pnodes to the same shared media segment, the software switch has an uplink to the hardware switch. All uplinks for a certain segment use the same tagged VLAN. If only vnodes on one single pnode are connected to a certain shared media segment, its software switch

would not have an uplink. As in the link based scenario, each vnmux owns a unique range of MAC addresses of which the first 5 octets identify a vnmux. Instances of our network emulation tool provide virtual network devices that are connected to a shared media segment. For instance, netsh0, netsh1, and netsh2 are local ports of vnmux0. Tool instances connected with a vnmux share the first 5 octets of their vnmux' MAC address and the remaining octet is uniquely chosen as in the link based scenario. Tool instances directly connected to a VLAN inherit their MAC address from the underlying NIC, for instance netsh0 on pnode2. Each tool instance on virtualized pnodes is associated with a certain vnode, e.g. netsh0 with vnode1, netsh1 with vnode2, and netsh2 with vnode3. Each shared media segment gets its own unique subnet, e.g. 10.2.1.0/24. From the range defined by the corresponding subnet, each emulation tool instance gets assigned a unique IP address. Reaching targets on subnets other than the own one, requires corresponding routing table entries specifying gateways. Wireless networks often exhibit dynamic topology changes especially when node mobility is involved. Each vnode connected to a MANET typically executes its own dynamic routing daemon updating the routing table in the OS kernel.

3.6.4 Hierarchical Emulation Control

While having discussed the architecture of virtualization on a single pnode so far, we now discuss key concepts for controlling comprehensive emulation scenarios consisting of multiple pnodes to fully support scalability of our system as a whole.

Each communicating node specified in a network emulation scenario model is represented by a vnode in the testbed. In order to prepare, measure, and cleanup a complete scenario consisting of a potentially large number of vnodes, we provide a central emulation controller (see Sections 2.1.3.1 and 2.2.3). The emulation controller uses communication on the administration network for two types of controlling purposes. First, it executes commands remotely on the pnodes for scenario setup. This includes the creation of virtual nodes and the configuration of the network topology. Secondly, the controller sends dynamic parameter updates to the distributed emulation tools. The controller does *not* process any network traffic but only controls the tools, that actually process the traffic in a distributed fashion.

Since direct control of all vnodes in the scenario does not scale, we use a hierarchical approach. For this, we introduce two proxy processes on each pnode (Fig. 3.11). Since the controller only communicates with those proxies, the number of network connections is limited to the number of pnodes. Such an amount of concurrent connections can easily be handled by the controller. One proxy takes care of remote command execution and passes commands to local vnodes on request. The other proxy demultiplexes update messages for emulation tool instances on the pnode. Additionally, we use a reliable transport protocol (TCP in this case)

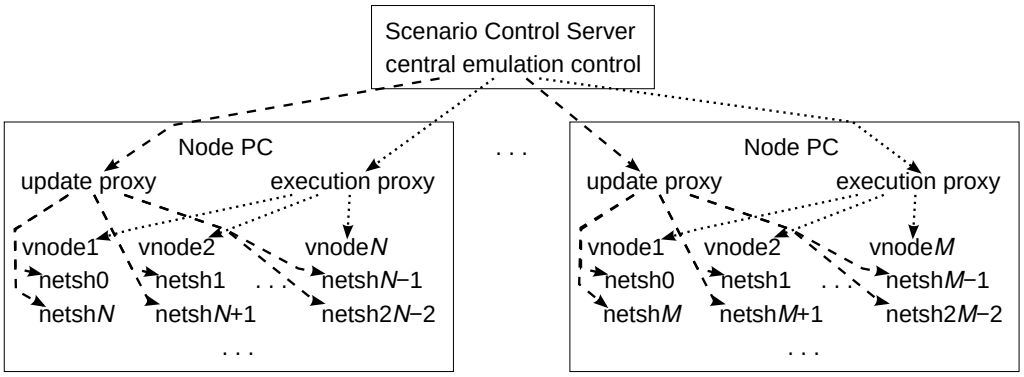

Figure 3.11: Hierarchical emulation control architecture.

for update messages, since unreliable datagram messages would be dropped for scenario sizes of 1000 or more vnodes due to missing flow control.

For MANET scenarios, the controller can provide a graphical user interface (GUI), that visualizes geographic node positions. The user may select a node to continuously display its current radio propagation map as a potential sender [SHR05, Ste09] along with its neighbors as depicted in Fig. 3.12. This helps the user to get a valuable impression of the causality between node movement and dynamic change of network topology at a glance. The update frequency of the GUI is entirely decoupled from the frequency of update messages sent to the emulation tools. In order to ensure that the controller generates update messages in real-time, we assign the update thread a higher priority than the thread painting the GUI.

With large emulation scenarios, the potential frequency of emulation parameter updates to the distributed emulation tools may increase by an order of magnitude or more. Hence, the configuration interface of an emulation tool may require special precautions, which we discuss in the next section.

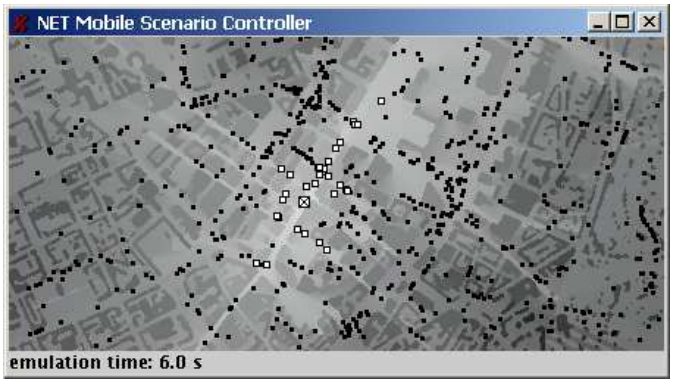

Figure 3.12: Visualization of a MANET emulation experiment ($2225 \times 1065\,\mathrm{m}^2$).

3.6.4.1 Configuration of Emulation Tool

Currently, we use one IOCTL system call for each update of a parameter in an emulation tool as described in Section 2.2.2.5. This implies one context switch for each updated parameter. Especially in scenarios with mobile wireless networks, the update frequency can become comparatively high due to the frequent dynamic change in network topology caused by node movement and wireless transmission (Section 2.2.3).

To reduce the frequency of context switches, an alternative mechanism for transferring parameter updates would be to transfer the whole set of parameters, that get changed at a certain point in time, with just one context switch. This requires the introduction of a new data structure for describing a set of parameter updates to a set of emulation tools, that are local to a pnode. Such a structure can then be handed to the kernel module representing all emulation tool instances on a pnode by different means: entry in the system file system (sysfs), IOCTL, entry in the proc file system (procfs), netlink socket, or custom character device. Entries in sysfs typically represent single parameters, whose values are additionally represented by human readable text strings requiring more memory than an equivalent binary representation. IOCTLs often deal with predetermined fixed size data structures, whereas we have data of dynamic size. Procfs entries transfer data in one chunk at a time having a maximum size equaling the page size, which is unhandy for arbitrarily large data. We consider sysfs, IOCTL, or procfs not ideally suited for our purpose. Netlink sockets are one preferred way for communication between a user space process and an OS kernel component such as our emulation tool driver. However, their ability to provide upwards compatible protocols requires metadata describing message content. Such metadata increases the size of data, that needs to be transferred from user space to kernel space, unnecessarily in our case. A custom character device provided by our emulation tool driver can take data of arbitrary size all at once and of binary structure providing optimally small data size.

The two different parameter update mechanisms in an example MANET scenario imply context switches per update period and per pnode as shown in Tab. 3.3. Each of the v vnodes is connected to $c = 2$ different wireless LANs, i.e. each vnode has c instances of emulated NICs. All vnodes are evenly distributed among $p = 64$ pnodes. We assume the worst case causing the entire network topology to change with each update period, i.e. $v - 1$ parameter updates per NIC. The number of context switches s_1 per update period and pnode with one switch per parameter update is

$$s_1 = \frac{v(v-1)c}{p}. \tag{3.1}$$

With an update set, the number of context switches s_2 per update period calculates as

$$s_2 = 1. \tag{3.2}$$

total vnodes v	**context switches**	
	single update s_1	**update set** s_2
64	128	1
640	12780	1
1000	31219	1

Table 3.3: Worst case number of context switches per pnode within one update period.

3.7 Evaluation

In the following, we provide an extensive evaluation of node virtualization as well as of the entire scalable emulation system. All measurements are performed on pnodes in our testbed equipped with an Intel Pentium 4 2.4 GHz processor, 512 MB RAM, and a Gigabit Ethernet adapter connected to a 32 bit, 33 MHz PCI bus. The experiments in Section 3.7.1 and Section 3.7.3 are conducted on Ethernet adapters, whose drivers do not support interrupt mitigation [MR96, SOK01], and are thus different from those presented in [MHR05]. Experiments in Section 3.7.2 are conducted on Ethernet adapters with driver support for interrupt mitigation.

First, we quantitatively compare different node virtualization approaches by experiment. Next, we evaluate each building block of our chosen scalable network emulation system architecture in micro benchmarks: software switch, node virtualization, and emulation tool. Subsequent macro benchmarks show the scalability of our system in typical network emulation scenarios. Finally, we present two case studies, where our system was successfully used outside of our emulation research project for the evaluation of distributed systems.

3.7.1 Quantitative Comparison of Node Virtualization Approaches

In order to validate the qualitative comparison from Section 3.5 and show that our architecture also fulfills the quantitative requirements from Section 3.3, we conduct experiments allowing a quantitative comparison of virtualization approaches. The

following results have been published previously in a subject study [GWS06] and a conference paper [MGWR07].

Hosting too many vnodes on the same pnode leads to resource contention, which influences performance evaluation results and hence is to be prevented. The overhead of virtualization has a strong impact on the possible degree of virtualization and hence on the total scenario size for a given testbed hardware. In order to allow a quantitative comparison with respect to scalability, we evaluate the efficiency of a representative implementation of each virtualization approach. We restrict our selection of evaluation candidates to open source implementations, that are easily available and work well with our existing NET environment, i.e. Linux as OS and our emulation tool implemented as a Linux kernel module. As a sample for hosted VMs, we evaluate User Mode Linux [Dik00]. Xen [BDF+03] serves as an instance of a classic system VM with para-virtualization. Our implementation [MHR05, MHR07] of the architecture presented in Section 3.6 represents a virtual protocol stack.

We use the Linux distribution Fedora Core 4 (FC4) with the following software version combinations: UML backend of Linux 2.6.16-bs2 hosted on Linux 2.6.11 patched with SKAS3-v8.2 [Gia06] to enable performance optimizations similar to [KDC03], Linux 2.6.11-1.1369_FC4 with Xen-2-20050522 (both from FC4), Linux 2.4.24 patched with VRF 0.100 plus our own virtual protocol stack extensions (Section 3.6.2) together with our software switch (Section 3.6.1).

Our evaluation experiments put focus on the network and transport layer protocol implementations. We consider two types of network: a wired network and a wireless ad hoc network, which applies a different routing protocol.

3.7.1.1 Wired Network Emulation

The network topology of the emulated wired scenario consists of a linear chain with a varying number of router vnodes using static routing. The point-to-point links connecting the routers are full duplex and have an emulated limited bandwidth of

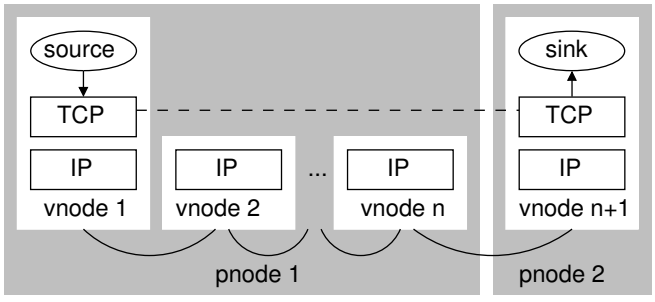

Figure 3.13: Wired infrastructure emulation scenario, virtualized case.

100 MBit/s in each direction. We obtain reference measurements once by executing each vnode on its own pnode. For each virtualization approach, we conduct the same experiment and execute all vnodes on a single pnode except for the last vnode, which resides on a separate pnode without virtualization (Fig. 3.13). Note that the protocol stack of each vnode is involved while a message is passed from source to sink.

On network layer, we measure maximum ICMP RTT (Internet control message protocol round trip time) between vnode 1 and vnode $n+1$. In Fig. 3.14, the reference values without virtualization show linear increase with an increasing number of hops in the routing chain. Both VM implementations cause RTTs that are larger than without virtualization. Due to time slices for the scheduling of guests by the VMM and the necessary virtualization context switches each router introduces additional delay which cannot be compensated. While the mean RTTs not shown here increase linearly with the number of vnodes for all virtualization approaches, the maximum RTT with Xen deviates significantly from the reference values starting with 6 vnodes. These maximum RTTs are several random outliers. They are caused by inevitable serialization on scheduling system virtual machines by the VMM (Section 3.5.1.1). Nondeterministically, the vnode having sent an ICMP echo request packet has to wait an increased duration until it gets scheduled again even though the ICMP echo reply packet is already waiting to be received. This delays the receive timestamp and the RTT appears longer than it actually should be. Virtual routing of NET causes a lower RTT than without virtualization.

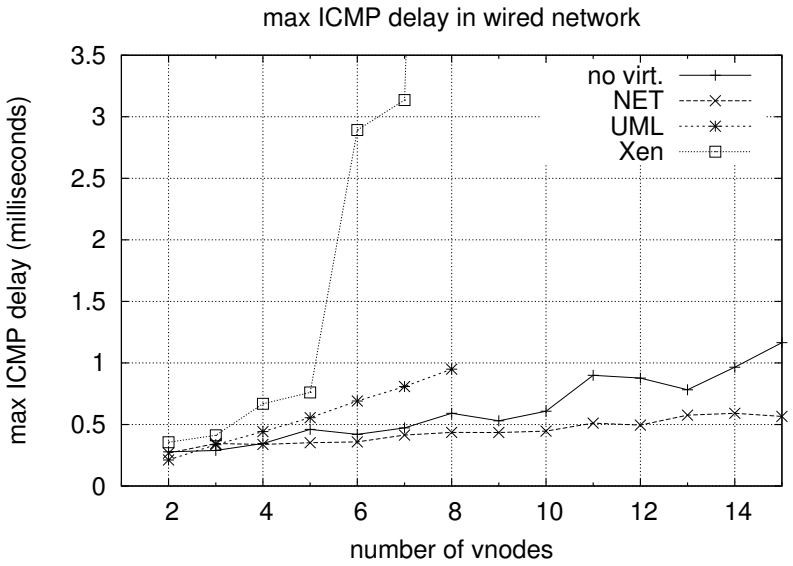

Figure 3.14: Maximum round trip time versus number of vnodes in wired network.

This is because the software switch has lower communication delay than the hardware emulation switch and there are no redundant virtualization context switches. Of course, the emulation tool could compensate for that, if a particular scenario requires inter-node delays to be exactly the same. We did not increase the degree of virtualization beyond 8 vnodes for the VM approaches, since main memory would become a bottleneck. Also their maximum degree lies below 8 vnodes for higher network loads as shown in the next paragraph.

On transport layer, we measure the throughput of one TCP flow over the router chain (Fig. 3.15). The source is located on vnode 1 and the sink on vnode $n+1$ (flow 1). In a subsequent experiment we interchange the source and sink locations (flow 2). Again we measure the throughput of one TCP flow. The earliest undesirable deviations from the reference case happen at a number of 3 vnodes for UML, 5 vnodes for Xen, and 13 vnodes for NET. For all candidates, the deviations are due to contention of the CPU resource only. When the TCP source is on vnode $n+1$ for flow 2, it does not have to compete for resources with other vnodes on the same pnode and is able to achieve a realistic throughput with more vnodes than flow 1 in the opposite direction. UML and NET are affected by this asymmetry, where the remaining resources on pnode 1 suffice TCP receive processing for 3 vnodes with UML or 50 vnodes with NET (the latter is not shown in Fig. 3.15). We assume that the scheduling of the Xen VMM and its CPU resource isolation between guests prevents such a behavior.

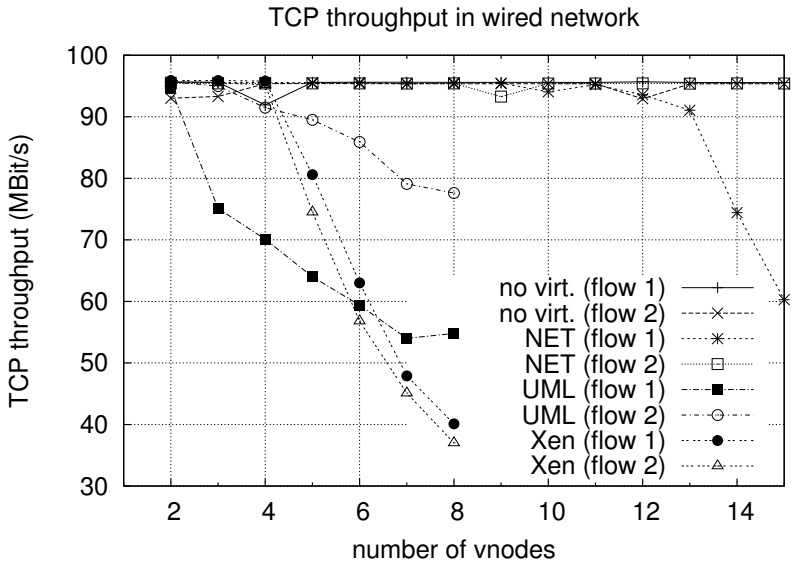

Figure 3.15: Throughput versus number of vnodes in wired network.

3.7.1.2 Wireless Ad Hoc Network Emulation

In the wireless scenario, we arrange the vnodes to also build a linear topology to make the results of our experiments comparable. Fig. 3.16 shows the scenario, where neighbors can communicate directly over a wireless communication link. This is accomplished by a frame loss ratio for ingress traffic of zero for frames from reachable neighbors, and of one for all others. The wireless links between vnodes are full duplex and have a limited bandwidth of 11 MBit/s. We use an implementation of the Ad-hoc On-Demand Distance Vector Routing protocol called AODV-UU [LNT02] in version 0.8 for dynamic routing on each vnode. Similar to the wired network scenario, we perform measurements once without virtualization and for each virtualization approach separately. The measurements are conducted analogously to the wired scenario.

Fig. 3.17 shows the typical behavior of maximum ICMP RTTs for AODV with different hop counts. Starting with one hop, we observe expanding ring search in combination with binary exponential backoff for outgoing route requests. Beyond the default time to live threshold, route requests work without expanding ring search leading to RTTs with linear increase starting at 10 hops, i.e. 11 vnodes. Virtual routing of NET matches the reference values without virtualization closely up to 10 vnodes. Starting with 11 vnodes, the load implied by the AODV routing daemon on each vnode causes increasingly larger RTTs with each additional vnode. UML has a relatively high cost for context switches and hence for each packet transmission. Each additional hop causes an increase in the RTT compared to the reference values. The RTT with Xen is lower than with UML but still larger than the reference values. As in the wired scenario, relatively coarse-granular scheduling of VMs by time slicing in the VMM causes delays that are too large and cannot be compensated. We did not increase the degree of virtualization beyond 8 vnodes for the VM approaches, since main memory would become a bottleneck. Also their maximum degree lies below 8 vnodes for higher network loads as shown

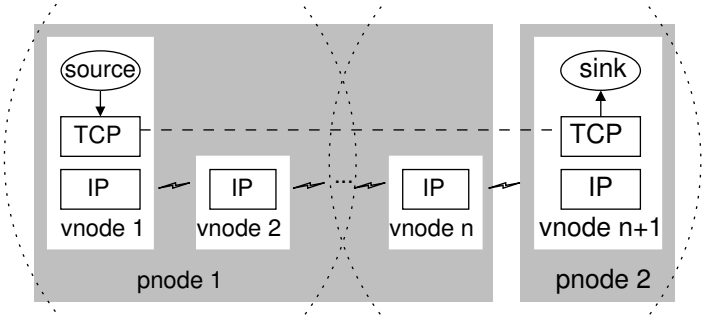

Figure 3.16: Wireless ad hoc network emulation scenario, virtualized case.

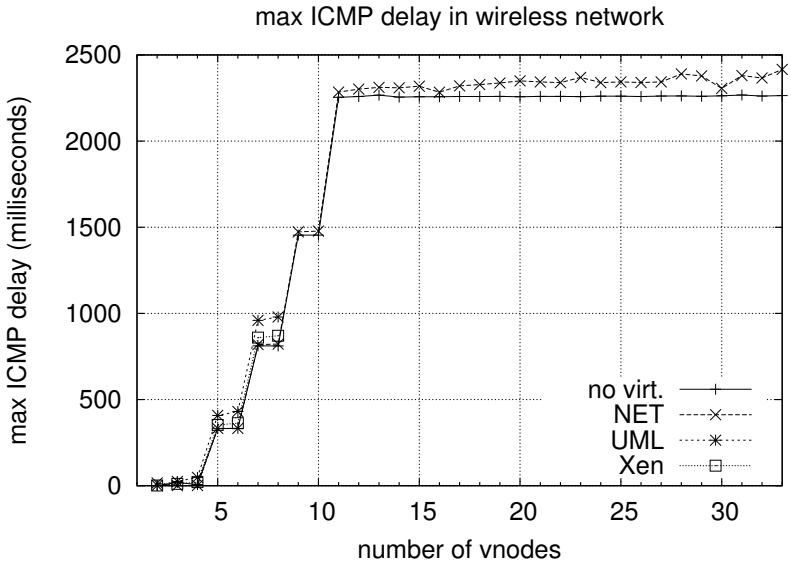

Figure 3.17: Maximum round trip time versus number of vnodes in wireless scenario.

in the next paragraph.

Measurement results for the transport layer are depicted in Fig. 3.18. TCP throughput starts deviating from the reference values without virtualization at 2 vnodes for UML, 7 vnodes for Xen, and 30 vnodes for NET. The emulated bandwidth and thus the network load is lower compared to the wired scenario. However, each AODV routing daemon overhears all IP packets from its two respective neighbors. This causes many guest OS context switches in order to pass received packets from the kernel to the daemon running in user space. Those context switches also involve the VMM and are thus expensive for the VM approaches. Therefore, they only support about the same number of vnodes per pnode as in the wired scenario. Virtual routing of NET however is not affected as strongly by the additional context switches and is able to take advantage of the comparatively low emulated bandwidth.

3.7.1.3 Discussion

CPU is the limiting factor for scalability, i.e. the maximum scenario size, with all virtualization approaches in the evaluation scenarios from the previous two subsections. UML's high context switch costs allow hardly more than one vnode per pnode. Hence, UML is not suitable for scalable network emulation in real-time—not even with applied optimizations similar to [KDC03]. This confirms results reported in [JX03] stating that such a virtualization approach is more suitable

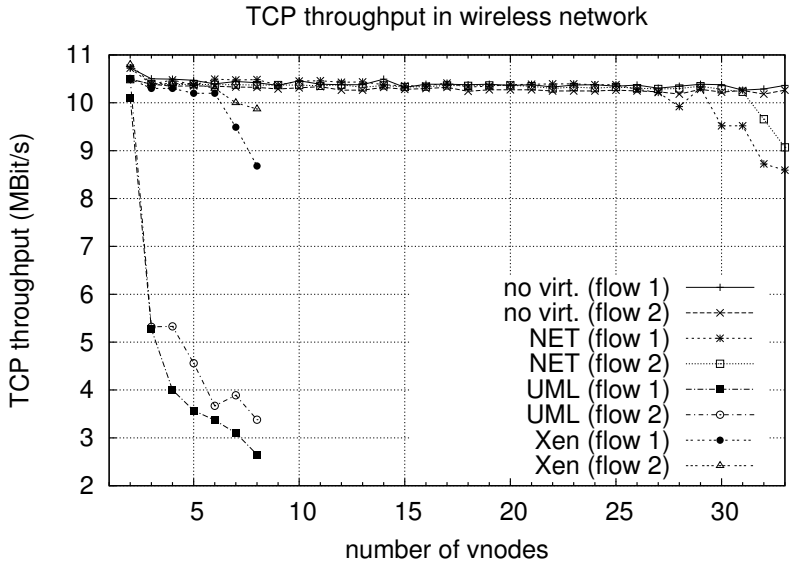

Figure 3.18: Throughput versus number of vnodes in wireless scenario.

for qualitative analysis than for performance measurements. Xen supports 4 to 6 vnodes per pnode on our testbed hardware depending on the scenario. On the one hand, Xen as a classic system VM allows the performance evaluation of arbitrary mixes of software under test on the same pnode, e.g. TCP implementations of Linux, BSD, and Windows. On the other hand, Xen suffers from virtualization context switches that are especially of disadvantage if software under test on the application layer is involved. Our virtual protocol stack architecture shows to be the most efficient alternative. The reduced context switch costs of virtual routing are especially advantageous in emulation scenarios, where software under test on the application layer generates significant traffic. Apparently, a performance analyst faces a trade-off between transparency or flexibility on the one side and efficiency on the other side. We favor efficiency in this thesis, since scalability is the main reason for introducing virtualization in the first place.

3.7.2 Micro Benchmarks

Having shown in the previous section, that our chosen approach to scalable network emulation fulfills not only qualitative but also quantitative requirements, we now evaluate each building block of our architecture in micro benchmarks. Passing through the different protocol stack layers from bottom to top, we start our evaluation with the software communication switch at the data link layer and then show the accuracy of our network emulation tool in variably virtualized scenarios.

3.7.2.1 Software Communication Switch

Our software communication switch is a core component of our scalable emulation environment, since it has to switch frames quickly and at the lowest overhead possible. In order to show that it fulfills the expectations, we measure both the duration of switching decisions and the resulting throughput.

The scenario for measuring the duration of switching decisions consists of two pnodes connected by a point-to-point link (Fig. 3.19). One of the pnodes hosts one switch instance with a varying number of vnode devices attached to its local ports. Although the local ports are represented by instances of the network emulation tool, we do not configure the tool instances to emulate network properties here. There is no bandwidth limitation, no additional delay, and no frame loss, in order not to influence the measurement results for the underlying software switch. For each measured value, the other pnode generates load by injecting 10^5 frames of sizes uniformly distributed between 64 and 1500 octets and randomly targeted at one of the software switch ports.

Fig. 3.20 shows the efficiency of unicast switching decisions. The average measured duration is about 98 ns, independent of the number of vnode devices per switch. The profiled machine code comprises 24 instructions, which take at least about 50 CPU clock cycles on the superscalar out-of-order core of our pnode CPU, if all data is available in the first level cache [Int04a]. At the CPU frequency of 2.4 GHz, 50 clock cycles take about 20 ns. This marks a lower bound for the execution time. Taking cache misses into account, our measured average duration constitutes a reasonable value. A few spikes in the maxima up to 1388 ns are due to cache effects and appear rarely so that the average is close to the minimum of 88 ns. We conclude from these measurements that our implementation of the switching decision is highly efficient.

The scenario for measuring switch throughput consists of one pnode with one switch instance having one uplink and a varying number of vnode devices attached (Fig. 3.21). Again, no network properties are emulated here. We vary frame sizes

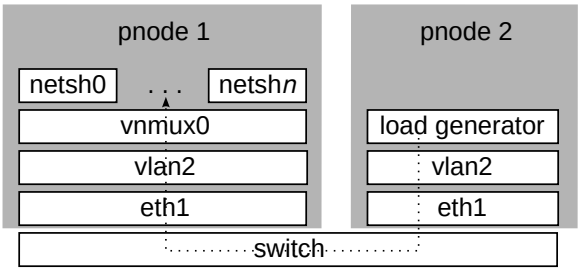

Figure 3.19: Setup for evaluating switching durations.

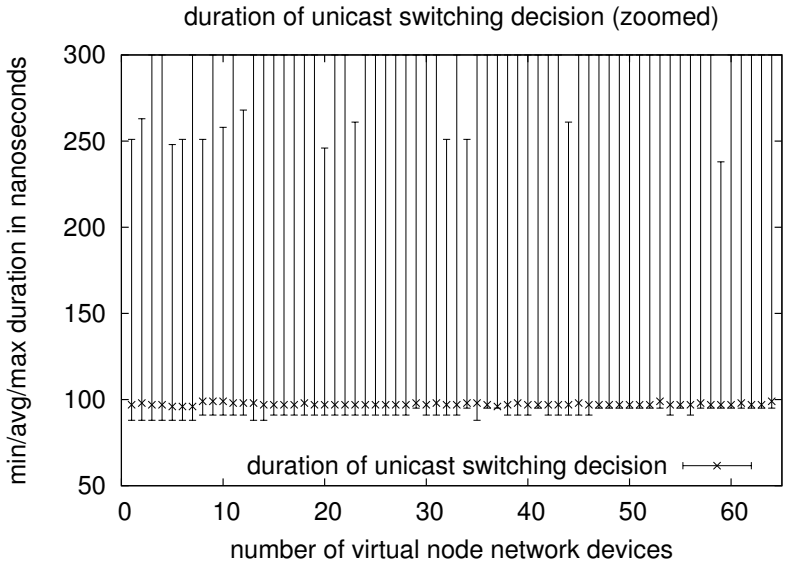

Figure 3.20: Duration of unicast switching decision versus number of vnode devices.

between 64 and 1500 Bytes. For each measured value, we locally inject 10^6 frames of the same size.

Fig. 3.22 shows constant throughput for unicast frames which only depends on the frame size. Small frames imply more overhead and thus less throughput. For comparison, we use the synthetic benchmark program STREAM [McC95] version 4.0 beta from October 1995 to measure sustainable main memory bandwidth. With our available hardware, a testbed computer provides a bandwidth of 1020 MByte/s. Obviously, frame handling overhead is the limiting factor in switch throughput. Nevertheless, a throughput of about 3 GBit/s can serve as an upper limit for aggregate link bandwidth inside one pnode and is 3 times larger than the external uplink over the Gigabit Ethernet network interface.

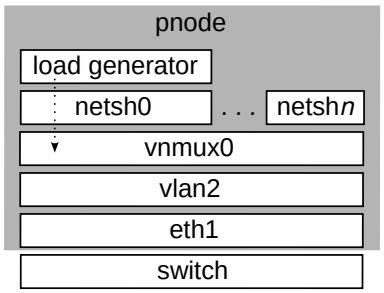

Figure 3.21: Setup for evaluating switching throughput.

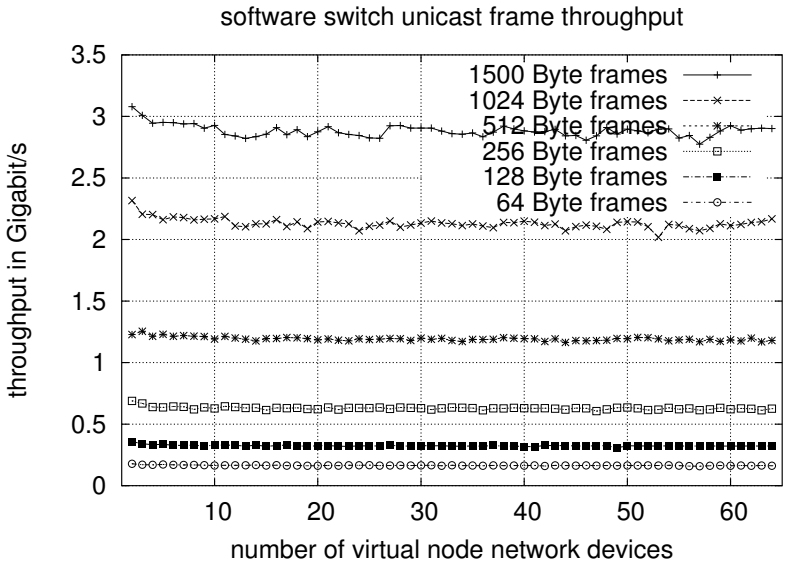

Figure 3.22: Unicast frame throughput versus number of vnode devices.

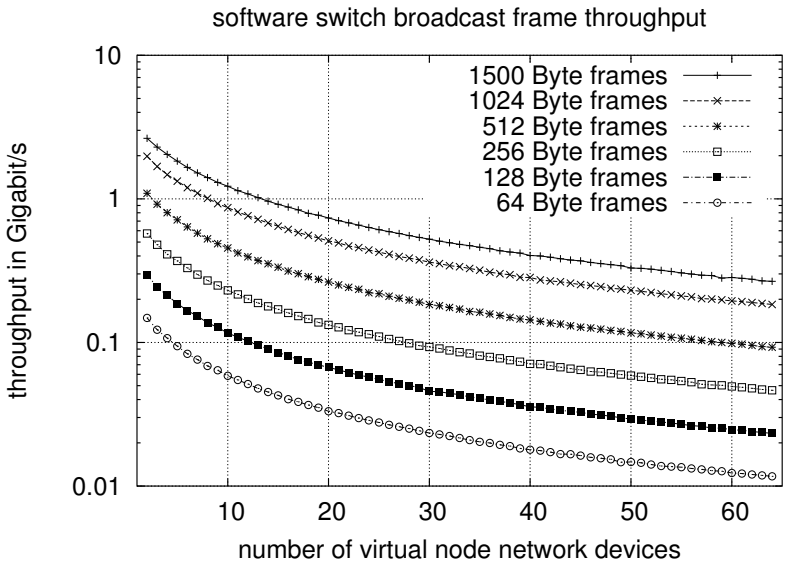

Figure 3.23: Broadcast frame throughput versus number of vnode devices.

For broadcast frames their administration structure sk_buff (not the payload) has to be cloned on delivery for each local recipient. This is necessary since the receive path assumes exclusive administrative frame data structures. We also evaluate switch throughput for broadcast frames. The throughput for the starting value of 2 vnode network devices is slightly lower than for the unicast case because an additional frame clone has to be transmitted on the uplink (Fig. 3.23 with logarithmic scale on the y-axis). With an increasing number of vnode devices per pnode, throughput decreases due to the overhead of cloning. Yet, aggregate switch throughput stays significantly above the memory bandwidth.

3.7.2.2 Network Emulation Tool

Our network emulation tool is able to accurately enforce specified network properties consisting of bandwidth limitation, delay, and frame loss ratio [HR02]. In this section, we show that our tool remains accurate in the virtualized case up to a machine dependent limit for the number of vnodes per pnode.

The scenario for measuring the accuracy of emulated network properties consists of a varying number of vnodes on a single pnode (Fig. 3.24). n vnodes are interconnected in a chain of $n-1$ full duplex links having either limited bandwidth or specific delay in each direction. To measure loss ratio only one direction of each link is configured to lose frames. Each measured value is the result of one emulation run.

In order to measure the accuracy of bandwidth limitation, we use multiple instances of the tool *netio* to put load on each link by measuring maximum TCP throughput concurrently. Fig. 3.25 shows the results consisting of the measured average link bandwidth with minimum and maximum over all links, i.e. TCP flows. Both axes have logarithmic scale. Depending on the number of vnodes, the specified bandwidth is accurately enforced by our network emulation tool. Up to an emulated bandwidth of 5 MBit/s, at least 64 vnodes can be hosted on a single pnode

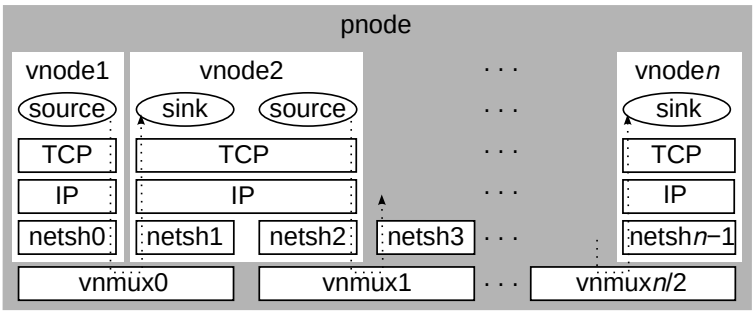

Figure 3.24: Setup for evaluating the network emulation tool.

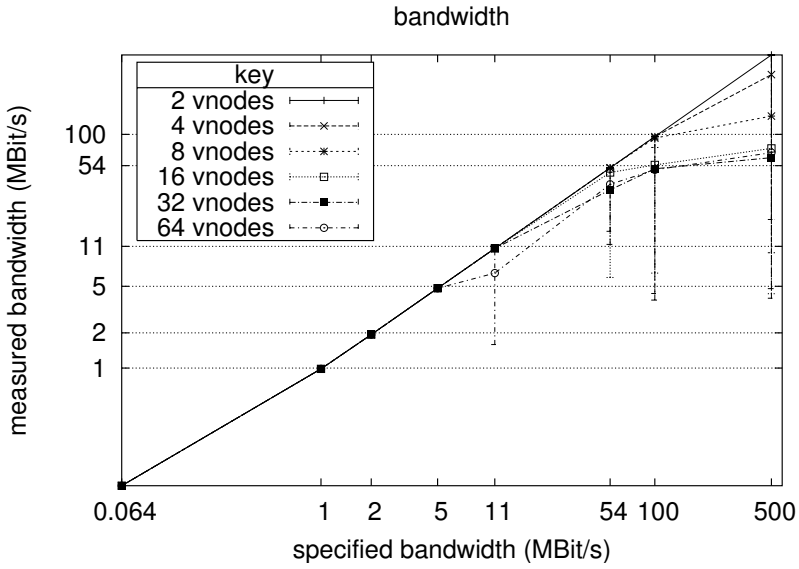

Figure 3.25: Enforced versus specified bandwidth for different numbers of vnodes.

without loss of accuracy. 8 to 16 vnodes can be safely interconnected at 54 MBit/s and at least 4 vnodes can be hosted on a pnode in a Fast-Ethernet scenario with 100 MBit/s.

We measure ICMP round trip times (RTTs) for 10^3 packets on each link concur-

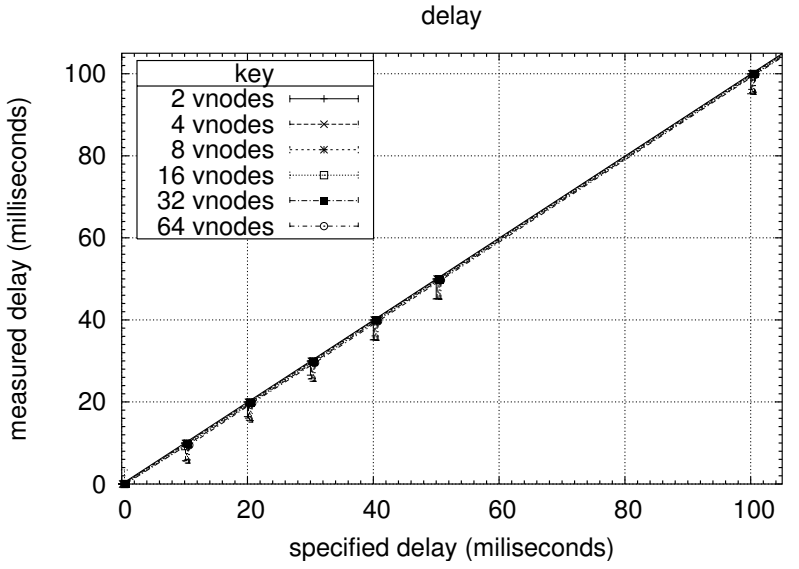

Figure 3.26: Enforced versus specified delay for different numbers of vnodes.

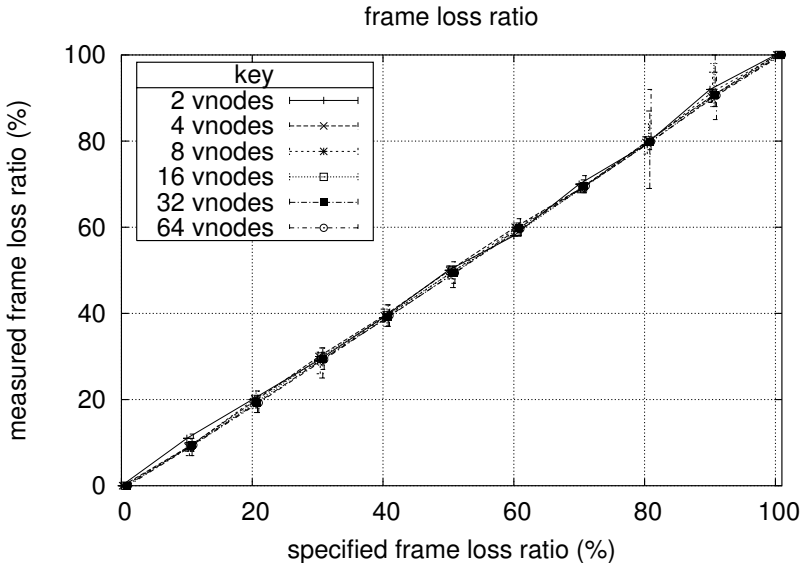

Figure 3.27: Enforced versus specified loss ratio for different numbers of vnodes.

rently to investigate the accuracy of delay emulation. Since the full duplex links are symmetric, the actual delay results from half the measured RTT. Fig. 3.26 shows mean, minimum, and maximum delay over all links. The results indicate that delay is emulated accurately independent of the number of vnodes per pnode. Thus, the emulation of delay scales perfectly with the degree of virtualization. The measured deviations from the average delay values stay within bounds of 5 ms and are due to the granularity of the timer used to introduce the delay [HR02].

Fig. 3.27 depicts results showing the fidelity of emulated frame loss ratio. We put load on each link concurrently using adaptive ping with 10^3 packets. On average, frame loss emulation scales well with the number of vnodes per pnode. Minima and maxima—especially at a loss ratio of 80 or 90 %—are outliers and the deviation is below or equal to 2.7 for all measured values.

We conclude from our measurements that our network emulation tool is able to accurately enforce specified network properties over a wide range of virtualization degrees.

3.7.3 Macro Benchmarks

After the evaluation of each architecture building block, we now show the scalability of our system in typical network emulation scenarios. Some of the evaluation scenarios are the same as in Section 3.7.1. However, in that earlier section, we compared different node virtualization approaches to show that virtual protocol

stacks enable the most scalable network emulation system. In this section, we provide more evaluation details on our emulation system. Load generators on application layer are used and measurements observed on network and transport layer. We consider two types of network requiring different routing algorithms: first a wired infrastructure based network, secondly a wireless ad-hoc network. For the wireless variant, we present one static scenario to draw comparisons with the wired network and one self-contained MANET scenario. The evaluation aims at showing the scalability of the system by comparing the non-virtualized cases to variably virtualized cases of the same scenarios.

Concerning the software under test, we would like to point out that it is by no means limited to protocols on the network layer. After all, our load generators are processes on application layer communicating through sockets with the transport layer. Of course, more complex applications such as peer to peer systems can also be analyzed in our emulation environment without modification.

3.7.3.1 Wired Infrastructure Based Network Emulation

We describe the system model for the evaluation of a wired infrastructure based network emulation scenario first. Subsequently, we present measurement results for the network and the transport layer. Finally, we report on the system utilization caused by executing multiple vnodes on the same pnode.

The network topology of the scenario consists of a linear chain with a varying number of router nodes using static routing. Point-to-point links connecting the routers are full duplex and have an emulated limited bandwidth of 100 MBit/s in each direction. For the same scenario, we conduct the experiments twice differing only in the mapping of the scenario to the testbed hardware. First, we map each router to one real pnode to obtain reference values. Secondly, we place all routers inside vnodes on a single pnode except for the last router, which resides on a separate pnode without any virtualization (Fig. 3.28). Thereby we show that communication over the software switch works transparently, and mixing of

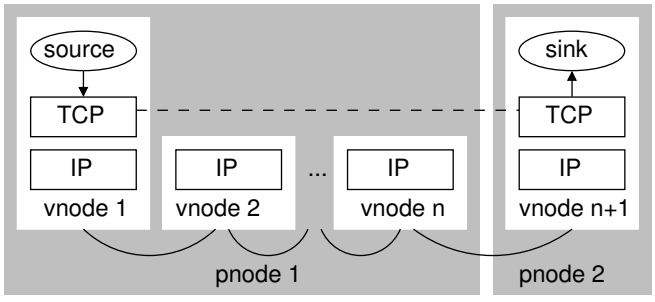

Figure 3.28: Wired infrastructure emulation scenario, virtualized case.

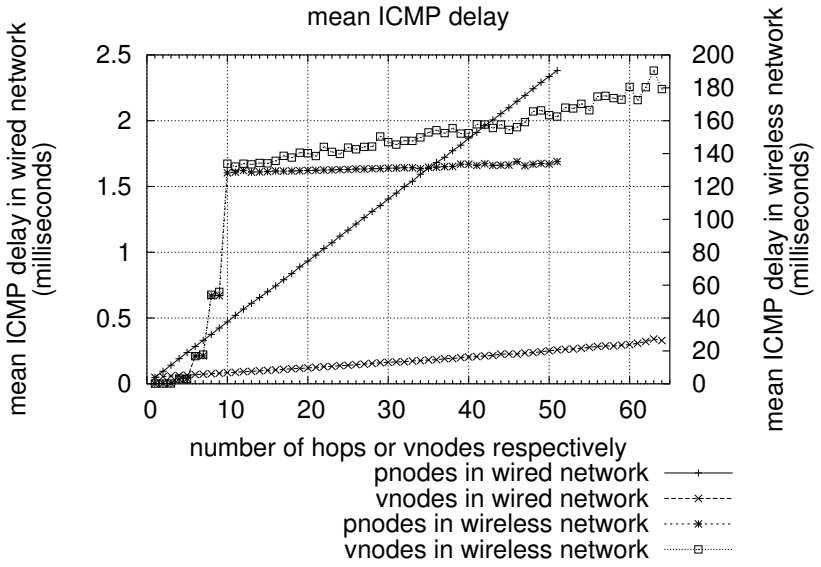

Figure 3.29: Mean round trip time versus number of vnodes.

arbitrarily configured pnodes is possible. Note that the layers of the real protocol stack implementation are always traversed on communication and on forwarding, even if the network traffic does not leave the left pnode except for the last hop.

On network layer, we measure ICMP RTTs through the router chain. Each

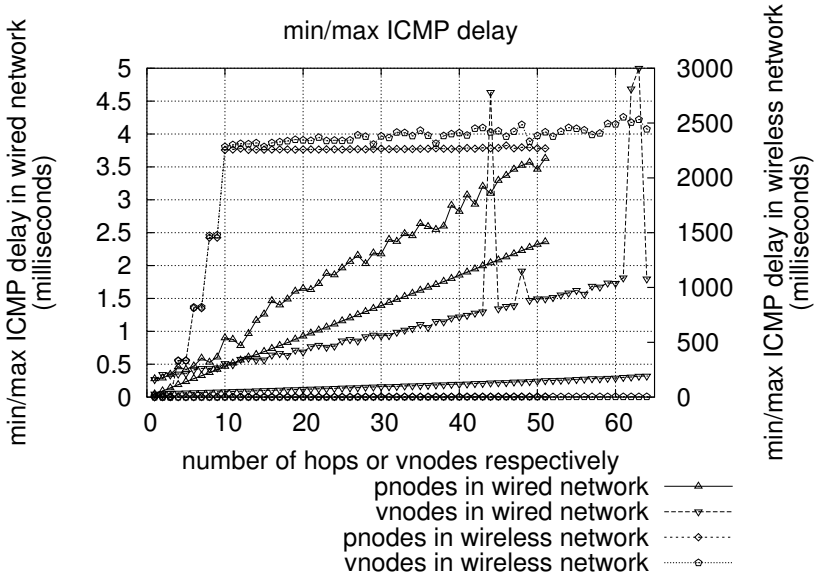

Figure 3.30: Round trip time minima and maxima versus number of vnodes.

measured value consists of one experiment with 10^3 packets sent at a rate of 50 Hz. Fig. 3.29 shows linear increase of the mean ICMP RTT with an increasing number of hops in the routing chain. The left y-axis corresponds to the results of this infrastructure based scenario. The figure also contains measurement results for the wireless ad-hoc scenario discussed in the next section. For the variant with pnodes only, we had 51 of 64 nodes available at the time of the experiment. For the virtualized variant of the scenario, the slope is more flat than with pnodes only. This is because the software switch has lower communication delay than the hardware emulation switch. The emulation tool could compensate for that, if a particular scenario requires inter-node delays to be exactly the same.

RTTs occur within time bounds depicted in Fig. 3.30. Maxima occur only for the first packet until its destination is in the route cache. Thus, mean values from Fig. 3.29 and minima fall close together. Outliers in the maxima, such as with 40 vnodes, are neither an effect of network emulation nor of virtualization. It is normal behavior of the route cache implementation and happens nondeterministically if the fastest code path cannot be executed. It also happens in the unvirtualized scenario variant even though not visible in the presented result set.

On transport layer, we measure TCP throughput over the router chain using the tool *iperf* with a TCP window size (socket send or receive buffer) of 512 KBytes and a maximum segment size (MSS) of 1448 Bytes. For each varying number of vnodes, we measure TCP throughput twice. First, the load generator is located on vnode 1 and the sink on vnode $n+1$ (Fig. 3.28). From the viewpoint of vnode 1, it

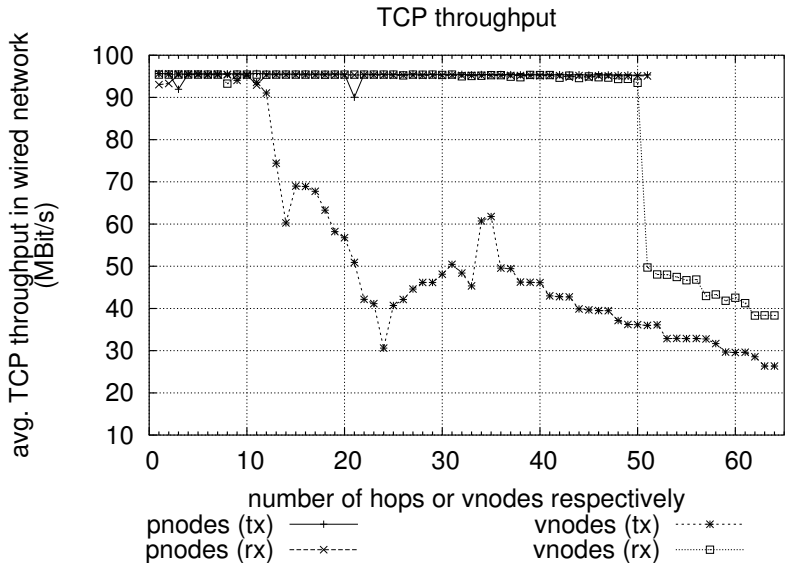

Figure 3.31: Throughput versus number of vnodes.

transmits data to measure the throughput and therefore those results are marked with "(tx)" in Fig. 3.31. Secondly, the locations of source and sink are swapped, i.e. the load generator is located on vnode $n+1$ and the sink on vnode 1. Those results are marked with "(rx)" in Fig. 3.31 since vnode 1 receives data from its point of view. For the virtualized scenario, we observe different behavior for each of those two flow directions. On the reverse direction (rx) from router $n+1$ on pnode 2 to router 1 in a vnode on pnode 1, throughput starts dropping at 51 vnodes due to resource contention. On the forward direction (tx), throughput drops earlier at 12 vnodes. Both the TCP sender with its timers in the leftmost vnode and all the other vnodes compete for the same resources of their shared pnode. TCP receive processing is no more the heavier side with modern CPUs [FHH+03]. Given the same resources, the TCP sender supports only lower throughput than the receiver could process.

Measurements for throughput show deviation from the reality, if too many vnodes are hosted on the same pnode. In order to gain insight into system utilization, we measure the remaining idle performance on the pnode hosting vnodes while executing the two previously mentioned experiments. For this purpose, a custom tool consumes all available idle time quite aggressively while the load generating process of the previous two experiments is active. The tool reports on its compute progress which corresponds to the remaining idle compute time. The results of all measurement runs are normalized to the result for one vnode per pnode resembling the unvirtualized case. Fig. 3.32 shows only one plot per throughput measurement,

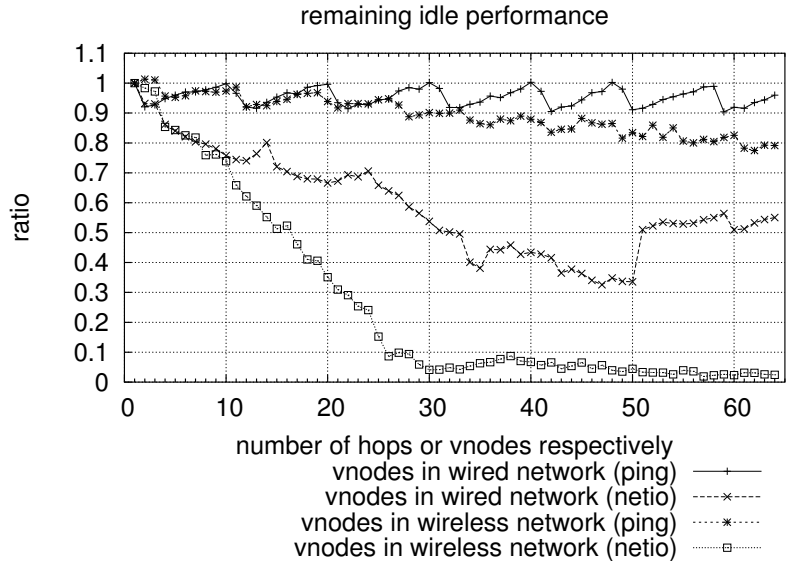

Figure 3.32: Remaining system compute performance versus number of vnodes.

since the tool *iperf* measures both flow directions (rx and tx in Fig. 3.31) back-to-back. In general, the remaining idle performance decreases with an increasing number of vnodes per pnode. This is an indicator for possible resource contention due to virtualization.

Hosting too many vnodes per pnode leads to severe resource contention which can lead to measurement artifacts. Since we are interested in realistic results, the number of hosted vnodes is limited. For the above measurements, the earliest undesirable deviation from the unvirtualized case happens for TCP throughput at a number of 12 vnodes (Fig. 3.31). Given the scenario above and our testbed hardware with 64 pnodes, we thus can support scenario sizes of up to 704 nodes for similar wired scenarios.

3.7.3.2 Wireless Ad hoc Network Emulation

Wireless ad hoc networks typically consist of a large number of communicating devices, which are often resource-poor. By using node virtualization, wireless ad hoc scenarios can be emulated with more devices than computers in an emulation testbed. Hence, we evaluate the scalability of our approach for the emulation of such scenarios. As before, we describe the system model, followed by evaluation results for the network and transport layer as well as for system utilization.

Fig. 3.33 shows the emulation scenario. For comparison with the infrastructure scenario, we configure the virtual node positions and the emulated wireless network transmission range—depicted by dotted circles—such that the connectivity of the nodes resembles a chain. This is accomplished by a frame loss ratio for ingress traffic of zero for frames from reachable neighbors, and one for all others, using the mechanism described in Section 2.1.3.2. The wireless links between nodes are full duplex and have a limited bandwidth of 11 MBit/s. Here, we do not emulate the effects of a MAC layer, i.e. there are no frame collisions. Incorporating a MAC layer emulation as mentioned in Section 2.1.3.3 requires more resources and

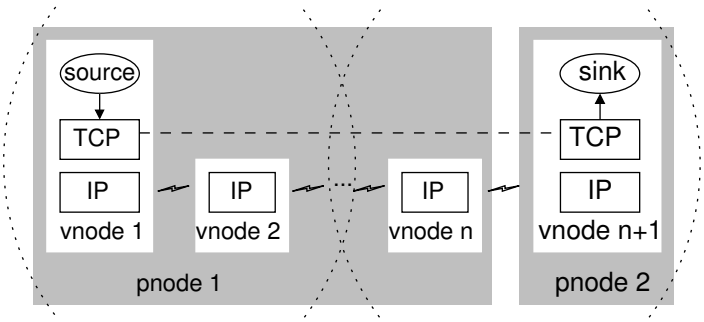

Figure 3.33: Wireless ad hoc network emulation scenario, virtualized case.

could reduce scalability. In this scenario, we use an implementation of the ad hoc on-demand distance vector routing protocol called AODV-UU [LNT02] in version 0.8 as software under test. On each node, an instance of the routing daemon is executed transparently with virtually no modifications necessary. Similar to the infrastructure scenario, we measure this scenario once with only pnodes and once with all vnodes on a single pnode, except for the last node, which resides on a separate pnode. The workload is the same as for the infrastructure scenario.

On network layer, we measure ICMP RTTs between node 1 and node $n+1$ in the scenario. The right y-axis in Fig. 3.29 corresponds to the measurement results for mean RTTs. Starting with one hop, we observe expanding ring search in combination with binary exponential backoff for outgoing route requests as described in [PR99], which is implemented by AODV-UU. Beyond the default time to live threshold, route requests work without expanding ring search leading to RTTs with linear increase starting at ten hops. For the virtualized variant, the slope is slightly larger for maxima and thus also mean RTTs due to increased latency on route establishment if multiple routing daemons compete for the resources of the same pnode. The right y-axis in Fig. 3.30 corresponds to minima and maxima in RTTs. Maxima resemble multiples of the mean values due to route cache misses on route establishment. Minima show very flat linear increase being observed for established routes with route cache hits.

Measurement results for the transport layer are depicted in Fig. 3.34. TCP throughput starts deviating from the reference values in the unvirtualized scenario

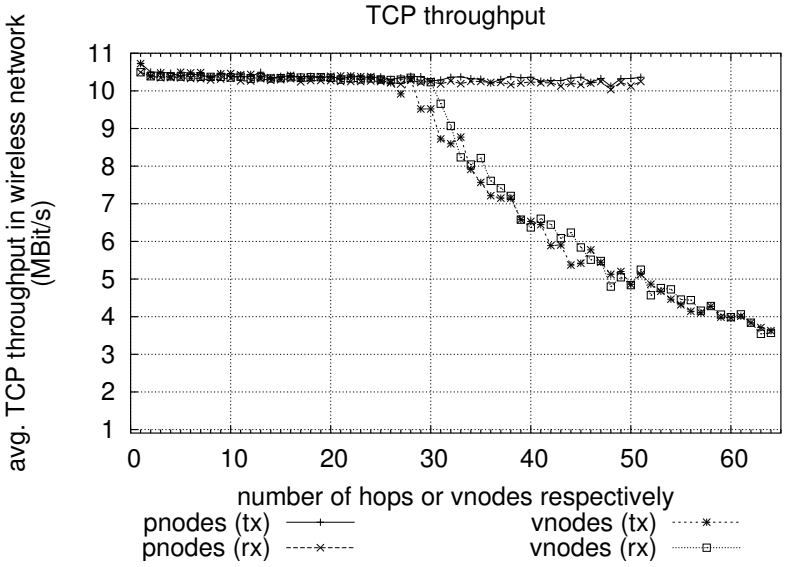

Figure 3.34: Throughput versus number of vnodes.

variant at about 29 hops. As a result of the lower limited bandwidth compared to the infrastructure scenario in the previous subsection, more virtual nodes can be executed on a pnode without interference.

Remaining idle performance for the virtualized wireless scenario is shown in Fig. 3.32. In addition to the forwarding operation on network layer as per the previous infrastructure based scenario, one ad hoc routing daemon is executed on application layer on each vnode. This results in higher system utilization, the more vnodes are hosted on a pnode. The consequence is remaining idle performance of about 10 % at a maximum of 28 vnodes.

The earliest undesirable deviation from the realistic reference values happens for TCP throughput at 29 vnodes per pnode (Fig. 3.34). Given the above scenario and our testbed hardware with 64 pnodes, we thus can support scenario sizes of up to 1792 nodes for similar wireless scenarios.

3.7.3.3 Comprehensive MANET Emulation

As a complement to the previous evaluations primarily investigating scalability, we show the utility of our system by means of an example for a scenario that is universally useful for developers to analyze the performance of a real protocol implementation.

We follow [BMJ+98] for the scenario model and evaluate AODV-UU 0.8 according to the metrics: packet delivery ratio and routing overhead. Each metric is plotted along the y-axis of the respective measurement result chart (Fig. 3.35 and 3.36). Packet delivery ratio is the fraction of delivered data packets and sent data packets. The optimum is represented by a value of one. The value is usually lower due to packet loss, e.g. caused by dynamic network topology changes. Routing overhead is the number of control packets transmitted by the routing algorithm to establish and maintain routes.

Each emulation run comprises 50 mobile nodes inside an area of $1500 \times 300\,\mathrm{m}^2$ and lasts 900 s. Each of 20 nodes starts uniformly distributed between 0 and 180 s generating constant bit rate load with 64 Byte packets at 4 Hz via UDP to one randomly chosen peer.

We vary the pause time of node mobility traces over 6 different values of 0, 30, 120, 300, 600, and 900 s. These are plotted along the x-axis of the charts. In a mobility trace, a node chooses a destination and travels to it with random velocity up to the maximum. On reaching its current destination, a node waits for the pause time, chooses a new destination and travels to it as described, repeating those steps until the end of the emulation run.

We also vary the maximum velocity over 2 different values of 20 m/s and 1 m/s. These are denoted by “max speed” followed by value and unit in the key of the charts.

For each of the above parameter variations, we compare the same implementation in three different settings, resulting in six measurement curves per chart due to the two different maximum velocities. Each data point of a measurement curve is the average of two runs with a different node mobility trace.

First, for comparison with [BMJ+98], we obtain reference measurements denoted by "(ref)" in the key of the charts. The nodes move according to the random waypoint mobility model and communicate within a fixed transmission range of 250 m based on free space propagation. In order to prevent influencing measurement results with potential resource contention, we place exactly one vnode per pnode denoted by "1 vnode/pnode" in the key of the charts. For both node velocities of 20 m/s and 1 m/s, the packet delivery ratio in Fig. 3.35 is comparable to [BMJ+98]. Routing overhead in Fig. 3.36 is different from [BMJ+98], since the AODV implementation there does not use "hello" messages for neighbor detection. We believe that this pays off for low speeds (1 m/s) but causes more overhead at high speeds (20 m/s).

Secondly, we employ a more realistic node mobility pattern and wave propagation model to infer how these influence routing algorithm performance. The nodes move along streets and choose destinations on them. The necessary geometric data corresponds to a part of the Stuttgart city center, for which we precomputed realistic wave propagation taking static obstacles into account [SHR05, Her05, Ste09]. A hierarchical emulation controller (see Section 3.6.4) maintains the dynamic aspects of the scenario such as node mobility and resulting connectivity. We do

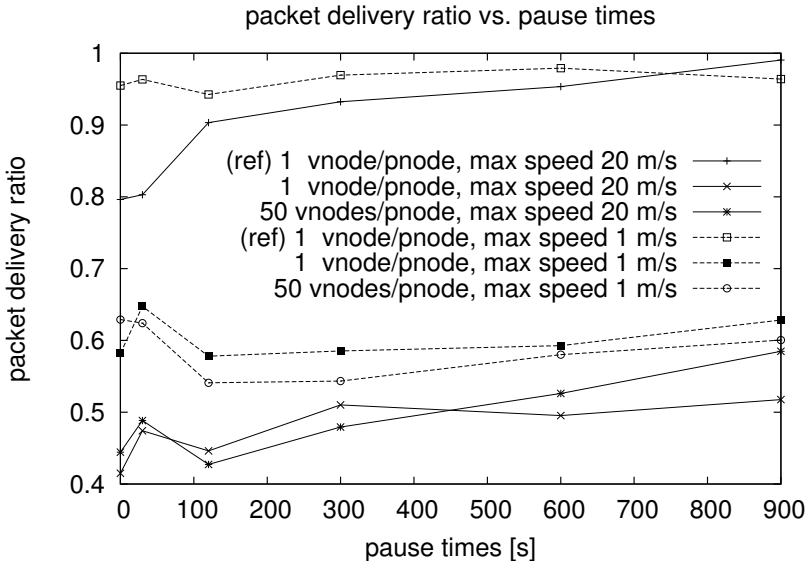

Figure 3.35: Packet delivery ratio versus pause times.

not consider effects of the MAC layer (cf. previous subsection). Again, we prevent influencing measurement results with potential resource contention by placing exactly one vnode per pnode denoted by "1 vnode/pnode" in the key of the charts. The packet delivery ratio in Fig. 3.35 is significantly lower than with the reference measurements because there is less connectivity due to obstacles. For the same reason, routing overhead in Fig. 3.36 is a multiple of the reference measurements.

Thirdly, we use the second scenario setting again but map all 50 nodes to a single testbed computer by means of node virtualization. This allows for the evaluation of the scalability of our emulation approach in such a real life scenario. Slight differences to the measurement results of the unvirtualized runs of the previous paragraph are due to inevitable nondeterministic behavior of the distributed system represented by the emulation testbed. Low traffic load in the scenarios and sufficiently comparable results for different amounts of virtualization lead us to the conclusion that such a real life scenario can be emulated on just a single pnode.

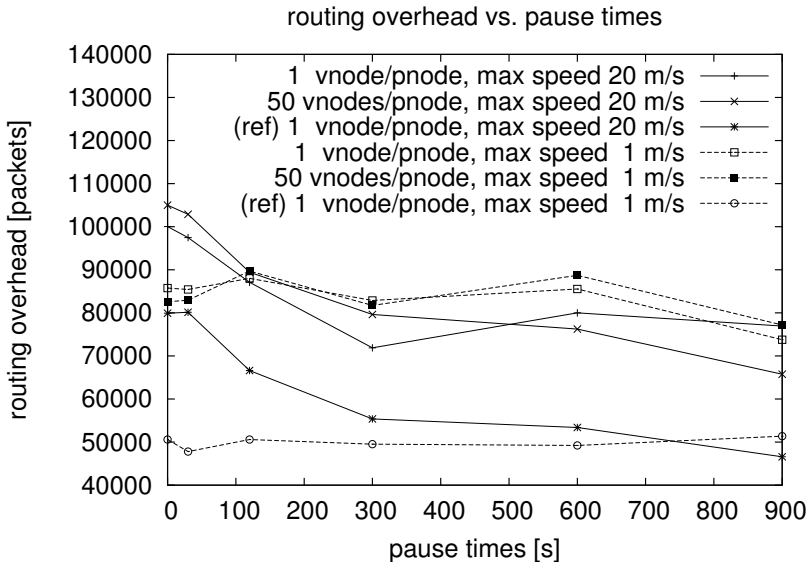

Figure 3.36: Routing overhead versus pause times.

3.7.4 Application Case Studies

Outside the NET research project, our scalable emulation system as described in this chapter, has been successfully used for performance evaluation of large scenarios with more nodes than assigned testbed computers in two PhD theses.

Schiele [Sch07] evaluated an implementation of a novel energy efficient service discovery for pervasive computing systems without need for external network

infrastructure. The emulation scenario comprised 40 nodes. Measured metrics were message overhead and other application specific metrics. Additionally, NET has been extended to dynamically report energy consumed by transmitted frames, in order to support energy consumption as another measured metric.

Wacker [Wac08] evaluated implementations of novel decentralized key distribution schemes, that enable the usage of symmetric cryptography for secure communication between highly resource-constrained devices. The emulated scenarios comprised between 25 and 250 nodes. Measured metrics were memory usage and message overhead.

3.8 Summary

We motivated node virtualization as a method to scale network emulation enabling performance evaluation of large network scenarios with many communicating nodes. Based on our three requirements to node virtualization—transparency, flexibility, and efficiency—, we discussed possible virtualization approaches. Of the three possible candidates—hosted virtual machine, classic system virtual machine (with para-virtualization), and virtual protocol stack—, we identified virtual protocol stacks as the best approach regarding our requirements. Our architecture consists of software switch, virtual protocol stack, and hierarchical emulation control. Extensive evaluation results showed the scalability of the building blocks of the architecture as well as of the whole network emulation system.

Chapter 4

Resource Contention in Virtualized Emulation

In this chapter, we discuss methods for detecting resource contention in scalable network emulation based on node virtualization. The detection of resource contention allows filtering of emulation runs with potentially unrealistic measurement results and thus ensures that only realistic emulation results are considered. After an introduction, we present related work and requirements for detecting resource contention. Considering different approaches, we derive definitions for quality criteria, that allow the characterization of resource contention. Our monitoring approach enables detection of resource contention based on our quality criteria. Finally, we evaluate the effectiveness of our quality criteria as well as the effectiveness and overhead of our monitoring approach.

4.1 Introduction

With node virtualization from the previous chapter, each instance of software under test is executed in a vnode. Vnodes on the same pnode share the limited resources of their hosting pnode. This may lead to resource contention, which can cause unrealistic measurement results by influencing an emulation run and is thus undesirable. Similar to the strategies for dealing with deadlocks in operating systems [Tan01], there are four basic approaches to cope with undesired resource contention: ignoring, prevention, avoidance, or detection and recovery.

Obviously, ignoring is not a solution since network emulation needs to provide realistic measurement results to be useful.

Some early approaches [WLS+02, ZN03a] choose to prevent resource contention by over-provisioning of resources. However, we think that this restricts scalability

of virtualization based emulation testbeds unnecessarily. There are many scenarios, where the testbed resources are not used to their full extent.

More recent approaches [JX03, ZM04, AH06] avoid certain kinds of resource contention between vnodes. Typically this is implemented for the CPU resource class. The scheduler ensures that each vnode is limited to consume a certain maximum percentage of available CPU time. Thus, each vnode gets its fair share. In the simplest case, all vnodes get an equal share. In general, each vnode can be assigned a weight to have different shares for the vnodes. Virtual machine approaches partition the main memory resource among the vnodes such that each vnode has its own exclusive fraction and thus even prevent contention for memory between vnodes [JX03, AH06]. In contrast, the virtual protocol stack approach Imunes [ZM04] limits main memory allocation pools for each virtual protocol stack instance, but assumes cooperative sharing of memory used by processes and by emulation tools to store frames in their delay queues. Disk resources, especially the scheduling of I/O requests, are not considered or are planned as future work by all approaches. Fair scheduling is feasible with scalable network emulation approaches based on virtual machines since the virtual machine monitor has ultimate control over the entirety of each resource class it manages. However, such approaches limit scalability due to the virtualization overhead for providing the illusion of full virtual machines. More scalable virtualization approaches for network emulation would require substantial modifications to existing software systems to allow for fair resource scheduling. While extending the process scheduler to provide fair scheduling with one process group for each vnode is transparent to software under test in the protocol stack, additional places require the consideration of adapted scheduling. Specifically, asynchronous I/O processing in an operating system [ZM04], such as transmitting frames from a network device queue and processing received frames in the protocol stack, consumes CPU resources not taken care of by the process scheduler. Also protocol specific timers, such as the TCP retransmit timer, consume CPU outside the process context. Introducing fair resource usage per vnode here would require the introduction of previously absent resource scheduling. Thus, it contradicts the desire for transparently supporting unmodified existing system software as test subjects. While the above-mentioned fair scheduling approaches prevent each vnode to consume resource fractions reserved for other vnodes, they do neither avoid nor detect the consumption of an entire resource by the union of all vnodes on a computer. In such cases, each instance of the software under test suddenly perceives a system with capped resource power. The experimenter is not informed that a resource has been consumed entirely even though this may have influenced measurement results.

Having scalability as our paramount goal, we pursue a solution based on

detection of undesirable resource contention with subsequent recovery. Detection of resource contention is enabled by monitoring certain emulation quality criteria.

4.2 Related Work

Detection of resource contention during emulation experiments, requires resource monitoring. Therefore, we review existing performance monitoring approaches in this section. We classify approaches into three categories: network monitoring, scalable network emulation, and code instrumentation.

4.2.1 Network Monitoring

Regarding the available network resources of an emulation system, it stands to reason to detect contentions by monitoring for potential bottlenecks. There are two basic approaches for monitoring: Either active by sending probe traffic into the network or passive by observing regular payload.

End-to-end bandwidth probing techniques investigate Internet path characteristics by actively sending probe packets into the network. The base mechanism is called packet pair. A sender sends a pair of packets back-to-back. The serialization delay, based on the length of the second packet and the bit rate at the sender, determines the time interval between the last bit of the two packets. On the bottleneck link, the bit rate is lower than at the sender. This causes an increased serialization delay and thus an increase of the time interval between the last bit of the two packets. The time interval is preserved on the path to the receiver since the remaining links have a bit rate equal to or larger than the bottleneck link. On receiving each of the packets, the receiver echoes the packet. Thus, the receiver preserves the time interval. On the path back to the sender, the time interval remains unmodified, since it was already increased on the path from sender to receiver. The packet length and bottleneck link bit rate remain the same such that the time interval is not further increased. Finally, the sender can measure the increased time interval and deduce the bottleneck link characteristics taking the packet length into account. Originally, the first approaches such as pathchar [Jac97] provide means to measure the capacity of the bottleneck link on a network path between two communication partners. Later bandwidth probing approaches, such as packet transmission rate, provide means to measure the available bandwidth on a network path [HS03]. This could be used to detect resource contention when the available bandwidth reaches zero. Inherent active probing with all these approaches means that additional network load flows over the emulation network, which is otherwise restricted to only carry payload of the software under test.

Passive network monitoring approaches circumvent additional network load by just observing regular network payload. An example for such an approach is SPAND [SSK97, SKS00]. Client applications observe network performance metrics for all their external communication partners. Clients within the same network domain report those metrics to a shared performance server. A packet capture host may sniff the traffic of a network domain and apply heuristics to generate performance reports even in the absence of client applications modified to generate reports themselves. Clients may query their local performance server for the network performance to reach certain communication partners. The performance reply may be used to choose among mirror servers serving the same content, to choose among different content representations having different quality, or to provide user feedback about the performance to expect when communicating with a remote server. The SPAND approach suits web browsing or bulk transfer applications rather than streaming applications. Therefore, it cannot easily be applied to network emulation with arbitrary traffic patterns in general.

Wren [ZL04] passively monitors the capacity of the bottleneck link by observing bursty traffic of high-performance grid computing applications. The Virtuoso project provides an execution environment for grid applications consisting of virtual machines. The virtual machine hosts are interconnected by VNET [SD04], a virtual overlay network to connect VMs and provide an illusion as if the VMs were part of the LAN where the user is located with storage and user interface to his distributed grid application. On startup, the VM hosts communicate in a star topology over a central VNET proxy in the user's LAN. Within Virtuoso, the virtual topology and traffic inference framework (VTTIF) [GD04] infers the communication topology of parallel programs. The overlay network can be adapted to the communication topology by adding virtual links between those VM hosts, that show significant communication with each other [SGD04]. Integrating both Wren and Virtuoso [GZS+06], allows to adapt both the mapping of VMs to hosts and the overlay network to the underlying host resources (CPU and memory) as well as to the provided physical network resources (bandwidth and delay). Instead of adaptation, the comparison of bottleneck link capacity and throughput of application traffic could be an approach to detect contention of a network resource.

Approaches such as SPAND and Wren observe transport layer metrics, which cannot necessarily be observed transparently for software under test in network emulation operating on the data link layer. Therefore, such passive network monitoring approaches are not suitable for monitoring emulation quality criteria.

Traffic accounting approaches represent another passive network monitoring class. They estimate frame or packet counts statistically for each source by sampling switched flows on a packet-basis [JPP92] as opposed to a time-basis. sFlow [PPM01] is an industry standard employing packet-based sampling and describing

an architecture to monitor large networks. In this paragraph, we use the terms router or packet also as substitutions for switch or frame respectively. An agent in each router maintains a counter variable. The variable is initialized with a random value where the average of the random number sequence should converge to a fixed sampling rate. For each packet that is not dropped, the counter is decremented and the number of total packets is incremented. If the counter reaches zero, the counter gets reset with a new random value, the number of samples taken is incremented, and a sample of the current packet is taken. The sample of the packet could consist of its protocol headers and is sent to a central analyzing instance. In effect, the random counter is used to skip packets which should not be sampled. Such an approach is best suited to answer questions for the largest flows, i.e. a proper subset of all traffic in the network. Since it is based on sampling, it is difficult to detect short peaks which can cause resource contention.

The quality of service (QoS) of networks can be monitored passively. Monitoring can be achieved with sampling or with an event driven approach [SB96]. With sampling it is difficult to determine the data collection rate. Also, sampling implies an inherent overhead due to periodic timer interrupts and the overhead is even present in idle systems. Therefore, an event driven approach is desirable to achieve a scalable solution with minimal overhead. Independent of the monitoring approach, providing a certain QoS level means prevention or avoidance of resource contention. Monitoring can be used to observe the fulfillment of a certain QoS level. As mentioned in the introduction, prevention or avoidance of resource contention restricts our paramount goal scalability.

With virtualized network emulation, the available network bandwidth of the software communication switch (Section 3.6.1) depends on the CPU load of the hosting pnode. The CPU load in turn depends on the consumption of other vnodes hosted on the same pnode. An intermediate system with CPU contention causes additional packet delay on top of the regular serialization delay on its outgoing link. An end system may similarly delay packet processing. Network monitoring may interpret the larger delay as e.g. less available bandwidth on the paths containing that link. However, such approaches detect CPU contention only indirectly if it happens to cause network contention. They cannot detect CPU contention in general, e.g. pure computation or computation related to forms of I/O other than network such as disk. Since these approaches only monitor the network resource, they cannot distinguish between different resource classes such as network, CPU, memory, or disk.

We consider network monitoring approaches to be possible software under test that can be executed as part of a network emulation experiment. Running such an approach within an emulation environment can also be useful to validate a scenario setup and emulated network properties.

4.2.2 Scalable Network Emulation

ModelNet [VYW+02] is a parallel network emulator. It is primarily designed to emulate a given network topology of point-to-point links. The topology is partitioned among a cluster of emulation computers, the so-called router core. Each cluster node processes network packets through internal arbitrarily connected links and routing instances. Since processing of packets already inside the router core takes precedence over receiving new packets from outside the router core, the system starts dropping packets entering the core when the CPUs are fully loaded. This way, ModelNet is able to detect CPU resource contention in its core. While such detection works for centralized parallel emulation, it is not applicable to distributed network emulation, where not only the emulation tool competes for CPU resources but also the software under test. So-called edge nodes are connected with the router core and execute the software under test. In order to scale the number of software under test instances, multiple instances may be executed as processes on each edge node. The authors use the fraction of computation and network I/O as a metric to report how many instances can be hosted by one edge node. With increasing instructions per byte, context switching overhead reduces the maximum number of software under test instances per edge node. However, the ratio of computation and network I/O depends on the individual software under test. The authors suggest to determine the ratio in advance. This benchmarking would be required as an additional step for each new software under test before performing the actual network emulation experiments.

Netbed [HRS+04, HRS+08] is an emulation testbed, that supports distributed scalable emulation of wired networks by employing protocol stack virtualization [GSHL03] which is comparable to our node virtualization approach. A daemon process in user space monitors the consumption of the following resource classes: CPU, interrupt load, disk, network, and memory. Monitoring is implemented by sampling existing kernel reports, that are provided through the proc virtual file system with Unix-like operating systems. On the one hand, it is difficult to observe short peaks in resource usage and thus potential contention with long sample intervals. On the other hand, shorter sample intervals imply significant monitoring overhead especially since each sample involves an expensive context switch between user space and kernel space. Increased monitoring overhead contradicts our paramount goal scalability.

4.2.3 Code Instrumentation

As we will see later in Sections 4.4.3 and 4.5.1, quality criteria are best observed at certain places inside the operating system. Therefore, it stands to reason to monitor by means of an event driven approach. This can be implemented by

instrumenting the respective code parts. Code instrumentation approaches can be classified along various dimensions such as: when to instrument (in source code: static; or in object code: static, binary translation, dynamic), what to instrument (address tracing, block counting, execution time, watching access, ..., arbitrary code), how to instrument (inline, separate). Since many approaches overlap on some dimensions, we discuss them in order of publication date.

Simple address tracing, block counting, or measurement of block execution times as is often used in profiling is not sufficient since our quality criteria need access to variables that are local to functions. Therefore, we need fine-grained instrumentation of the operating system kernel.

This can be achieved by manually instrumenting the source code and building it, which requires compilation and linking. Since this has to be done only once for an operating system used on the emulation testbed computers, it is not much of a problem.

Static binary rewriting approaches such as ATOM [SE94] or EEL [LS95] help circumventing the build process by providing abstractions to modify object code directly. They also allow the user to specify the instrumentation code separately from the original code to be instrumented.

Fine-grained dynamic kernel instrumentation is for instance described by KernInst [TM99] of the Paradyn project. The approach allows the separate specification of instrumentation code. Instrumentation can be dynamically inserted ("spliced") into running operating system code by automatically executing the following two steps: (1) In free memory, save original first machine code instruction following the insertion point and append code to insert as well as a jump back to the instruction following the original first instruction; (2) atomically overwrite original first instruction with jump or trap to saved first instruction.

Since the content of instrumentation code is completely up to the user with the above approaches, safety cannot be guaranteed, i.e. interference with or breaking the running OS cannot be prevented. Consequently, approaches have been developed that restrict the power of instrumentation code, e.g. by using script languages and interpreting them in a safe environment whenever such code is hit during execution of an OS. As a further development of Dynamic Trace under the IBM OS/2 operating system, Dynamic Probes (DProbes) [Moo01] supports a script language in reverse Polish notation (RPN) and provides a stack-based interpreter. DTrace [CSL04] also provides fine-grained dynamic instrumentation with guaranteed safety. It is more powerful and e.g. supports thread-local variables, distinguishes real parallel execution of instrumentation code, and provides data aggregation.

Interpreted instrumentation code might have undesired monitoring overhead. Minimal overhead is especially important for the purpose of continuously moni-

toring quality criteria in virtualized network emulation. More recent fine-grained dynamic instrumentation approaches with safety do not interpret instrumented code but rather compile it into native binary code, which can be executed at full system speed. SystemTap [PCE+05] builds on the dynamic instrumentation mechanism of Kprobes (Kernel Dynamic Probes), which is in turn a descendant of DProbes. It provides a script language, programs of which are compiled into kernel modules. To enable the instrumentation, a kernel module is loaded and the probes are dynamically installed. Trace output can be provided by means of efficient kernel to user space communication mechanisms. After enough traces have been collected, SystemTap can again dynamically remove the probes and finally unload the kernel module.

While dynamic instrumentation provides a mechanism for monitoring, the actual instrumentation code has to be created manually based on quality criteria for detecting resource contention.

4.3 Requirements for Detection of Resource Contention

Using quality criteria, a monitoring algorithm must be able to detect resource contention during an emulation experiment. Detection should happen *fast* within a split second—ideally within tens of milliseconds—after the resource contention actually happened. This allows to abort an emulation experiment as early as possible and avoids wasting time continuing an experiment with probably unrealistic measurement results. Since it is necessary to monitor resources in order to detect contention, monitoring should be possible with *minimal overhead*. In case of a detected resource contention, it should be possible to retrieve each vnode's *share of resource consumption*. This information is required to aid an automatic directed adaptation of the mapping between vnodes and pnodes for another emulation run avoiding the detected contention.

4.4 Quality Criteria

In this section, we discuss different possible approaches to develop quality criteria, that are suitable for detection of resource contention by means of resource monitoring. The most promising approach turns out to be empirical. Therefore, we describe it in more detail, before we conclude the section with a definition of the chosen criteria.

4.4.1 Approaches to Definition

We classify approaches to defining suitable quality criteria into three types: mathematical, systematic, and empirical.

A mathematical approach to defining quality criteria would be to model the relevant parts of the existing system. However, an emulation system typically contains a mature production operating system making an accurate mathematical model hardly feasible.

Instead, we could tackle the problem by systematically understanding the system and its behavior on the basis of its implementation, e.g. using the source code. Again, the system itself is very complex and important aspects may be overlooked. For example, we might assume, that a resource contention on the network link to the emulation switch would lead to the device transmission ring filling up (tx at NIC in Fig. 4.1), since the arrival rate of frames would be larger than the queue's service rate. However, this assumption is false, if there is a bottleneck even before the physical network link namely on the data path from main memory to the network interface. This bottleneck artificially limits the maximum throughput and causes the device transmission ring to never fill significantly. We could confirm such behavior for Gigabit Ethernet cards attached to a 32 bit, 33 MHz PCI bus, which can hardly transport 1 GBit/s in a best case without overhead, in a diploma thesis [Hoh05].

Finally, we propose an empirical approach. The process of developing suitable quality criteria is based on a manual feedback loop for each different criterion. In the loop, the actual dynamic system behavior with respect to resource consumption is observed for a typical network emulation scenario. Based on the observation, a new criterion is proposed in the first iteration or the existing criterion is adapted in subsequent iterations. The suitability of the new criterion is validated in the next loop iteration. If validation was successful, the loop terminates and the next

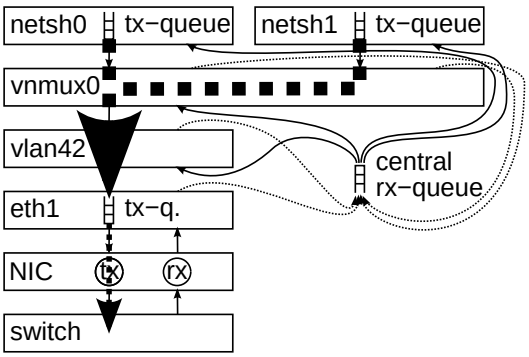

Figure 4.1: Data path of a protocol stack with virtual routing.

criterion for another resource class is developed in the same way. An optimization of the process is to execute the typical emulation scenario only once in advance and record traces of the resource consumption progress during the experiment. Those traces can then be reused during each loop iteration to propose a new criterion or to refine an existing criterion offline without having to rerun an experiment in each iteration.

4.4.2 Empirical Approach

As described in the previous section, we adopt an empirical approach for developing quality criteria. Please note, that we discuss the process of developing criteria in this section. To this end, we record traces of resource events in the operating system kernel with minimal perturbation of the system, in order to develop, refine, and validate quality criteria offline, i.e. independent of a running emulation experiment. This is different from the actual resource monitoring described later in Section 4.5 which uses the criteria to be developed in Section 4.4.3.

A straightforward way to observe dynamic system behavior would be to make use of already existing mechanisms for system accounting and reporting. However, the provided data is often sampled by means of the periodical timer interrupt. For CPU accounting, the current execution context is considered on taking each sample. In case of process context, the entire CPU time during the past timer interval is assumed to having been consumed by the current process. In case of non-process context, such as interrupt service routines or asynchronous operating system kernel tasks, the entire CPU time during the past timer interval is assumed to having been consumed by the "system," i.e. kernel. This may cause erroneous statistics. For instance, the compute time of a process won't appear in the accounting data, if the process happens to yield the CPU each time before the next sample is taken. [LSW03]. In that case, the past timer interval is erroneously added to the account of another process or the system.

Additionally, the provided interface for accessing the data requires periodical readings in order to record traces of resource events. We would have to access the data many times a second. This would cause significant overhead, since each access usually implies one or more context switches.

In order to record resource event traces with minimal system perturbation, we instrument the system directly at those places, where the resource classes network, CPU, memory, and disk are managed by the operating system. This provides time-stamped low level events about the resource usage. Recorded events are the union of resource events caused by all vnodes on the one pnode, where the recording is performed. Dynamic code instrumentation with compiled trace code such as the SystemTap [PCE+05] approach (Section 4.2.3) would be a suitable mechanism to observe dynamic system behavior. However, our prototype based on Linux kernel

version 2.4.24 (Section 3.6.2) does not support SystemTap. For that reason, we manually instrument the kernel source code once. For our purposes, this provides the same functionality and behavior as using dynamic instrumentation.

For time-stamping we use the time stamp counter (TSC) register of the CPU which is incremented at clock cycle frequency. On our emulation testbed computers with a CPU clock frequency of 2.4 GHz this translates into a time stamp granularity of ≈ 0.417 ns (Section 3.7). In order to keep logging overhead low, we allocate areas of the main memory where physical pages are already mapped and pinned to prevent paging. On starting observation, events are then logged into these memory areas in binary form to avoid conversion overhead. We have found a few megabytes of main memory to be sufficient to obtain meaningful event traces. In order to prevent influencing the system with disk I/O for storing traces, the binary logs are copied to disk after the observation period. The saved traces allow flexible offline analysis without influencing the system behavior.

4.4.3 Definition

Our aim is to detect resource contention that only occurs due to node virtualization. We base our quality criteria for contention on detecting whether a resource is utilized to the full extent. Therefore, one instance of software under test must not fully utilize any resource on a pnode without virtualization. Only on a virtualized pnode, the aggregate resource consumption of all the vnodes hosted on that pnode may lead to contention. Without this assumption, no conclusion on resource contention caused by virtualization would be possible, since resources would already be fully occupied without any virtualization.

From the perspective of network emulation, the resource class “network” is the most important. Since we limit consumption of the network resource by means of network emulation for each network interface in a vnode, as an exception, software under test may utilize its granted share of the network resource to the full extent. It is the aggregate utilization of the pnode’s network resource by the hosted vnodes that can cause contention and has to be detected. For the other resource classes—namely CPU, memory, and disk—there is no enforced limitation of consumption for each instance of software under test. Therefore, each instance must adhere to the above rule and not fully utilize any of these resource classes when being executed on a pnode without virtualization.

In the remainder of this section, we define the chosen quality criteria for each of the resource classes network, CPU, memory, and disk. Additionally, we allow optional user-provided application level metrics.

4.4.3.1 Network

We have to distinguish internal and external network traffic. As mentioned in Section 4.4.1, external traffic might be limited by a system inherent bottleneck between memory and NIC. In contrast, internal traffic is limited by the available CPU resources since the virtual software switch (Section 3.6.1) uses the CPU to forward frames. In turn, the vnodes on a pnode compete for its shared CPU resource.

Traffic can be measured in three different units: Frame rate, delay, and throughput.

The maximum frame rate is observed for minimum sized frames. In our experiments, the frame rate remains constant for both external an internal traffic load even under resource contention. Thus, the frame rate is not suitable as indicator for resource contention.

The frame delay is caused by the scheduler serializing the shared CPU resource for different vnodes on the same pnode. As discussed in Section 3.5.2.1, the scheduling granularity of our virtual routing approach is by interrupts in the OS and by process. Therefore, the delay varies between software under test within the OS kernel, i.e. network and transport layer, and software under test within user space, i.e. application layer. On network and transport layer, ingress traffic is processed in the asynchronous receive software interrupt as without any virtualization (see also Section 2.2.2.4). If the rate of ingress frames is higher than what can be processed within the quota for one receive software interrupt execution, additional delay for ingress frames occurs. The next software interrupt execution is after one CPU quantum at the latest and thus typically within 1 or 10 ms. The amount of additional delay only depends on the frame rate, but not directly on the number of vnodes as with virtual machine approaches. Transmission from within the network and transport layer happens promptly, since these are triggered by interrupts in the OS which do not have to wait until the respective vnode is scheduled. Transmission can be triggered by the reception of a frame in the receive software interrupt, e.g. sending the reply to an ARP request. Alternatively, protocol-specific timers, such as for TCP retransmissions, allow prompt transmission. On application layer, there is receive or transmit delay until this vnode gets scheduled [AC06]. In contrast to virtual machine based approaches, we do not have any higher-level CPU scheduler, that performs world switches [SVL01] between vnodes causing additional delay. Yet, the more vnodes per pnode and the more load on those vnodes, the less concurrency exists between the vnodes. In contrast to true parallelism in a distributed system of pnodes, virtualization inevitably introduces pseudo-parallelism similar to multiprogramming. This may influence distributed applications that rely on synchrony. For instance, the convergence time of routing protocols within autonomous systems increases to a lower bound

dependent on number of vnodes per pnode and load [AC06]. Group communication perceives increased inter-receiver delay jitter between vnodes on the same pnode. Since our virtual routing approach implies minimal scheduling overhead and the transmission delays are unavoidable in virtualized environments, we do not consider such delays here. It is easy for an experimenter to limit the worst case transmission delay by limiting the number of vnodes per pnode.

Maximum throughput is achieved for frames having the maximum size limited by the maximum transfer unit (MTU). In our experiments, we find a capping throughput to indicate network resource contention. We focus on network throughput in the following.

Receiving systems do not have any direct influence on their ingress traffic. If multiple pnodes send traffic to one single pnode, contention may arise at the receiving pnode or the emulation hardware switch. This is by no means specific to node virtualization and can occur on any unvirtualized pnode. Therefore, we do not consider network throughput of ingress traffic as quality criterion. Regarding the emulation hardware switch, we employ the same assumption as for emulation without node virtualization: An emulation scenario is mapped to the testbed hardware such that, for each pnode, the aggregated throughput at any time during a scenario or the sum of bit rates on the emulated links of vnodes on a pnode does not exceed the bit rate of the underlying physical network link, i.e. 1 GBit/s for our hardware. If such a mapping does not exist, the specific scenario cannot be emulated. This assumption only applies to inter-pnode traffic. Intra-pnode traffic between local vnodes handled by the software switch depends on the CPU resource. For such local traffic, the aggregate throughput of egress traffic with internal destination suffices as quality criterion since all senders and receivers share the same CPU resource.

Within the source code of our virtual software switch and thus on pnode scope, we instrument network events for egress traffic, i.e. traffic leaving vnodes. There are two types of egress frame events: *e*xternal and *i*nternal. Frames with external destination MAC address on another pnode are sent via the NIC over the physical uplink to the emulation switch. We denote such an event as $e_{n_e} = (t, l)$ with t being the event time stamp and l being the frame length in bytes. Frames with an internal destination MAC address on another vnode on the same pnode are delivered by the virtual software switch. We denote such an event as $e_{n_i} = (t, l)$. In order to reference the elements of a particular tuple we use a dotted notation similar to accessing fields of a record type in programming languages. Hence, $e_{n_e}.t$ and $e_{n_e}.l$ denote the time stamp and the frame length of the event e_{n_e} respectively. Similarly, this applies to internal events by substituting the index n_e with n_i.

Fig. 4.2 shows the data of an e_{n_e} trace, where load is generated with maximum sized frames of 1500 bytes. The trace contains the union of events triggered by

all vnodes on the one pnode the trace is recorded on. In this example, each of five vnodes executes one load generator sending frames as fast as possible. The throughput of each vnode is limited by an emulated bandwidth with a fraction of the pnode's physical network link speed. Each egress frame event with external destination is depicted by an impulse bar at a certain point in time defined by the event time stamp t relative to experiment begin. The impulse height represents the size l of this particular frame.

The average external network throughput on a pnode within a time interval is calculated by summing the frame lengths of all frame events during the interval and dividing the sum by the interval duration δ_{n_e}. We compare the throughput to a threshold θ_{n_e}. The threshold defines the lowest throughput indicating resource contention. Equation 4.1 evaluates to true, if resource contention happened in the time interval up to and including the current time t.

$$1/\delta_{n_e} \cdot \sum_{e \in \{e_{n_e} | t - \delta_{n_e} < e_{n_e}.t \leq t\}} e.l \geq \theta_{n_e} \tag{4.1}$$

The threshold has to be calibrated for the emulation testbed hardware. We show θ_{n_e} for our emulation system in Section 4.6.

Without loss of generality, we assume one physical uplink to the emulation switch. An extension to multiple uplinks per emulation testbed computer is straightforward. Each uplink would have to be monitored separately. As soon

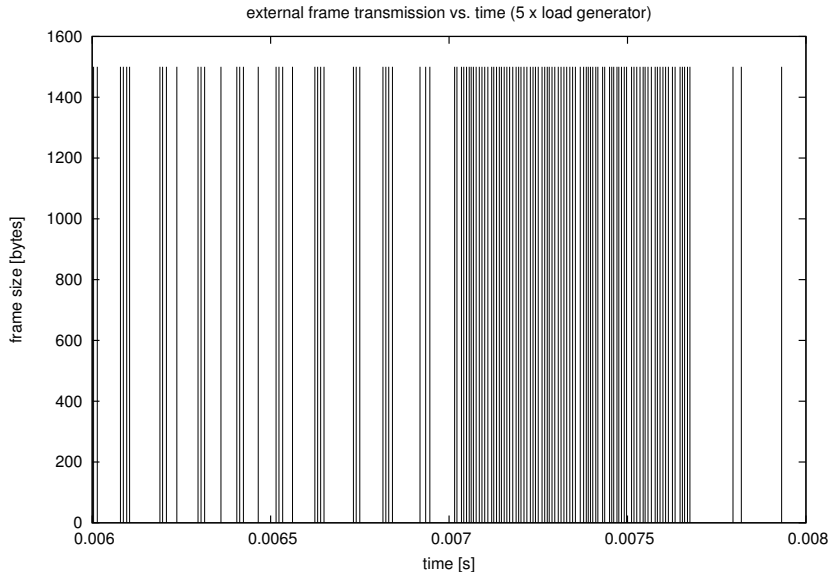

Figure 4.2: Example of external network transmission events on a pnode.

as Equation 4.1 evaluates to true for one or more uplinks, resource contention occurred.

The above said likewise applies to egress frame events with internal destination by substituting the index n_e with n_i.

4.4.3.2 CPU

CPU resource contention occurs if and only if all CPU cycles have been scheduled to runnable processes during a certain time interval. During time frames without runnable processes, the process scheduler switches to a pseudo process typically referred to as "idle." This process usually does not compute anything but rather puts the CPU into halt or a power safe mode from which the CPU only wakes up on receiving a hardware interrupt. It stands to reason to instrument the idle process for monitoring CPU resource contention [Lep05].

Within the source code of the process scheduler and thus on pnode scope, we instrument the event of the idle process being scheduled. This CPU event $e_c = (t_s, t_e)$ includes the time stamp t_s when the scheduler switches from a runnable process to the idle process and the time stamp t_e when the scheduler switches from the idle process to a runnable process. Hence, the CPU is idle during the interval $[t_s, t_e)$.

Fig. 4.3 shows a step function over time of an e_c trace. The trace contains the union of events triggered by all vnodes on the one pnode the trace is recorded

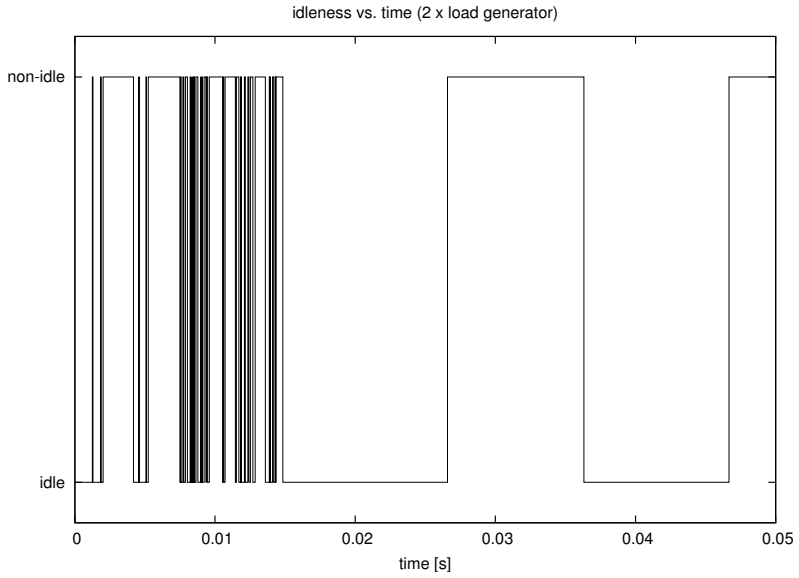

Figure 4.3: Example of CPU events on a pnode.

on. In this example, each of two vnodes executes one load generator consuming a fraction of the processor time. If the function value is at the bottom of the y-axis, the idle process is scheduled. If the function value is at the top of the y-axis, a runnable process is scheduled and the CPU actually computes. A time interval during which the function value remains at the top (non-idle), indicates a fully loaded CPU during this interval.

We can calculate the interval duration of a fully loaded CPU. We store CPU events e_c in a one-dimensional array chronologically indexed by an integer i. For the example, we assume index i points to the current event. In order to reference a particular tuple we use a notation appending the array index in squared brackets as for accessing array elements in programming languages: $e_c[i]$. The elements of a particular tuple are referenced by using a dotted notation similar to accessing fields of a record type in programming languages. Hence, $e_c[i].t_s$ and $e_c[i].t_e$ denote the time stamps of the event e_c at array index i. The duration of a continuously busy CPU is the time stamp of the end of the previous event subtracted from the time stamp of the beginning of the current event. This corresponds to a rising edge followed by a falling edge in Fig. 4.3. We compare the non-idle interval duration to a threshold θ_c. Equation 4.2 evaluates to true, if the CPU has been fully loaded for at least the time interval θ_c.

$$e_c[i].t_s - e_c[i-1].t_e \geq \theta_c \tag{4.2}$$

The choice of the threshold for a certain pnode depends on the total number of software under test processes over all vnodes on that pnode. If all such processes are compute bound, each of them fully consume its CPU quantum until a preemptive scheduler takes CPU away to schedule another runnable process. This situation corresponds to our definition of CPU resource contention. Hence, the threshold θ_c should be chosen as the CPU quantum multiplied by the total number of software under test processes over all vnodes on that pnode. On Linux, the quantum typically lies between 10 ms and 1 ms which is the reciprocal value of the timer interrupt frequency of 100 Hz or 1 kHz respectively.

Without loss of generality, we assume a single CPU system. An extension to symmetric multiprocessor systems is straightforward. Each CPU would have to be monitored separately. As soon as Equation 4.2 evaluates to true for one or more CPUs, resource contention occurred.

As an alternative criterion, similar to the network criterion in Section 4.4.3.1, it would be possible to compare the average CPU utilization in a time interval δ against a threshold θ. This would require the summation of all sub-intervals, during which non-idle processes are scheduled, over the time interval δ. We detect contention if the average CPU utilization is close or equal to 1. This only happens, if mostly or only non-idle processes are scheduled within δ. In case of

only non-idle processes, we get $1/\delta \cdot \sum = 1/\delta \cdot \delta = 1 \geq \theta$ which is $\delta \geq \theta \cdot \delta$ (with $\theta = 1$). Then, $e_c[i].t_s - e_c[i-1].t_e \geq \delta$ holds true for Equation 4.2 and we can choose a $\theta_c \geq \delta$ as described above. Therefore, calculating the average CPU utilization is not necessary and a comparison of one difference against the time interval θ_c is sufficient. The less complex criterion also requires less instructions in the monitoring algorithm (Section 4.5.1.2) and thus minimizes monitoring overhead.

4.4.3.3 Memory

In contrast to network and CPU, the utilization of memory resources is difficult to measure. Mechanisms to share data between processes such as shared library code, shared memory, or buffer and page cache prevent unique accounting of resource consumption to processes. Buffer and page cache typically use all remaining free memory to speed up I/O and also dynamically adapt their size to make memory available again if needed. This renders the measurement of free available memory infeasible. Virtual memory enables over-commitment by paging to disk storage. Since page faults make memory access slower by orders of magnitude, we consider the page fault rate as quality criterion for memory resources.

Within the source code of the page fault handler and thus on pnode scope, we instrument memory events for page-in $e_{m_i} = (t)$ and for page-out $e_{m_o} = (t)$. Each such event holds a time stamp t of when it happened. In order to reference the element of a particular tuple we use a dotted notation similar to accessing fields of a record type in programming languages. Hence, $e_{m_i}.t$ denotes the time stamp of the event e_{m_i}. Similarly, this applies to page-out events by substituting the index m_i with m_o.

Fig. 4.4 shows the data of an e_{m_o} trace, where load is generated by writing into large memory regions. The trace contains the union of events triggered by all vnodes on the one pnode the trace is recorded on. In this example, each of nine vnodes executes one load generator continuously writing into a memory area of size 60 MB. Each page-out event is depicted by an impulse.

We can calculate the average page fault rate for page-out events within a time interval δ_{m_o}. The number of all such events during the interval is represented by the cardinality of the set of all events that happened during the interval up to and including the current time t. Dividing the set's cardinality by the interval duration results in the page fault rate. We compare the rate to a threshold θ_{m_o}. Equation 4.3 evaluates to true, if resource contention occurred in that interval.

$$1/\delta_{m_o} \cdot |\{e_{m_o} | t - \delta_{m_o} < e_{m_o}.t \leq t\}| \geq \theta_{m_o} \tag{4.3}$$

Depending on the experimenter's expectations regarding the occurrence of paging, the threshold can be chosen. The most cautious setting is a paging rate of

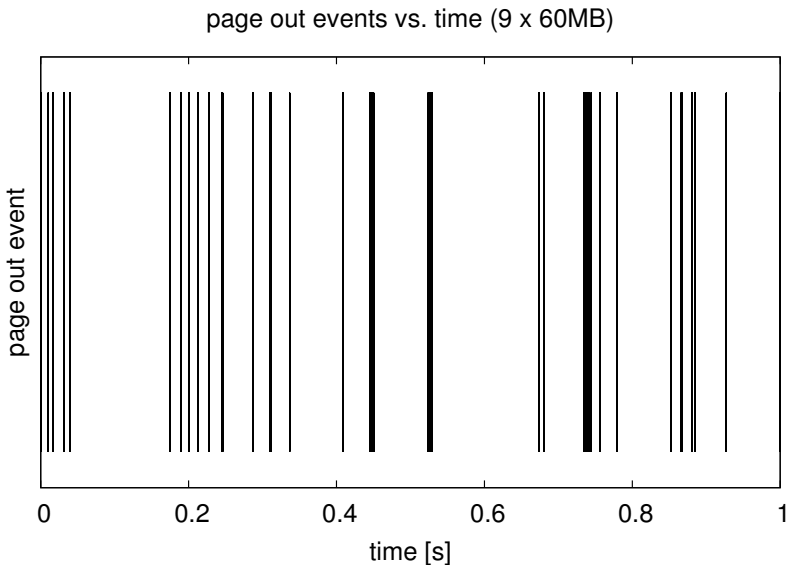

Figure 4.4: Example of page-out events on a pnode.

zero or very close to zero to only accept occasional paging, which might occur due to page replacement algorithms without actual resource contention.

The above said likewise applies to the page fault rate for page-in events by substituting the index m_o with m_i. Page-in events only represent page faults, that lead to reading data back in from the paging devices which has previously been paged out. This does not include demand paging where data is read directly from an object file on disk, for instance for the code or data segment. We consider demand paging a usual and unavoidable operating system mechanism since all binaries are loaded this way. Therefore, it is not part of the memory quality criteria.

4.4.3.4 Disk

A disk device is busy as long as it has at least one I/O request to complete. An operating system typically manages one request queue per device. The queue length represents the number of I/O requests, that are pending on a device. Requests, that have not been submitted to the device itself yet, can be merged with new requests being placed into the same queue. Otherwise new requests increase the queue length. Requests, that have been finished by the device, are removed from the queue and thus decrease the queue length. During times, when there is no entry in a device queue, the device is idle.

Within the source code of the block device subsystem and thus on pnode scope, we instrument device queue operations, that modify the number of queue entries.

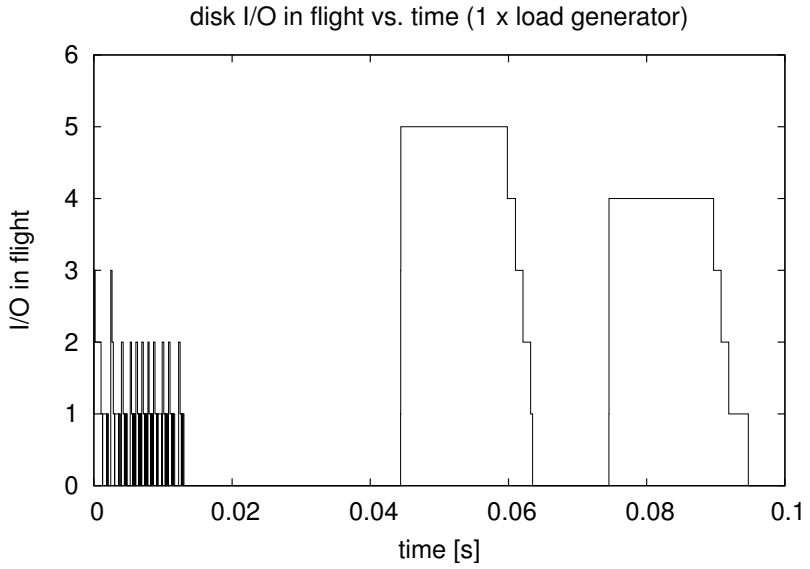

Figure 4.5: Example of disk-I/O events on a pnode.

For each such operation, we record a disk event $e_d = (t, r)$ with t being the time at which a number of r I/O requests are pending in the queue.

Fig. 4.5 shows a step function over time of an e_d trace. The trace contains the union of events triggered by all vnodes on the one pnode the trace is recorded on. In this example, one vnode executes one load generator utilizing the disk during parts of the time. At any time on the x-axis, the function value represents the number of pending I/O requests. For a function value of zero, the disk is idle. Otherwise, the disk is busy.

We can calculate the time interval during which a disk is continuously busy. A one-dimensional array chronologically stores disk events indexed by integers i, j, k. In order to reference a particular tuple we use a notation appending the array index in squared brackets as for accessing array elements in programming languages: $e_d[i]$. The elements of a particular tuple are referenced by using a dotted notation similar to accessing fields of a record type in programming languages. Hence, $e_d[i].t$ and $e_d[i].r$ denote the time stamp and the number of pending requests of the event e_d at array index i. The interval during which a disk is continuously busy starts with the event at which the previously empty device queue fills with one entry. The next event at which the device queue becomes empty again represents the end of the interval. We compare the interval duration to a threshold θ_d. Equation 4.4 evaluates to true, if the disk has been constantly busy for at least the time interval θ_d.

$$e_d[k].t - e_d[i+1].t \geq \theta_d \tag{4.4}$$
$$\text{with } \forall j,\ i < j < k : e_d[i].r = 0,\ e_d[j].r > 0,\ e_d[k].r = 0$$

The choice of the threshold depends on the ratio of arrival rate of new I/O requests and the service rate, i.e. the speed, of a disk device. We show an example θ_d for our emulation system in Section 4.6.

Without loss of generality, we assume one disk in a system. An extension to multiple disks per system is straightforward. Each disk would have to be monitored separately. As soon as Equation 4.4 evaluates to true for one or more disks, resource contention occurred.

As an alternative criterion, similar to the network criterion in Section 4.4.3.1, it would be possible to compare the average disk utilization in a time interval δ against a threshold θ. As with the CPU criterion in Section 4.4.3.2, this would require the summation of all sub-intervals, during which continuously > 0 requests are pending, over the time interval δ. We detect contention if the average disk utilization is close or equal to 1. This only happens, if most or all of the time > 0 requests are pending within δ. In case of continuously > 0 requests over the entire interval, we get $1/\delta \cdot \sum = 1/\delta \cdot \delta = 1 \geq \theta$ which is $\delta \geq \theta \cdot \delta$ (with $\theta = 1$). Then, $e_d[k].t - e_d[i+1].t \geq \delta$ holds true for Equation 4.4 and we can choose a $\theta_d \geq \delta$. Therefore, calculating the average disk utilization is not necessary and the comparison of one difference against the time interval θ_d as described above is sufficient. The less complex criterion also requires less instructions in the monitoring algorithm (Section 4.5.1.4) and thus minimizes monitoring overhead.

4.4.3.5 Application Level Metrics

In case an experimenter has special needs for quality criteria, e.g. with respect to the timing of application specific tasks, that cannot be captured with the previously defined criteria, he may introduce his own. Since these application specific quality criteria may be arbitrary, we can only provide an interface for reporting detected resource contentions.

If, for instance, an experimenter needs to investigate performance metrics of group communication, which relies on a low inter-receiver delay jitter as discussed in Section 4.4.3.1, an additional application level criterion might be useful. Each instance of the software under test on a pnode can obtain a time stamp for each multicast packet it gets delivered. Since those instances run on the same pnode, they use the same clock and the time stamps can be perfectly compared. As processes on the pnode, the instances can communicate efficiently by means of a suitable existing inter process communication mechanism in order to exchange

packet time stamps. To keep exchange overhead low, it stands to reason to collect time stamps for a number of packets and exchange batches of time stamps instead of separately for each packet. Assuming each multicast packet contains a sequence number, one instance per pnode can associate time stamps of different vnodes with the respective packet and thus keep track of the earliest and the latest delivery time stamp for each multicast packet. The inter-receiver delay jitter for a certain packet can be calculated by subtracting the time stamp of its earliest delivery from the time stamp of its latest delivery. Memory is only required to store two time stamps for each packet, that has not yet been delivered to all vnodes on the pnode. Since the jitter is bounded by the number of vnodes, which is in turn finite, the space requirement is limited. Depending on the scenario, a certain threshold θ_j represents the upper limit for the jitter. If the threshold is exceeded for any of the packets, rate limited reporting to the system log should be performed as will be described in Section 4.5.2.

4.5 Monitoring Approach

With the defined quality criteria from Section 4.4.3, we design a monitoring instrumentation with minimal overhead. The main part of our monitoring approach consists of instrumenting basic resource scheduling events in the operating system kernel. On detecting a resource contention, the instrumentation code triggers a user notification, which we briefly discuss at the end of this section.

4.5.1 Instrumentation of Basic Events

The mechanism to instrument the operating system kernel for monitoring is similar to the mechanism used for recording resource scheduling event traces in Section 4.4.2. However, instead of recording each event and analyzing the trace offline, monitoring of resource usage and evaluation of quality criteria works self-contained in the instrumentation code and online during emulation experiments. As shown in Section 4.4.3, quality criteria for the four resource classes differ slightly for reasons of monitoring efficiency and can be classified into two groups. Criteria for CPU and disk compare a time difference between two events with a threshold. Criteria for network and memory compare an average resource consumption over a time interval with a threshold.

Fig. 4.6 shows a timeline with either CPU or disk events marked as arrows pointing perpendicularly onto the timeline. Each such event could be the scheduling between an idle and non-idle process, or the disk request queue entering or leaving the state of being empty. The instrumentation code initializes variables with an event of scheduling from the idle to a non-idle process or with an event of the

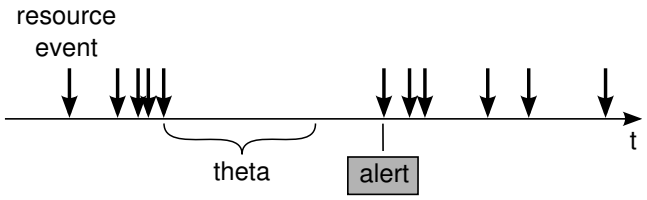

Figure 4.6: Event based approach for monitoring CPU and disk.

disk request queue leaving the state of being empty. On each next complement scheduling or disk request event, the monitoring code compares the elapsed time interval since the last initialization with a threshold θ specific to each resource class. If the interval exceeds the threshold, resource contention occurred and user notification is triggered.

Fig. 4.7 shows a timeline with either network or memory events marked as arrows pointing perpendicularly onto the timeline. Each such event could be the transmission of a frame or swapping of a page. The timeline is equally divided into epochs, which have the duration of the time interval δ specific to each resource type. On being triggered for the first time by a resource event, the instrumentation code starts the first epoch. This includes the initialization of variables. On each occurrence of an event during the current epoch, metrics of the resource quality criterion are aggregated. Whenever the time of an event lies beyond the current epoch, a new epoch starts just like for the very first epoch. If the metrics aggregation within the current epoch exceeds the resource threshold θ, a resource contention occurred and user notification is triggered.

To keep computation time for the aggregation of quality criteria metrics minimal, we can transform the equations for network and memory from Section 4.4.3. Multiplying those inequalities containing a time interval with δ eliminates an expensive division operation. The value of the resulting multiplication on the right hand side only changes on user configuration. Hence, it is constant during an emulation experiment. Therefore, we can pre-calculate the product Π once on

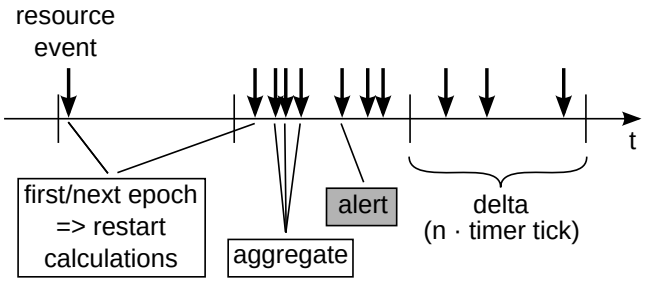

Figure 4.7: Event based approach for monitoring network and memory.

configuration before an experiment. During an experiment, the monitoring code only needs to perform cheap integer additions and comparisons.

Equation 4.1 for the network resource criterion transforms into the following:

$$1/\delta_{n_e} \cdot \sum e.l \geq \theta_{n_e} \Rightarrow \underbrace{\sum e.l}_{\text{add}} \geq \underbrace{\theta_{n_e} \cdot \delta_{n_e}}_{\text{constant}} = \Pi_{n_e} \tag{4.5}$$

Similarly, we transform Equation 4.3 for the memory resource criterion into the following:

$$\begin{array}{rrcl} & 1/\delta_{m_o} \cdot |\{e_{m_o} | t - \delta_{m_o} < e_{m_o}.t \leq t\}| & \geq & \theta_{m_o} \\ \Rightarrow & |\{e_{m_o} | t - \delta_{m_o} < e_{m_o}.t \leq t\}| & \geq & \theta_{m_o} \cdot \delta_{m_o} = \Pi_{m_o} \end{array} \tag{4.6}$$

Both variants of event-based monitoring described above have the advantage that they do not introduce any additional context switches or interrupts. Unlike sampling, the approach does not miss any event and accounts even short peaks in resource usage.

Other than the event recording described in Section 4.4.2, we do not need the the very fine granular CPU time stamp counter for monitoring here. As we will show in Section 4.6.2.1, the threshold θ for CPU and disk as well as the time interval δ for network and memory can be chosen greater than or equal to a few milliseconds. Therefore, we base our monitoring on the traditional kernel variable, which is increased on every timer interrupt. In Linux this variable is called “jiffies” and contains the number of timer interrupts since system boot. By making use of this already existing variable, that can be accessed efficiently, we can keep monitoring overhead minimal.

The above-mentioned properties of the monitoring approach—event-based, simple integer arithmetic, using existing time variable—ensure minimal monitoring overhead.

In the following, we discuss monitoring algorithms in pseudo-code for each of the resource classes network, CPU, memory, and disk. Their real implementations all have default values for the configuration variable θ as well as for δ with network and memory resources. This may lead to early notifications about resource contentions during booting the operating system. However, those notifications can be safely ignored. Before starting actual emulation experiments, the user configures the variables and resets any previous notifications. User configuration works by means of a user space to kernel space interface. On Linux, we use the virtual system file system (sysfs).

4.5.1.1 Network

Algorithm 4.1 shows the pseudo code matching our implementation for monitoring network resource quality criteria. It begins with a set of preconditions, that need to be fulfilled for the algorithm to work properly. Code of the parameter configuration interface ensures that each θ_n is greater than zero and it calculates product values for Π_n. Variables for the epoch beginning t_n are initialized to zero. Booting the OS takes at least as long as δ_n, so a new epoch is started before emulation experiments run. This is important for the correct initialization of the remaining variables, which we describe below.

The procedure VNMUXTRANSMIT is called within a virtual software switch instance (Section 3.6.1) to transmit frames originating from any vnode, which is local to the pnode running the switch instance. Based on the destination MAC address, it either delivers the frame to a local vnode or transmits the frame on the physical uplink to the emulation switch. For the purpose of readability, we left the handling of broadcast and multicast frames out of the pseudo code. The real implementation transmits frames with broadcast or multicast MAC address on the physical uplink and also delivers to all local vnodes but the one where the frame originated.

Between the switching decision and the further frame handling, the procedure NETWORKEVENT is called. It contains the actual instrumentation code. We distinguish network events for frames with local destination and non-local destination, since the averaging interval δ_n and especially the threshold θ_n might need to be different. Other than that, both cases contain the same calculations with only the variable index being different. By comparing the last epoch time stamp t_n plus the epoch length δ_n with the current time, we can determine the beginning of a new epoch.

In that case, the new epoch starts at the current time and we remember the current event in terms of the frame length. In order to provide each vnode's share of resource consumption (see Section 4.3) on occurrence of a contention, we aggregate the criterion metrics also for each vnode identifier j. On starting a new epoch, all those aggregates in the array v_n are reset.

If the current event falls into the current epoch, the metric frame length is aggregated into l_n and also into the array field of the current vnode. A resource contention occurred, if the sum of frame lengths l_n in the current epoch exceeds the product of epoch duration δ_n and threshold θ_n. In that case, user notification is triggered. It is possible that more events will follow in the same epoch and thus trigger notification multiple times until a new epoch will begin. A maximum number of notifications can be configured by the user. This way, the notification code prevents a flood of notifications.

Require: $\Pi_{n_i} = \theta_{n_i} \cdot \delta_{n_i} \wedge t_{n_i} + \delta_{n_i} \leq \text{currentTime} \wedge \theta_{n_i} > 0 \wedge$
$\Pi_{n_e} = \theta_{n_e} \cdot \delta_{n_e} \wedge t_{n_e} + \delta_{n_e} \leq \text{currentTime} \wedge \theta_{n_e} > 0$

```
    procedure VNMUXTRANSMIT(frame)
        destination ← LOOKUPSWITCHINGTABLE(destinationMacAddress)
        NETWORKEVENT(frame, destination)    // instrumentation
 5:     if ISLOCAL(destination) then
            LOCALDELIVER(frame)    // destination is a local vnode
        else
            DOTRANSMIT(frame)    // destination is on other pnode
        end if
10: end procedure

    procedure NETWORKEVENT(frame, destination)
        if ISLOCAL(destination) then
            if t_{n_i} + δ_{n_i} ≤ currentTime then    // new epoch
15:             t_{n_i} ← currentTime
                l_{n_i} ← GETLENGTH(frame)
                ∀j ∈ Vnodes \ currentVnode : v_{n_i}[j] ← 0    // vnodes' fractions
                v_{n_i}[currentVnode] ← GETLENGTH(frame)
            else    // still same epoch
20:             l_{n_i} ← l_{n_i} + GETLENGTH(frame)
                v_{n_i}[currentVnode] ← v_{n_i}[currentVnode] + GETLENGTH(frame)
                if l_{n_i} ≥ Π_{n_i} then
                    NOTIFYUSER    // internal network threshold triggered
                end if
25:         end if
        else    // external destination on other pnode
            ...    // analogous to previous branch by replacing index n_i with n_e
        end if
    end procedure
```

Algorithm 4.1: Monitoring of internal and external network events.

4.5.1.2 CPU

In Algorithm 4.2, we present the pseudo code for monitoring the CPU resource quality criterion. Since there is no rate calculation involved, there is no averaging interval δ. Only the threshold θ_c has to be greater than zero to fulfill the preconditions.

Require: $\theta_c > 0$

```
    procedure PROCESSACCOUNTING(process, userTicks, systemTicks)
        ...
        v_c[currentVnode] ← userTicks + systemTicks
        ...
 5: end procedure

    procedure SCHEDULE
        prev ← GETCURRENTPROCESS
        next ← COMPUTENEXTPROCESS
10:     CPUEVENT(prev, next)    // instrumentation
        SWITCHTOPROCESS(next)
    end procedure

    procedure CPUEVENT(prev, next)
15:     if prev = idleProcess then
            if next ≠ idleProcess then    // transition from idle to busy
                t_c ← currentTime
                ∀j ∈ Vnodes : v_c[j] ← 0    // vnodes' fractions
            end if
20:     else    // prev ≠ idleProcess
            if t_c + θ_c ≤ currentTime then    // threshold triggered
                NOTIFYUSER
            end if
        end if
25: end procedure
```

Algorithm 4.2: Monitoring of CPU events.

The procedure PROCESSACCOUNTING is part of the default process accounting in the operating system kernel. It gets called on each timer interrupt and thus samples CPU usage of processes. The current context is either process (userTicks) or kernel (systemTicks). We add additional accounting per vnode to obtain each vnode's CPU usage share during an epoch in case of resource contention.

Whenever the OS kernel wants to give CPU time to another process, it calls the procedure SCHEDULE. It remembers the current process in the variable prev. The result of the actual scheduling algorithm is the process to switch to and is stored in the variable next. This is all information we need to call our instrumentation procedure. Finally, the process scheduler code finishes by actually switching to the next process.

Since we want to measure the time during which the CPU is continuously busy, we identify the beginning of such an interval by the fact that the scheduler switches from the idle process to a non-idle process. This corresponds to a rising edge in Fig. 4.3. We remember the current time as beginning of the interval and reset the CPU usage shares per vnode. Such an event can be considered the beginning of a new epoch although epochs for CPU monitoring are not equally long since there is no averaging interval δ involved.

If the scheduler switches from a non-idle process to any process, a busy interval lasted to at least the current time. The interval may either continue, if the scheduler switches again to a non-idle process, or it may end, if the scheduler switches to the idle process. However, this distinction is unnecessary here. In any case, a resource contention occurred, if the busy interval exceeds the threshold θ_c.

4.5.1.3 Memory

Algorithm 4.3 shows the pseudo code for monitoring memory resource quality criteria. Preconditions are similar to those of the network quality criteria in Algorithm 4.1. Configuration interface code can ensure that θ_m are greater than zero and it can pre-calculate the products Π_m. Variables for the epoch beginning t_m have static initializers in the implementation code setting them to zero. Again we assume that booting the OS takes longer than the epoch duration δ_m. Therefore, a new epoch is started before emulation experiments run.

On a page fault, that can be handled by reading to or writing from paging space, the OS kernel calls the procedure READWRITESWAPPAGE. Before the actual I/O operations happen, we call our instrumentation procedure. It gets the I/O operation direction as argument, which is either read for paging in or write for paging out.

For the instrumentation, we distinguish page-in and page-out events, in order to provide the user with different averaging interval lengths δ_m and especially different thresholds θ_m. Both cases are handled analogously and only the variable subindex is different. If the current time has passed the end of the previous epoch $t_m + \delta_m$, a new epoch starts. We remember the epoch beginning in t_m and the fact that this one paging event has occurred in n_m. Also, the number of paging events is reset for each vnode but the current one, which is re-initialized to 1 representing this one event in this new epoch.

If the current event happens in the current epoch, we aggregate the number of paging events in n_m and also in the array field of the current vnode. A resource contention occurred, if the number of paging events n_m in the current epoch exceeds the product of epoch duration δ_m and threshold θ_m.

Require: $\Pi_{m_i} = \theta_{m_i} \cdot \delta_{m_i} \wedge t_{m_i} + \delta_{m_i} \leq \text{currentTime} \wedge \theta_{m_i} > 0 \wedge$
$\Pi_{m_o} = \theta_{m_o} \cdot \delta_{m_o} \wedge t_{m_o} + \delta_{m_o} \leq \text{currentTime} \wedge \theta_{m_o} > 0$

procedure READWRITESWAPPAGE(page, direction)
  PAGINGEVENT(direction) // instrumentation
  DOPAGEIO(page, direction)
end procedure

procedure PAGINGEVENT(direction)
  if direction = read **then**
    if $t_{m_i} + \delta_{m_i} \leq \text{currentTime}$ **then** // new epoch
      $t_{m_i} \leftarrow \text{currentTime}$
      $n_{m_i} \leftarrow 1$
      $\forall j \in \text{Vnodes} \setminus \text{currentVnode} : v_{m_i}[j] \leftarrow 0$ // vnodes' fractions
      $v_{m_i}[\text{currentVnode}] \leftarrow 1$
    else // still same epoch
      $n_{m_i} \leftarrow n_{m_i} + 1$
      $v_{m_i}[\text{currentVnode}] \leftarrow v_{m_i}[\text{currentVnode}] + 1$
      if $n_{m_i} \geq \Pi_{m_i}$ **then**
        NOTIFYUSER // page-in threshold triggered
      end if
    end if
  else // direction = write
    ... // analogous to previous branch by replacing index m_i with m_o
  end if
end procedure

Algorithm 4.3: Monitoring of page-in and page-out events.

4.5.1.4 Disk

Algorithm 4.4 contains the pseudo code for monitoring the disk resource quality criterion. As for the CPU monitoring in Algorithm 4.2, there is no averaging interval δ. The only precondition is to have the number of currently pending disk I/O requests r_d initialized to zero on system boot. Existing OS kernel code for disk accounting ensures this initialization.

The low-level block device subsystem of the OS uses two different procedures on enqueuing a new disk I/O request. It calls the procedure IOREQUESTNEW for each new request. This procedure increases the number of pending requests, calls our instrumentation code, and finally appends the new request to the queue. All requests in the device queue but the first one are typically not yet submitted to

Require: r_d originally initialized to 0

```
    procedure IoRequestNew(request)
        ...
        r_d ← r_d + 1    // accounting
        DiskEvent(new)    // instrumentation
 5:     EnqueueRequest(request)
    end procedure

    procedure IoRequestMerge(request)
        ...
10:     MergeRequest(request)
        r_d ← r_d − 1    // accounting
        DiskEvent(merge)    // instrumentation
    end procedure

15: procedure IoRequestFinish(request)
        ...
        FinishRequest(request)
        r_d ← r_d − 1    // accounting
        DiskEvent(finish)    // instrumentation
20: end procedure

    procedure DiskEvent(eventType)
        if eventType = new ∧ r_d = 1 then    // new busy period
            t_d ← currentTime
25:         ∀j ∈ Vnodes \ currentVnode : v_d[j] ← 0    // vnodes' fractions
            v_d[currentVnode] ← 1
        else    // (eventType = new ∧ r_d > 1) ∨ eventType ∈ {merge, finish}
            if eventType = new then
                v_d[currentVnode] ← v_d[currentVnode] + 1
30:         else if eventType = merge then
                v_d[currentVnode] ← v_d[currentVnode] − 1    // merge saves request
            end if
            if ((r_d = 0 ∧ eventType = finish) ∨ r_d ≥ 1) then    // busy until now
                if (t_d + θ_d ≤ currentTime) then    // threshold triggered
35:                 NotifyUser
                end if
            end if
        end if
    end procedure
```

Algorithm 4.4: Monitoring of disk events.

the device driver for actual processing. If there is a request in the queue whose block range is close to the block range of a new request, both requests can be merged into one single request. In such a case, IOREQUESTMERGE handles the request combination, undoes the accounting of IOREQUESTNEW, and calls our instrumentation code.

When the disk device has finished a request, the disk device driver triggers a call of the procedure IOREQUESTFINISH. This procedure handles the cleanup of the finished request from the device queue, decrements the number of pending requests, and calls our instrumentation code.

In order to measure the time during which the disk is continuously busy, we first need to detect the beginning of such a time interval. It begins with the first request to an idle disk, i.e. a new request being the only one in the device queue. We remember the interval beginning in t_d and reset the resource consumption in v_d for each vnode.

All other disk events, that do not indicate the beginning of a busy interval, are handled in the else case. For a new I/O request, we increment the array field for the current vnode to account for the vnode's resource usage share during a busy interval. For a merge of two requests, we undo the per-vnode accounting for the second request.

The disk busy interval may have already ended with a previous event, end with this event, or continue with this event. We simply ignore the first case, because the disk is now idle. If the busy interval ends with this event, it is a finish event and the number of pending requests r_d just dropped to zero. If the busy interval continues, the number of requests r_d is greater than zero. The latter two cases describe the fact that the disk has been continuously busy up to the current point in time. If this busy duration exceeds the threshold, resource contention occurred and user notification is triggered.

4.5.2 User Notification

Our prototype uses simple kernel messages to inform the user of a resource contention when the procedure NOTIFYUSER has been called by the monitoring algorithms in Section 4.5.1. In order to prevent message flooding, the user can define a limit how many contentions are reported for each resource type. This limit number gets decremented with each notification and reporting stops at zero. The user may reset the number to a positive integer at any time. Existing standard logging mechanisms report kernel messages into the system log. There are system log daemon implementations, that allow the execution of arbitrary commands when certain messages get logged. This can be used to abort an emulation run promptly on resource contention.

Alternatively, other existing communication mechanisms between kernel space and user space could be used. Even though our discussion in Section 3.6.4.1 refers to communicating from user space to kernel space, it also applies here where we need communication from kernel space to user space.

4.6 Evaluation

We run all evaluation experiments on one single pnode having exactly the same system configuration as described in Section 3.7. Such an emulation testbed computer is equipped with an Intel Pentium 4 2.4 GHz processor, 512 MB RAM, and a Gigabit Ethernet adapter connected to a 32 bit, 33 MHz PCI bus. The device driver for this Ethernet adapter makes use of interrupt mitigation.

Fig. 4.8 shows the emulation scenario for our evaluations which is similar to Fig. 3.33 in Section 3.7.3.2 (see also Fig. 3.10 in Section 3.6.3). All nodes participate in the same WLAN. Here, the wireless links are also full duplex but have a limited bandwidth of 100 MBit/s to reach the theoretical maximum line speed of 1 GBit/s with at most 10 vnodes. We do not emulate the effects of a MAC layer and here we do not execute AODV-UU or any other dynamic routing protocol since there is no need for multi-hop communication.

Load generators and measurements run only on pnode 1. We generate load for each resource type separately. Several experiments with increasing load allow the controlled overloading of each resource class. A vnode generates a defined load, that only uses a fraction of the available resource. We vary the load by increasing the number of vnodes on the same pnode. Each vnode generates the same load. Therefore, the number of vnodes is a unit-less, discrete measure for the generated load.

In the following subsections, we reuse the event recording from Section 4.4.2 to evaluate the effectiveness of our quality criteria defined in Section 4.4.3. With our implementation of the monitoring algorithms from Section 4.5.1, we then evaluate the effectiveness of monitoring quality criteria during an emulation experiment. Since monitoring should be as lightweight as possible in order not to influence

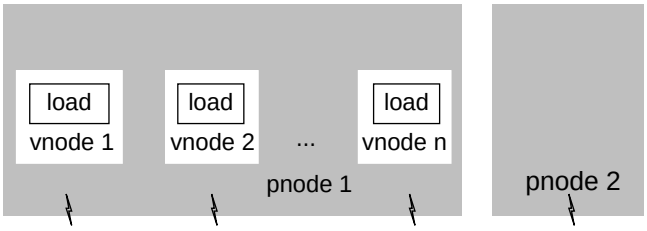

Figure 4.8: Network emulation scenario for evaluations.

emulation measurement results, we close the section with an evaluation of the monitoring overhead.

4.6.1 Effectiveness of Quality Criteria

To validate that our defined quality criteria equations evaluate to true if and only if contention occurred for the respective resource type, we reuse the event recording from Section 4.4.2. The quality criteria are expected to indicate resource contention for only those experiments where we intentionally overload the resource. As in the previous sections, we discuss all four different resource types in the order: network, CPU, memory, and disk.

For evaluating network quality criteria, each vnode transmits frames of 1500 Bytes on its emulated WLAN connection as fast as it can, i.e. limited by the emulated bandwidth of 100 MBit/s. Each frame has a destination MAC address, which does not exist in the emulation testbed. This forces network traffic to be sent on the physical uplink of the pnode's NIC. The emulation hardware switch (Section 2.2.1.1) works as sink and drops all those frames. Since the physical uplink has a line speed of 1 GBit/s, we expect saturation or overload with at most 10 vnodes.

Fig. 4.9 shows processed data of the event recording during the evaluation experiments each of them lasting 10 seconds. We choose 10 ms for the averaging interval duration δ_n. This corresponds to the timer interrupt period or process CPU

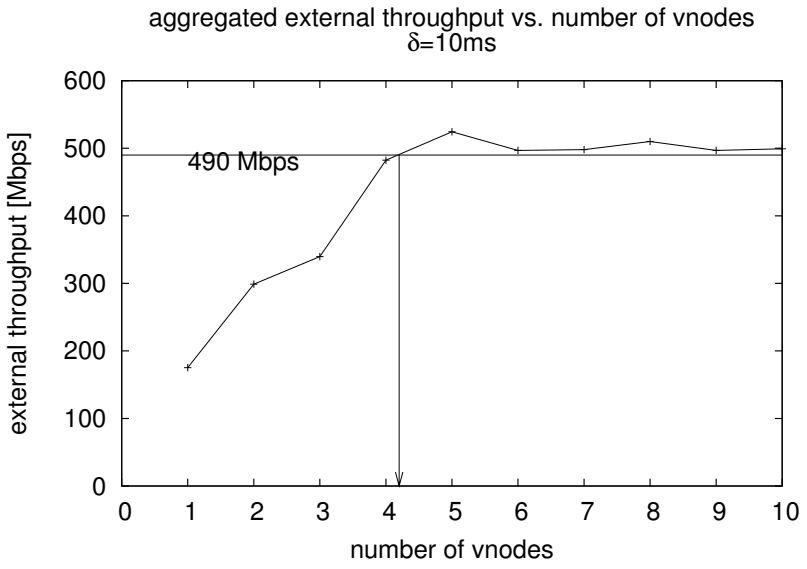

Figure 4.9: Example of varying aggregated external throughput on a pnode.

quantum respectively. The interval is long enough to allow for enough network events to calculate an average bit rate and short enough to promptly indicate resource contention. As mentioned previously in Section 4.4.1, the bottleneck is not line speed but rather the PCI bus by which the NIC is connected to the computer. Therefore, we see a saturation of the bit rate for external network traffic at a little over 490 MBit/s. Hence, we use 490 MBit/s as threshold θ_{n_e}.

We base our evaluation of the CPU quality criterion on a load generator performing computation workload without any I/O. Each vnode uses roughly a quarter of the available CPU resource. To this end, the load generator executes an endless loop. Within the loop, it first sleeps for the minimal amount of time possible with the OS. In our case, this is at least one period of the timer interrupt being 10 ms. Then, the load generator spins reading the CPU time stamp counter register in an inner loop until the fraction of total time spun and total time since the start of the generator is greater than or equal to the user-specified CPU utilization. This ends the inner loop and another cycle of the outer loop starts with sleeping the shortest possible time. On average over time this utilizes the CPU by the user-specified value.

Fig. 4.10 shows the maximum duration of non-idle intervals during the experiments for different numbers of vnodes. Each experiment lasts 20 seconds. Please note that the y-axis has logarithmic scale. The shortest sleep time s defines the duration c of the inner computation loop in order to generate the specified load l: $c/(s+c) = l \Rightarrow c = l/(1-l)s$. With $s = 10\,\text{ms}$ and $l = 1/4$, the inner loop has to

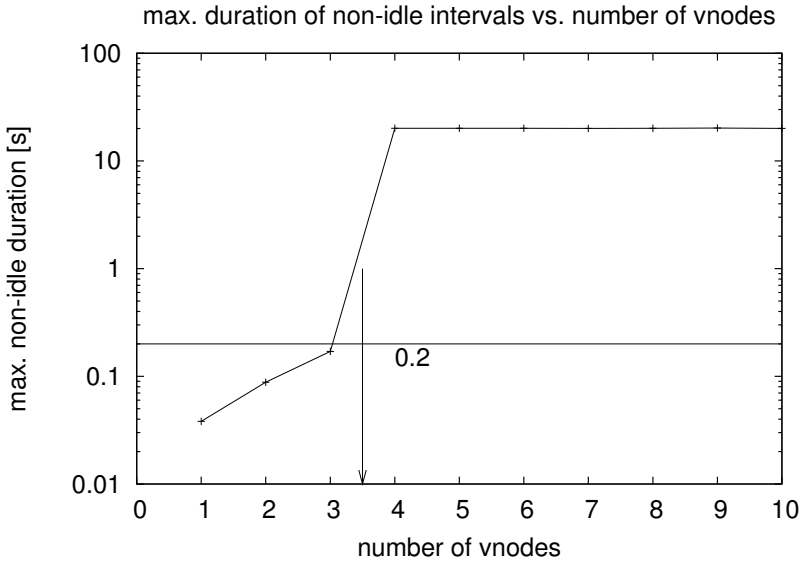

Figure 4.10: Example of varying intervals with non-idle CPU on a pnode.

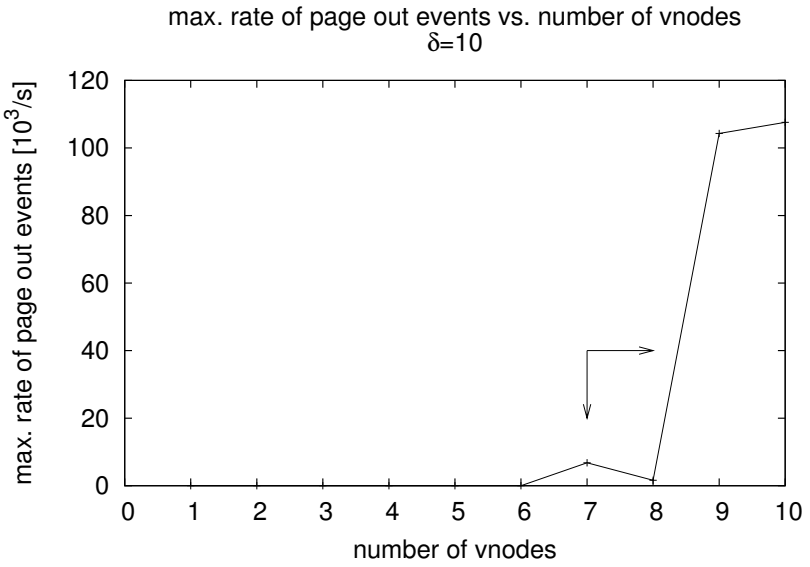

Figure 4.11: Example of varying page-out rates on a pnode.

spin for $c = 10/3$ ms. However, the outer loop cycles do not align with the timer interrupt. Hence the sleep duration can be longer than 10 ms and therefore the inner loop has to spin longer to reach CPU utilization. Together with potential other standard processes running in the OS, the maximum non-idle duration over the course of an experiment is larger than just the sum of one inner loop cycle over all vnodes. As expected, we see a significant increase and saturation in the non-idle duration starting with 4 vnodes resembling 100% load. For this experiment, we use 0.2 seconds as threshold θ_c which is slightly above the safe CPU load of 75% with 3 vnodes.

To evaluate memory quality criteria, each vnode continuously writes on its own allocated memory region of size 60 Megabytes. With the pnode having 512 MB RAM, we expect to overload memory resources with $\lceil 512/60 \rceil = 9$ vnodes. Since some memory is already used by the OS and by some user space processes, paging might even occur with less load.

Fig. 4.11 depicts the maximum rate of page out events for different numbers of vnodes. Each experiment runs for 20 seconds. As for network, we choose 10 ms for the averaging interval duration δ_m. Up to and including 6 vnodes corresponding to 360 MB load, there is no paging at all. This is a safe behavior and we choose a threshold $\theta_m = 0$ which does not permit any paging. With 7 or 8 vnodes, there is some paging. As expected, the paging rate increases significantly on overload starting with 9 vnodes.

We evaluate the disk quality criterion by letting each vnode read from its own

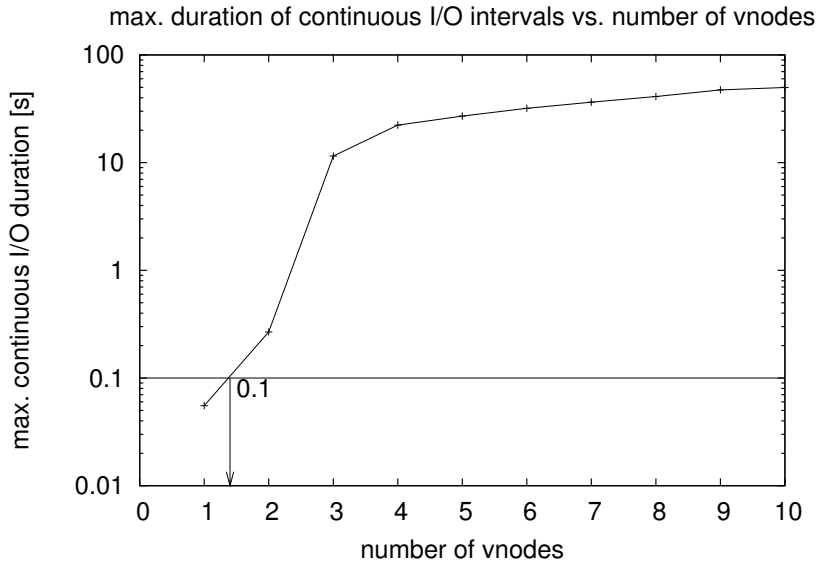

Figure 4.12: Example of varying durations with continuous disk I/O on a pnode.

file on the same disk locally attached to the pnode. The file with size 100 MB is created once before the evaluation experiments. The Data Test Program (dt) version 15.14 [Mil04] acts as load generator and sequentially reads a file. By means of command line options, we disable data comparison during read, i.e. dt reads the data and just throws it away. We also disable verification of written data, since we do not even write data. To ensure that each vnode can only access its own file and does not share it with others, we use the exclusive flag on opening the file (O_EXCL). Since the load generator must utilize the disk device and not the caches of the OS in between, we make dt use the direct open flag (O_DIRECT). With this flag, I/O requests bypass caches and unconditionally put load on the disk even if the file has been read previously and would have fit entirely into main memory due to its relatively small size. We instruct dt to not delete the file on local disk after it has been read, so it remains available for subsequent experiment runs. In order to utilize the disk only by a fraction, dt generates a mix of disk I/O and pauses. We choose the necessary parameters such that 2 vnodes would fully utilize the disk on a 1 second average but 1 vnode would not. Dt performs a 1 microsecond sleep before each I/O request. Each request uses a block size of 256 Kilobytes.

Fig. 4.12 displays the maximum duration of continuous I/O intervals for different numbers of vnodes. Please note that the y-axis is drawn with logarithmic scale. In contrast to the previous resource types, each disk experiment does not run for a fixed amount of time but rather until each vnode has read its file completely. Knowing that overload starts with 2 vnodes, we select a threshold $\theta_d = 0.1\,\mathrm{s}$,

which lies between the maximum continuous I/O duration of 1 and 2 vnodes. The more overload, the longer continuous I/O lasts. This allows detection of resource contention by means of a threshold.

4.6.2 Effectiveness of Monitoring

After having shown the effectiveness of our defined quality criteria, we now evaluate the effectiveness of our approach to monitor the criteria during emulation experiments. In contrast to Section 4.6.1, the event recording is replaced with our implementation of the monitoring algorithms from Section 4.5.1. We expect user notification for those numbers of vnodes that cause resource contention.

First, we use the same load generation scenarios as in Section 4.6.1 to intentionally overload each resource type separately. Secondly, measurements with real-life network emulation scenarios put monitoring under test.

4.6.2.1 Triggering Each Criterion Separately

The rows in Tab. 4.1 show evaluation results for each scenario, that puts load on a specific resource class with network being separated for external and internal traffic. The columns under "quality criterion" display the smallest number of vnodes that lead to a contention of the specific resource class denoted by the column name. It is important that the bold values in the descending diagonal are the smallest in each row. In that case, monitoring reports contention for the overloaded resource before notifying of any other resource type. Also, the bold values should correspond to the number of vnodes where contention occurred first in Section 4.6.1. Please note that the numbers of vnodes per pnode are relative and depend on the specific scenario and testbed hardware.

In comparison to Fig. 4.9, monitoring detects resource contention for external network traffic also with 5 vnodes. If we put even more vnodes on the same pnode, we see CPU contention starting with 8 vnodes.

experiment	**quality criterion**					
	n_e	n_i	c	m_i	m_o	d
n_e	**5**	–	8	–	–	–
n_i	–	**11**	18	–	–	–
c	–	–	**4**	–	–	–
m	–	–	1	8	**8**	8
d	–	–	–	–	–	**2**

Table 4.1: Triggering of each criterion depending on the load (number of vnodes).

For internal network traffic, we assume a default threshold $\theta_{n_i} = 1$ GBit/s. Since this resource type depends on CPU contention, it could just as well be detected with the CPU quality criterion. To be on the safe side, we assume a threshold which does not fully load the CPU. Therefore, we see contention for n_i at 11 vnodes, while 18 vnodes would definitely contend for the CPU.

CPU becomes fully loaded or overloaded respectively with 4 vnodes each of them using one quarter of the CPU resource.

In Fig. 4.11, paging starts with 7 vnodes. Here, we have a different evaluation experiment run and memory contention occurs with 8 vnodes. The small difference in the number of vnodes is due to page replacement depending on previous memory access patterns which are simply different in two experiment runs. Since all vnodes contend for memory, the OS does not only page out frames but also has to page in frames, that have been paged out previously, when the load generator writes into the same memory regions again. Therefore, we see also m_i trigger for 8 vnodes. By coincidence, the disk also becomes overloaded with 8 vnodes due to the paging activities. Here, we may ignore the fact that CPU contention seems to occur with only 1 vnode and therefore triggers before the actually loaded resource memory. It is the load generator writing repeatedly into its allocated memory space as fast as it can and therefore fully loads the CPU with even just one instance.

Two vnodes overload the pnode's local hard disk on purpose.

In rows n_i and m of Tab. 4.1, we see that quality criteria are not necessarily orthogonal but rather depend on other resource types. Internal network throughput depends on the CPU because the virtual communication switch forwards frames in software. Memory contention leads to paging, which in turn utilizes a disk to store page frames, that have been paged out, or to read them back in.

4.6.2.2 Evaluation of Typical Emulation Scenarios

After having shown that monitoring works separately for each resource type, we now evaluate monitoring with two network emulation scenarios taken from Section 3.7.3: (1) A wired router chain using static routing, and (2) a wireless ad hoc router chain using AODV-UU.

Fig. 4.13 shows the network topology of the wired router chain (1). It consists of a linear chain with a varying number of router nodes using static routing. Point-to-point links connecting the routers are full duplex and have an emulated limited bandwidth of 100 MBit/s in each direction. We place all routers inside vnodes on a single pnode except for the last router, which resides on a separate pnode without any virtualization. In Section 3.7.3.1, we find resource contention to occur with 12 vnodes when measuring TCP throughput between vnode 1 and vnode $n+1$. Therefore, *iperf* also serves as load generator here with the same TCP window size

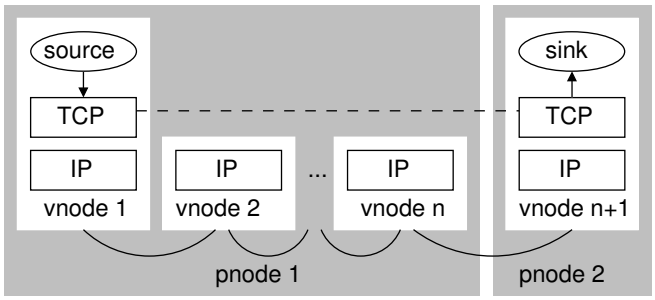

Figure 4.13: Wired infrastructure emulation scenario (same as Fig. 3.28).

(socket send or receive buffer) of 512 KBytes and maximum segment size (MSS) of 1448 Bytes.

Fig. 4.14 shows the emulation scenario of the wireless ad hoc router chain (2). For comparison with the infrastructure scenario, we configure the virtual node positions and the emulated wireless network transmission range—depicted by dotted circles—such that the connectivity of the nodes resembles a chain. The wireless links between nodes are full duplex and have a limited bandwidth of 11 MBit/s. We do not emulate the effects of a MAC layer, i.e. there are no frame collisions. Software under test is an implementation of the ad hoc on-demand distance vector routing protocol called AODV-UU [LNT02] in version 0.8. Each vnode executes an instance of the routing daemon. Similar to the wired scenario, we measure this scenario with all vnodes on a single pnode, except for the last node, which resides on a separate pnode. In Section 3.7.3.1, we find resource contention to occur with 29 vnodes when measuring TCP throughput with the same workload configuration as for the wired scenario.

The rows of Tab. 4.2 show our evaluation results. The “experiment” column identifies the emulation scenario. The “reference” column contains the smallest number of vnodes hosted on pnode 1 for which the expected measurement results

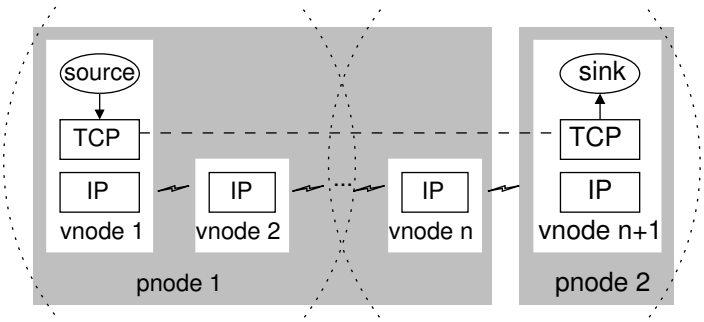

Figure 4.14: Wireless ad hoc network emulation scenario (same as Fig. 3.33).

experiment	quality criterion						reference
	n_e	n_i	c	m_i	m_o	d	Sec. 3.7.3
(1)	–	**11**	34	–	–	–	12
(2)	–	–	**27**	3	–	3	29

Table 4.2: Triggering of quality criteria in typical emulation scenarios.

start to deviate in Section 3.7.3 due to resource contention. We use those values as a reference to compare with the smallest number of vnodes for each quality criterion in the middle columns.

For the wired scenario (1), monitoring reports contention of the network resource for internal traffic at 11 vnodes. Compared to 12 vnodes for the reference, monitoring detects contention with slightly less load which is fine. Apparently, $\theta_{n_i} = 1$ GBit/s turns out to be a good threshold for our testbed hardware. If we had only used the CPU criterion, monitoring would have reported contention starting with 34 vnodes. In comparison to Fig. 3.31, this would not have detected contention starting already with less vnodes.

While the reference indicates contention with 29 vnodes for the wireless scenario (2), our monitoring of quality criteria reports CPU contention with 27 vnodes. Starting with 3 vnodes, some page-in events occur and get reported since we use a strict threshold $\theta_{m_i} = 0$ not allowing any paging. In turn, the resulting bursts of disk read activity contend for the disk resource. However, this only happens at the beginning of each experiment, when the AODV routing daemons build up routing tables on the first packets being sent by the load generator. We are confident that paging does not occur afterward in the measurement phase. The scenario automation scripts for the experiment preparation phase (Section 2.1.3.1) could be improved such that they start the routing daemons first and send some packets before resetting the monitoring parameters and finally starting the measurement and load generators.

4.6.3 Overhead of Monitoring

In order to evaluate the overhead of monitoring quality criteria, we use the same evaluation scenarios as in Section 4.6.2.1, where we trigger each quality criterion separately. For the experiments n_e, n_i, c, m, d, we use the lowest load in number of vnodes that triggers the report of resource contention, i.e. $5, 11, 4, 8, 2$ correspondingly. In contrast to the evaluation of monitoring, the monitoring procedure for each resource type now contains additional profiling code at the beginning and the end. We use the CPU time stamp counter register to measure the execution duration of each monitoring procedure. The profiling code also aggregates the durations and number of procedure executions. Those aggregates enable the calculation of the

average execution duration for each monitoring procedure by taking the fraction of execution duration sum and execution count. In addition, the code keeps track of the minimal and maximum duration of each procedure execution.

Except for the disk resource scenario, all experiments load the CPU fully. The memory resource scenario and the CPU scenario with ≥ 4 vnodes take up all CPU resources, in order to generate load (see Tab. 4.1). During the evaluation of network resource criteria, we run a simple CPU consumer process in parallel to the actual network load generator. The consumer process spends CPU in an endless loop by counting iterations in a variable. That variable is declared as volatile, in order for the compiler not to eliminate the loop body when building with optimization enabled. Having the CPU fully utilized during the evaluation experiments allows the following definition of monitoring overhead: ratio of the sum of time spent in monitoring code and the experiment duration.

The rows of Tab. 4.3 correspond to the evaluation experiments named in the first column. The two rows m_i and m_o are results from the same experiment m putting load on the memory resource and causing both page-in and page-out events.

Execution counts in the second column show how often the corresponding monitoring algorithm procedure is called during each experiment. For the first three experiments, we can approximate the execution count to validate the measured numbers. For n_e, 5 vnodes generate load with frames of 1514 bytes at a rate of 100 Mbit/s for 10 seconds resulting in $5 \cdot 10^8 \cdot 10/(1514 \cdot 8) \approx 413{,}000$ calls closely matching the measured 415,170. With n_i, we use 11 vnodes instead resulting in $\approx 908{,}000$ procedure executions compared to the measured 913,329. When a CPU load generator gets scheduled it sleeps for at least 10 ms. By doing so, it yields the CPU and calls the scheduler and thus the CPU monitoring procedure once. When the scheduler wakes up the load generator process, the monitoring procedure is called a second time and the generator loops for 10/3 ms. For the experiment duration of 20 seconds, the procedure gets called twice every $(10 + 10/3)$ ms $= 40/3$ ms resulting in an estimated execution count of 750. However, the generators loop more than 10/3 ms to catch up with unavoidable sleeps longer than 10 ms. Therefore, there are less process switches and thus less monitoring procedure executions leading to measured 507 executions.

Minimal monitoring procedure execution duration is below 40 nanoseconds for all resource types.

The maximum execution durations times between 17 and 25 milliseconds occur on resource contention. Monitoring then triggers user notification, which prints out kernel messages. We have a serial console attached to the testbed computer running the evaluation experiments and kernel messages are copied onto this serial console. Since this happens synchronously, the printing of the notification message

experiment	execution count	min (ns)	max (ms)	average (ns)	experiment duration (s)	overhead (10^{-6})
n_e	415,170	38	22	248	10	0.025
n_i	913,329	38	25	122	10	0.012
c	507	37	17	101,905	20	5.095
m_i	9,592	38	17	5,457	20	0.273
m_o	9,653	38	18	5,495	20	0.275
d	51,333	38	18	1,109	11	0.101

Table 4.3: Overhead of monitoring.

is relatively slow. At a serial line speed of 115.2 kbit/s the minimal time to print just one line with 80 characters of 8 bits each is $8\,\text{bit/char} \cdot 80\,\text{char}/(115{,}200\,\text{bit/s}) \approx 5.6\,\text{ms}$. This is not much of a problem since logging without a serial console is much faster. Also, an emulation experiment with resource contention would be aborted anyway so the notification overhead does not matter. In any case, even tens of milliseconds fulfill our requirement for fast notification within a split second.

Average execution duration of monitoring procedures is in the order of hundreds of nanoseconds up to hundred microseconds. By default, a maximum of three contentions is reported for each resource type and our experiments' load triggers all three notifications. Therefore, three monitoring procedure executions are likely to last close to the maximum duration, while all other executions are closer to the minimal duration. The less executions in total, the larger is the influence of the three contention notifications on average duration and overhead. CPU monitoring generates the largest overhead of $\approx 0.0005\,\%$, which we consider negligible. Monitoring overhead for other resource types is even less.

4.7 Summary

We discussed four different approaches to cope with resource contention in virtualized network emulation and came to the conclusion that detection and recovery suits our goals for scalability best. Detection of resource contention should fulfill the requirements: fast contention detection, monitoring with minimal overhead, and tracking resource consumption separately for each vnode. Detection is enabled by monitoring resources and applying quality criteria, that characterize contention. For the three different approaches to define quality criteria, we ruled out mathematical and systematic. Following an empirical approach, we instrumented an operating system kernel to obtain fine-granular resource scheduling event traces that can be analyzed offline. Based on observed events, we defined quality criteria for four resource classes: network, CPU, memory, and disk. For each of those

classes, we described monitoring algorithms, that work inside an OS kernel and track resource consumption for each vnode. Our evaluations showed that our quality criteria enable the detection of contention for each resource class. Our monitoring prototype in the Linux kernel notifies the user on resource contention, if we intentionally overload each resource class, as well as in typical network emulation scenarios, if too many vnodes are mapped to a pnode. Mapping vnodes to pnodes and therefore also adapting the mapping to recover from contention is beyond the scope of this thesis. Finally, the evaluation of overhead demonstrates that monitoring notifies fast on detecting contention and it works with minimal overhead, which is negligible.

Chapter 5

Conclusion

5.1 Summary

The ongoing advancement of computer network technologies and corresponding distributed application scenarios require improved or new protocols on all layers of the protocol stack including the application layer. The performance of protocols and applications needs to be evaluated. Network emulation enables reproducible performance measurements of unmodified implementations of software under test including protocols and applications.

In this thesis, we presented the foundations of network emulation. Regarding the architecture of a network emulation system, we discussed the integration into the protocol stack, different emulation modules, and emulation functionality. Our discussion showed that an emulation tool should reproduce the properties of the medium access control layer. There are three types of emulation modules: the model storing static and dynamic parameters of an emulation scenario, the tool reproducing MAC layer properties based on the model state, and the control maintaining the model. Each module type can be centralized or distributed among the multiple computers of an emulation testbed. We showed that the following architecture promises to be the most scalable: a hybrid model consisting of a global centralized part and distributed parts at each tool, tools distributed per emulated network connection, and a centralized control. We presented the necessary emulation functionality, that control and tools need to implement. Based on this architecture, we described our implementation of the so-called „Network Emulation Testbed." The testbed comprises computers and their network interconnection as well as distributed emulation tools and a centralized emulation control component.

Realistic network emulation scenarios often require a number of nodes which is larger than the available number of emulation testbed computers. Hence, there is need for emulation that scales in the number of nodes. We classified the

related work for scalable network emulation into parallelization, emulation model abstraction, and node virtualization. Parallelization requires synchronization and its processing of network traffic may constitute a bottleneck. Emulation model abstraction is more suitable to generate synthetic network load for cross or background traffic. Since a multitude of software under test only needs a fraction of a testbed computer's resources, each such computer (pnode) can execute multiple nodes of an emulation scenario (vnode). Therefore, node virtualization enables scalable network emulation.

For node virtualization, we stated the following three requirements: Transparency to support unmodified software under test, flexibility to mix different software under test on each testbed computer, and efficiency to make best use of available testbed resources. As a basis for further discussion, we presented foundations of protocol stacks with focus on the protocol entities and their data structures holding state.

There are two major approaches to virtualize those entities: virtual machines and operating system partitioning. From the different classes of virtual machines, the system virtual machine with its two subtypes hosted virtual machine and classic system virtual machine are most relevant for scalable network emulation. Virtual protocol stacks from the area of operating system partitioning fulfill our requirements for transparency and flexibility partially and they fulfill our requirement for efficiency best. Therefore, we considered virtual protocol stacks the best approach to scalable network emulation.

Our architecture consists of software communication switches, virtual protocol stacks, design patterns for virtual node configuration, and hierarchical emulation control. A software communication switch (or virtual switch) interconnects vnodes on the same as well as on different pnodes. It does so in a highly efficient manner without copying traffic payload. A virtual protocol stack on each pnode provides multiple instances of software under test with execution environments. Hierarchical emulation control consists of a central controller component, proxies on each pnode, and clients on each vnode, in order to scale emulation control to the potentially large number of vnodes.

We validated our choice of virtual protocol stack over hosted virtual machine and classic system virtual machine with regard to efficiency by experiments. Our software communication switch decides on average within 98 ns and processes up to 3 GBit/s aggregate traffic with 1500 Byte unicast frames on our given hardware. Our emulation tool faithfully reproduces delay and frame loss independent of the number of vnodes per pnode. We showed the inherent limits of emulating bandwidth with a variable number of vnodes per pnode. With our emulation scenarios, TCP throughput turned out to be the limiting factor as to how many vnodes can be hosted by a pnode. For a wired infrastructure scenario with a chain

of routers, 11 vnodes per pnode still allow realistic measurement results. With a wireless ad hoc network scenario, we were able to map 28 vnodes on a single pnode. Node virtualization thus enables the support of up to 1792 nodes for similar wireless scenarios on our available testbed hardware with 64 pnodes.

Vnodes on the same pnode share the limited resources of their hosting pnode. This may lead to resource contention, which can cause unrealistic measurement results and is thus undesirable. We discussed four possible approaches to cope with undesired resource contention: ignoring, prevention, avoidance, detection and recovery. The latter suits our paramount goal for scalability best. Detection is enabled by monitoring quality criteria. We handle recovery by aborting emulation runs with resource contention. Adapting the mapping of vnodes to pnodes, to avoid contention in another emulation run, is beyond the scope of this thesis. We classified our related work into network monitoring, scalable network emulation, and code instrumentation. Since other resources besides network, e.g. CPU, need to be monitored with emulation based on node virtualization, network monitoring is not sufficient and considered possible software under test. Existing scalable emulation approaches require previous benchmarking runs or can hardly detect resource usage peaks without significant monitoring overhead. Code instrumentation can serve as the base mechanism to implement monitoring of quality criteria.

We defined the following requirements for detection of resource contention: fast detection within a split second, monitoring with minimal overhead, and account for each vnode's share of resource consumption. We discussed three approaches to defining quality criteria: mathematical, systematic, and empirical. Based on an empirical approach, we defined quality criteria for the following resource types: network, CPU, memory, and disk. Our monitoring approach is based on instrumenting basic resource events in the operating system kernel. We presented algorithms to monitor quality criteria for network, CPU, memory, and disk resources.

With extensive evaluations, we showed the effectiveness of our quality criteria by deliberately overloading each resource type separately. Measurements for the effectiveness of our monitoring approach were in accordance with the evaluation of quality criteria effectiveness as well as with our findings for the evaluation of node virtualization. We measured monitoring overhead not exceeding $\approx 0.0005\,\%$ which is negligible.

Node virtualization in combination with monitoring of emulation quality criteria provides network emulation, that scales in the number of nodes and the network traffic within an emulation scenario.

5.2 Future Work

Scale Wave Propagation Look-up

In Section 2.1.1.6, we showed that the optimal emulation abstraction layer is the medium access control protocol layer and emulation tools should reproduce its network properties. On top of this layer, node virtualization based on virtual protocol stacks presented in Section 3.5.2.1 partitions network, transport, and application layer. This virtualization enables scaling the number of nodes in a network emulation scenario. To realistically emulate frame loss in wireless scenarios with obstacles and mobile nodes, we periodically look wave propagation properties up on disk storage as described in Section 2.2.3. The increase of the possible number of nodes per scenario also increases the number of look-ups. In order for the wave propagation look-up not to become a bottleneck and potentially cause unrealistic measurement results, the look-up needs to scale with the number of virtual nodes. Since reading from disk storage is expensive, methods are required to reduce the number of disk accesses and especially disk seeks, which are particularly expensive. A first step towards this goal is to reduce wave propagation data size by selection and compression. Then, the data can be organized and indexed for efficient sequential read access. Each disk access should read a block of data which provides all possible propagation properties for a certain interval of future experiment time. This reduces disk accesses and disk seeks. The data blocks read can be stored in a cache for looking up properties within the time interval. Application specific replacement algorithms can optimize the cache usage. Initial research was conducted for wave propagation data selection and organization in a study thesis [Dic07] and for caching in a diploma thesis [Häu07].

Exploit Multi-core Processors

While testbed computers used in this thesis were single processor systems, modern computers tend to be built of multiple multi-core processors. With single processor computers, the number of virtual nodes, that can be executed safely and efficiently on each testbed computer, solely depends on the resource requirements of each virtual node and its contained software under test. Multi-processor and multi-core systems introduce a non-uniformity with regard to resource access from each processor or core since they share some resources such as main memory. The mapping of virtual nodes onto testbed computers should take this fact into account, in order to best exploit such non-uniform systems. For instance, network communication between cores on the same processor can be more efficient than between processors or even testbed computers. To avoid expensive locking when

accessing the shared memory resource, the main memory could be partitioned and thus provide each processor or core with its own exclusive memory.

Virtualize Time

Our summary of comparing node virtualization approaches in Section 3.5.3 showed that virtual machines fulfill our requirements transparency and flexibility well, whereas virtual protocol stacks excel at our efficiency requirement. A performance analyst faces a trade-off between scalability on the one side and transparency or flexibility on the other side. This limitation could be mitigated by virtual machine monitors, that provide software under test with virtual time to enable non-real-time network emulation [GYM+06]. The ability to execute software under test slower than real-time provides virtually increased resources of a network emulation testbed computer. By multiplying the wall clock time with a time dilatation factor (TDF), consumable testbed computer resources such as CPU and especially network appear scaled by the same factor. The software under test is able to do more work in its perceived virtual time. Virtual machines fulfill our requirements transparency and flexibility, virtual time enables scalability in terms of large emulation scenarios at the cost of emulation experiment duration. Thus, experiment execution time may be traded for scalability, transparency, *and* flexibility in the future.

Executing an emulation experiment with a constant time dilatation factor would require selection of a factor that provides enough virtual resources during peak usage. However, during experiment times with lower than maximum resource usage, the experiment duration would be unnecessarily stretched. Total experiment duration can be optimized by adapting the time dilatation factor dynamically during an experiment depending on the resource demand of the software under test. As opposed to a constant TDF, the other extreme case would be parallel discrete event simulation (PDES) which effectively resembles adaptation of the TDF for each event. Yet, PDES requires synchronization between parallel instances which can become a significant overhead. An intermediate approach between constant TDF and PDES is to divide experiment duration into epochs. During each epoch, distributed emulation tools run with constant TDF, independent, and without any need for expensive distributed synchronization. Epoch transitions and the calculation of a new TDF can be based on the results on monitoring resource contention of this thesis. A different sub-project of the network emulation testbed research project studies scalable network emulation with epoch-based virtual time [GMHR08, GHR09].

List of Figures

List of Tables

List of Equation Symbols

Chapter 2: Foundations of Computer Network Emulation

b bit rate

t_s serialization delay

t_p propagation delay

t_m medium access delay

p_b bit error rate (BER)

p_f frame error rate (FER)

t time of local clock

l frame length

e_{sd} 3-tuple (p_b, b, t_p) of global model parameters between node s and node d

Chapter 4: Resource Contention in Virtualized Emulation

t time stamp

l frame length

$e_{n_e} = (t, l)$ network event for egress traffic with external destination

$e_{n_i} = (t, l)$ network event for egress traffic with internal destination

θ_{n_e} threshold for transmission rate of egress traffic with external destination

δ_{n_e} time interval for transmission rate measurement of egress traffic with external destination

θ_{n_i} threshold for transmission rate of egress traffic with internal destination

δ_{n_i} time interval for transmission rate measurement of egress traffic with internal destination

t_s beginning of idle interval

t_e end of idle interval

$e_c = (t_s, t_e)$ CPU event

θ_c threshold for time interval with non-idle CPU

$e_{m_i} = (t)$ memory event for page in

$e_{m_o} = (t)$ memory event for page out

θ_{m_i} threshold for page in rate

δ_{m_i} time interval for measurement of page in rate

θ_{m_o} threshold for page out rate

δ_{m_o} time interval for measurement of page out rate

r number of I/O requests waiting in device queue to be processed

$e_d = (t, r)$ disk event

θ_d threshold for time interval with non-idle disk

Bibliography

[ABCG04] G. Anastasi, E. Borgia, M. Conti, and E. Gregori. Wi-Fi in Ad Hoc Mode: A Measurement Study. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, pages 145–154, Orlando, FL, USA, March 2004.

[ABKM01] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 131–145, Banff, Alberta, Canada, October 21–24 2001.

[ABKM03] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Experience with an Evolving Overlay Network Testbed. *ACM SIGCOMM Computer Communication Review*, 33(3):13–19, July 2003.

[AC06] George Apostolopoulos and Constantinos Chasapis. V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation. Technical Report 371, ICS-FORTH, Greece, January 2006.

[AH06] George Apostolopoulos and Constantinos Hasapis. V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS'06)*, pages 117–126, Monterey, CA, USA, September 11–14 2006.

[Aiv01] Tigran Aivazian. Linux Kernel 2.4 Internals, 23 August 2001. Linux Documentation Project.

[APST05] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the Internet Impasse through Virtualization. *IEEE Computer*, pages 34–41, April 2005.

[Ara03] Jesus Arango. Virtual IP Machines: A System Framework for Emulating Multiple IP Hosts. Technical Report TR03-14, University of Arizona, Gould-Simpson 721, Tucson, AZ 85721, October 10 2003.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[AWW05] Ian F. Akyildiz, Xudong Wang, and Weilin Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445–487, March 15 2005.

[Bau07] Martin Peter Bauer. *Observing Physical World Events through a Distributed World Model*. PhD thesis, University of Stuttgart, Institute of Parallel and Distributed Systems, May 11 2007.

[BBC+04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, USA, March 29–31 2004.

[BBD+01] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Claus Schröter, and Dirk Verworner. *Linux-Kernelprogrammierung: Algorithmen und Strukturen der Version 2.4*. Addison-Wesley, 6., aktualisierte und erweiterte edition, 2001.

[BBHL08] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual Servers and Checkpoint/Restart in Mainstream Linux. *ACM SIGOPS Operating Systems Review*, 42(5):104–113, July 2008.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, Bolton Landing, New York, USA, October 19–22 2003.

[BEF+00] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[BFH+06] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '06)*, pages 3–14, Pisa, Italy, September 11–15 2006.

[BL03] Paul Barford and Larry Landweber. Bench-style Network Research in an Internet Instance Laboratory. *ACM SIGCOMM Computer Communications Review*, 33(3):21–26, 2003.

[BMJ+98] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, pages 85–97, Dallas, Texas, United States, October 25–30 1998.

[BS01] Kerstin Buchacker and Volkmar Sieh. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects. In *Proceedings of the Third IEEE International High-Assurance System Engineering Symposium (HASE 2001)*, pages 95–105, Boca Raton, Florida, 2001.

[Cas05] Mirko Casper. Verzögerungszeiten von Vermittlungsstellen in einem Netzwerkemulationssystem. Studienarbeit Nr. 1969, Universität Stuttgart, IPVS, Abteilung Verteilte Systeme, March 2005.

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communications Review*, 33(3):3–12, July 2003.

[Com91] Commission of the European Communities. Urban transmission loss models for mobile radio in the 900 and 1,800 MHz bands. Vol. COST 231 TD (91) 73, European Cooperation in the Field of Scientific and Technical Research, September 1991.

[Cox96] Alan Cox. Kernel Korner: Network Buffers and Memory Management. *Linux Journal*, Tuesday, 01 October 1996.

[CS99] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP, Volume 2: Design, Implementation, and Internals*. Prentice Hall, 3rd edition, 1999.

[CS03] Mark Carson and Darrin Santay. NIST Net — A Linux-based Network Emulation Tool. *ACM SIGCOMM Computer Communications Review*, 33(3):111–126, July 2003.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, USA, June 27–July 2 2004.

[DC01] Chris Dalton and Tse Huong Choo. An Operating System Approach to Securing e-Services. *Communications of the ACM*, 44(2):58–64, 2001.

[Dic07] Björn Dick. Organisation multidimensionaler Wellenausbreitungsdaten auf Festplatte für effizienten Lesezugriff. Studienarbeit Nr. 2099, Universität Stuttgart, IPVS, Abteilung Verteilte Systeme, August 31 2007.

[Dik00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta, Georgia, USA, October 10–14 2000.

[Dud02] Dominique Dudkowski. Emulationskonzepte für Mobilfunk- und Ad-Hoc-Netze. Diplomarbeit Nr. 2004, Universität Stuttgart, Abteilung Verteilte Systeme, 2002.

[EKL05] Sherif M. ElRakabawy, Alexander Klemm, and Christoph Lindemann. TCP with Adaptive Pacing for Multihop Wireless Networks. In *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'05)*, pages 288–299, Urbana-Champaign, IL, USA, 2005.

[EKL06] Sherif M. ElRakabawy, Alexander Klemm, and Christoph Lindemann. Gateway Adaptive Pacing for TCP across Multihop Wireless Networks and the Internet. In *Proceedings of the 9th ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM'06)*, pages 173–182, Terromolinos, Spain, 2006.

[Ela02] Hala Elaarag. Improving TCP Performance over Mobile Networks. *ACM Computing Surveys*, 34(3):357–374, 2002.

[ELK07] Pavel Emelianov, Denis Lunev, and Kirill Korotaev. Resource Management: Beancounters. In *Proceedings of the Linux Symposium (OLS '07)*, pages 285–292, Ottawa, Ontario, Canada, June 27–30 2007.

[ESHF04] Michael Engel, Matthew Smith, Sven Hanemann, and Bernd Freisleben. Wireless Ad-Hoc Network Emulation Using Microkernel-Based Virtual Linux Systems. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*, pages 198–203, Marne la Vallée, France, 2004.

[ESW01] David Ely, Stefan Savage, and David Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS 2001)*, San Francisco, California, USA, 26–28 March 2001.

[Fal99] Kevin Fall. Network Emulation in the Vint/NS Simulator. In *Proceedings of the Fourth IEEE Symposium on Computers and Communications (ISCC'99)*, pages 244–250, Red Sea, Egypt, July 6–8 1999.

[FHH+03] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. TCP Performance Re-Visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2003)*, pages 70–79, Austin, TX, USA, March 6–8 2003.

[FHN+04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the OASIS ASPLOS 2004 Workshop*, 2004.

[FL06] Vince Fuller and Tony Li. *Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632*, August 2006.

[Fri46] H. T. Friis. A note on a simple transmission formula. *Proceedings of the IRE*, 34:254–256, 1946.

[FV02] Kevin Fall and Kannan Varadhan. *The ns Manual*. UC Berkeley, LBL, USC/ISI, and Xerox PARC, 2002.

[GD04] Ashish Gupta and Peter A. Dinda. Inferring the Topology and Traffic Load of Parallel Programs Running In a Virtual Machine Environment. In *Proceedings of the 10th Annual Workshop on Job Scheduling Policies for Parallel Processing (JSSPP) in conjunction with Sigmetrics 2004*, pages 62–72, Columbia University, New York, NY, USA, June 13 2004.

[GEN06] GENI Planning Group. GENI Design Principles. *IEEE Computer*, 39(9):102–105, September 2006.

[GF04] Yan Gu and Richard Fujimoto. A Flexible Architecture for Remote Server-Based Emulation. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004)*, pages 447–454, Volendam, The Netherlands, October 5–7 2004.

[GHR09] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Efficient and Scalable Network Emulation using Adaptive Virtual Time. In *Proceedings of the 18th International Conference on Computer Communications and Networks (ICCCN 2009)*, San Francisco, CA, USA, August 2–6 2009.

[Gia06] Paolo Giarrusso. SKAS3-v8.2 for UML-hosts. http://www.user-mode-linux.org/~blaisorblade/patches/skas3-2.6/skas-2.6.11-v8.2/, visited November 2006.

[GMHR08] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. Time Jails: A Hybrid Approach for Scalable Network Emulation. In *Proceedings of the 22nd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2008)*, pages 7–14, Rome, Italy, June 3–6 2008.

[Gol73] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Div. Engineering & Applied Physics, Harvard University, Cambridge, MA, USA, February 1973.

[GSHL03] Shashi Guruprasad, Leigh Stoller, Mike Hibler, and Jay Lepreau. Scaling Network Emulation with Multiplexed Virtual Resources, August 26 2003. SIGCOMM 2003 Poster Abstract.

[Gu08] Yan Gu. *ROSENET: A Remote Server-Based Network Emulation System*. PhD thesis, College of Computing, Georgia Institute of Technology, April 2008.

[GWS06] Andreas Grau, Harald Weinschrott, and Christopher Schwarzer. Evaluating the Scalability of Virtual Machines for Use in Computer Network Emulation. Fachstudie Softwaretechnik Nr. 60, Universität Stuttgart, IPVS, Abteilung Verteilte Systeme, October 31 2006.

[GYM+06] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the 3rd*

ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI 06), pages 87–100, San Jose, CA, USA, May 8–10 2006.

[GZS+06] Ashish Gupta, Marcia Zangrilli, Ananth I. Sundararaj, Anne I. Huang, Peter A. Dinda, and Bruce B. Lowekamp. Free Network Measurement For Adaptive Virtualized Distributed Computing. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 25–29 2006.

[Hen02] Bryan Henderson. Linux Loadable Kernel Module HOWTO, 21 May 2002. Linux Documentation Project.

[Her05] Daniel Herrscher. *Emulation von Rechnernetzen zur Leistungsanalyse von verteilten Anwendungen und Netzprotokollen*. PhD thesis, University of Stuttgart, Institute of Parallel and Distributed Systems, December 2005.

[HH02] Edwin Hernandez and Abdelsalam (Sumi) Helal. RAMON: Rapid-Mobility Network Emulator. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)*, pages 809–817, Tampa, Florida, November 6–8 2002.

[HLR02] Daniel Herrscher, Alexander Leonhardi, and Kurt Rothermel. Modeling Computer Networks for Emulation. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '02)*, pages 1725–1731, Las Vegas, June 2002.

[HM04] Daniel Herrscher and Steffen Maier. *NET: The Network Emulation Testbed Manual*. IPVS, 2004.

[HML07] Jörg Hähner, Steffen Maier, and Andreas Lachenmann. Konzepte der Netzprogrammierung. Lecture at the University of Stuttgart, Institute of Parallel and Distributed Systems, Distributed Systems Department, 2007.

[HMR03] Daniel Herrscher, Steffen Maier, and Kurt Rothermel. Distributed Emulation of Shared Media Networks. In *Proceedings of the 2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2003)*, pages 226–233, Montréal, Quebec, Canada, July 20–24 2003.

[HMTR04] Daniel Herrscher, Steffen Maier, Jing Tian, and Kurt Rothermel. A Novel Approach to Evaluating Implementations of Location-Based Software. In *Proceedings of the 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2004)*, pages 484–490, San Jose, CA, USA, July 25–29 2004.

[Hoh05] Jörg Hoh. Definition und Überwachung von Qualitätskriterien für realitätsnahe und skalierbare Netzwerkemulation. Diplomarbeit Nr. 2267, Universität Stuttgart, IPVS, Abteilung Verteilte Systeme, April 3 2005.

[HR02] Daniel Herrscher and Kurt Rothermel. A Dynamic Network Scenario Emulation Tool. In *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN 2002)*, pages 262–267, Miami, FL, USA, October 2002.

[HRS+04] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Feedback-directed Virtualization Techniques for Scalable Network Experimentation. University of Utah Flux Group Technical Note FTN-2004-02, School of Computing, University of Utah, May 2004.

[HRS+08] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 113–128, Boston, MA, USA, June 22–27 2008.

[HS03] Ningning Hu and Peter Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE Journal on Selected Areas in Communications*, 21(6):879–894, August 2003.

[HSK99] X. W. Huang, R. Sharma, and Srinivasan Keshav. The ENTRAPID Protocol Development Environment. In *Proceedings of the Conference on Computer Communications (INFOCOM '99)*, volume 3, pages 1107–1115, New York, NY, USA, March 21–25 1999.

[Häu07] Philipp Häuser. Prefetching und Caching für effizienten Lesezugriff auf mehrdimensionale Wellenausbreitungsdaten auf Festplatte. Diplomarbeit Nr. 2604, Universität Stuttgart, IPVS, Abteilung Verteilte Systeme, October 31 2007.

[IEE99a] IEEE, New York, NY, USA. *ANSI/IEEE Std 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[IEE99b] IEEE, New York, NY, USA. *IEEE Std 802.11b: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band*, September 16 1999.

[IEE01] IEEE, New York, NY, USA. *IEEE Std 802.11b-1999/Cor 1-2001: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band – Corrigendum 1*, November 7 2001.

[IEE03] IEEE, New York, NY, USA. *IEEE Std 802.1Q: Virtual Bridged Local Area Networks*, 2003.

[IEE05] IEEE, New York, NY, USA. *IEEE Std 802.3-2005: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, December 9 2005.

[Int04a] Intel, USA. *IA-32 Intel Architecture Optimization Reference Manual*, 2004. Order number: 248966-011.

[Int04b] International Business Machines Corporation. *How to Improve the Performance of Linux on z/VM with Execute-in-Place Technology*, 1st edition, March 2004. LNUX-HTDS-00.

[Jac97] Van Jacobson. Pathchar — a tool to infer characteristics of Internet paths, April 21 1997. Talk presented at the Mathematical Sciences Research Institute.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., New York, NY, 1991.

[JM96] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In Tomasz Imielinski and Hank Korth, editors, *Mobile Computing*, volume 353, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[JPP92] Jonathan Jedwab, Peter Phaal, and Bob Pinna. Traffic Estimation for the Largest Sources on a Network, Using Packet Sampling with Limited Storage. HP Labs Technical Report HPL-92-35, (Colorado Telecommunications Division) Management, Mathematics and Security Department. HP Laboratories Bristol, March 5 1992.

[JS03] Glenn Judd and Peter Steenkiste. Repeatable and Realistic Wireless Experimentation through Physical Emulation. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, MA, USA, November 20–21 2003.

[JSF+06] David Johnson, Tim Stack, Russ Fish, Daniel Montrallo Flickinger, Leigh Stoller, Robert Ricci, and Jay Lepreau. Mobile Emulab: A Robotic Wireless and Sensor Network Testbed. In *Proceedings of the 25th Annual Conference on Computer Communications (INFOCOM 2006)*, Barcelona, Spain, April 23–29 2006.

[JWL04] Cheng Jin, David X. Wei, and Steven H. Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, volume 4, pages 2490–2501, Hong Kong, March 7–11 2004.

[JX03] Xuxian Jiang and Dongyan Xu. vBET: a VM-Based Emulation Testbed. In *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools '03)*, pages 95–104, Karlsruhe, Germany, August 25 2003.

[Kar04] Mehdi Karim. Emulation of an Algorithm for Data Replication in Mobile Ad Hoc Networks. Master's Thesis No. 2132, Universität Stuttgart, IPVS, Abteilung Verteilte Systeme, 2004.

[KDC03] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 71–84, San Antonio, Texas, USA, June 9–14 2003.

[KHF06] Eddie Kohler, Mark Handley, and Sally Floyd. Designing DCCP: Congestion Control Without Reliability. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '06)*, pages 27–38, Pisa, Italy, September 11–15 2006.

[KHS+03] Kenichi Kourai, Toshio Hirotsu, Koji Sato, Osamu Akashi, Kensuke Fukuda, Toshiharu Sugawara, and Shigeru Chiba. Secure and Manageable Virtual Private Networks for End-users. In *Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks (LCN'03)*, pages 385–394, Bonn/Königswinter, Germany, October 20–24 2003.

[Kid04] Cameron Kiddle. *Scalable Network Emulation*. PhD thesis, Department of Computer Science, University of Calgary, November 2004.

[KKKM07] Nate Kushman, Srikanth Kandula, Dina Katabi, and Bruce M. Maggs. R-BGP: Staying Connected in a Connected World. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, pages 341–354, Cambridge, MA, USA, April 11–13 2007.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium (OLS '07)*, pages 225–230, Ottawa, Ontario, Canada, June 27–30 2007.

[kl99] kossak and lifeline. Building Into The Linux Network Layer. *Phrack Magazine*, 9(11), 09 September 1999. `http://www.phrack.org/issues.html?issue=55&id=12#article`.

[KMJ00] Qifa Ke, David A. Maltz, and David B. Johnson. Emulation of Multi-Hop Wireless Ad Hoc Networks. In *Proceedings of the Seventh International Workshop on Mobile Multimedia Communications (MoMuC 2000)*, Tokyo, Japan, October 23–26 2000.

[KSU05] Cameron Kiddle, Rob Simmonds, and Brian Unger. Improving Scalability of Network Emulation through Parallelism and Abstraction. In *Proceedings of the 38th Annual Simulation Symposium (ANSS'05)*, pages 119–129, San Diego, CA, USA, April 4–6 2005.

[Kuc03] Amit P. Kucheria. Scalable Emulation of IP Networks through Virtualization. Master's thesis, Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas, May 9 2003.

[KW00] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, Maastricht, The Netherlands, May 22–25 2000.

[Lan99] F. M. Landstorfer. Wave propagation models for the planning of mobile communication networks. In *Proceedings of the 29th European Microwave Conference*, pages 1–6, Munich, Germany, October 1999.

[Law96] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29):7, September 1 1996.

[Law00] Kevin Lawton. plex86: an i80x86 virtual machine. In *Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta, Georgia, USA, October 10–14 2000. USENIX Association.

[LC04] Xin Liu and Andrew A. Chien. Realistic Large-Scale Online Network Simulation. In *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, page 31, Pittsburgh, Pennsylvania, USA, November 6–12 2004.

[LCR+07] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving Convergence-Free Routing using Failure-Carrying Packets. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*, pages 241–252, Kyoto, Japan, August 27–31 2007.

[Lep05] Jay Lepreau. Personal communications, August 2005. Flux Group, School of Computing, University of Utah.

[Leu04] James R. Leu. Linux virtual routing and forwarding. http://linux-vrf.sourceforge.net, 2004.

[LNT02] Henrik Lundgren, Erik Nordström, and Christian Tschudin. Coping with Communication Gray Zones in IEEE 802.11b based Ad hoc Networks. In *Proceedings of the 5th ACM International Workshop on Wireless Mobile Multimedia (WoWMoM'02)*, pages 49–55, Atlanta, Georgia, USA, September 28 2002.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*, pages 291–300, La Jolla, CA, USA, June 18–21 1995.

[LSW03] Volker Lindenstruth, Timm M. Steinbeck, and Arne Wiebalck. Prozessor-Schwinger – Überwachung der CPU-Auslastung im Kernel. *Linux-Magazin*, pages 44–47, May 2003.

[Mai02] Steffen Maier. Emulationskonzepte für Netze mit gemeinsamem Medium. Diplomarbeit Nr. 2000, Universität Stuttgart, Institute of Parallel and Distributed Systems, Distributed Systems Department, April–October 2002.

[McC95] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee*

on Computer Architecture (TCCA) Newsletter, pages 19–25, December 1995.

[MGWR07] Steffen Maier, Andreas Grau, Harald Weinschrott, and Kurt Rothermel. Scalable Network Emulation: A Comparison of Virtual Routing and Virtual Machines. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC'07)*, pages 395–402, Aveiro, Portugal, July 1–4 2007.

[MHR05] Steffen Maier, Daniel Herrscher, and Kurt Rothermel. On Node Virtualization for Scalable Network Emulation. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*, pages 917–928, Philadelphia, PA, USA, July 24–28 2005.

[MHR07] Steffen Maier, Daniel Herrscher, and Kurt Rothermel. Experiences with node virtualization for scalable network emulation. *Computer Communications*, 30(5):943–956, March 8 2007.

[MI04] Daniel Mahrenholz and Svilen Ivanov. Real-time network emulation with ns-2. In *Proceedings of the 8-th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT'04)*, pages 29–36, Budapest, Hungary, October 21–23 2004.

[MI05] Daniel Mahrenholz and Svilen Ivanov. Adjusting the ns-2 Emulation Mode to a Live Network. In *Proceedings of Kommunikation in Verteilten Systemen 2005 (KiVS 2005)*, Kaiserslautern, Germany, February 28 – March 3 2005.

[Mil04] Robin T. Miller. Data Test Program (dt). http://www.scsifaq.org/RMiller_Tools/dt.html, visited August 2004.

[Moo01] Richard Moore. A Universal Dynamic Trace for Linux and other Operating Systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, MA, USA, June 25–30 2001.

[MR96] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, USA, January 22–26 1996.

[MRBV04] Priya Mahadevan, Adolfo Rodriguez, David Becker, and Amin Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and

Wireless Networks. Technical Report CS2004-0792, Department of Computer Science, University of California, San Diego, June 14 2004.

[MRBV05] Priya Mahadevan, Adolfo Rodriguez, David Becker, and Amin Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. In *Proceedings of the International Workshop on Wireless Traffic Measurements and Modeling (WiTMeMo '05)*, pages 7–12, Seattle, WA, USA, June 5 2005.

[MRBV06] Priya Mahadevan, Adolfo Rodriguez, David Becker, and Amin Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 10(2):26–37, April 2006.

[MS70] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[MYV02] Priya Mahadevan, Ken Yocum, and Amin Vahdat. Emulating Large-Scale Wireless Networks using ModelNet. In *Proceedings of Mobicom'02, Poster Session*, pages 62–64, January 2002.

[NSL+06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel(R) Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(03):167–177, August 10 2006.

[PCE+05] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Bred Chen. Locating System Problems Using Dynamic Instrumentation. In *Proceedings of the Linux Symposium (OLS 2005)*, volume two, pages 49–64, Ottawa, Ontario, Canada, July 20–23 2005.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM (CACM)*, 17(7):412–421, July 1974.

[PHM06] Himabindu Pucha, Y. Charlie Hu, and Z. Morley Mao. On the Impact of Research Network Based Testbeds on Wide-area Experiments. In *Proceedings of the 6th ACM Internet Measurement Conference (IMC'06)*, pages 133–146, Rio de Janeiro, Brazil, October 25–27 2006.

[Pom99] Ori Pomerantz. Linux Kernel Module Programming Guide, 26 April 1999. Version 1.1.0, Linux Documentation Project.

[PPM01] P. Phaal, S. Panchen, and N. McKee. *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176*, September 2001.

[PR99] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc On-Demand Distance Vector Routing. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications (WMCSA'99)*, pages 90–100, New Orleans, Louisiana, February 25–26 1999.

[PT04] Daniel Price and Andrew Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*, pages 241–254, Atlanta, GA, USA, November 14–19 2004.

[Rap01] Theodore Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2001.

[RC01] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, 2nd edition, June 2001.

[RFA99] George F. Riley, Richard M. Fujimoto, and Mostafa H. Ammar. A Generic Framework for Parallelization of Network Simulations. In *Proceedings of the 7th International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS'99)*, pages 128–135, College Park, Maryland, March 24–28 1999. PDNS notes online: `http://www.cc.gatech.edu/computing/compass/pdns/index.html`.

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, Denver, Colorado, USA, August 14–17 2000.

[Ril03] George Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools '03)*, pages 5–12, Karlsruhe, Germany, August 25 2003.

[RKM+05] Kishore Ramachandran, Sanjit Kaul, Suhas Mathur, Marco Gruteser, and Ivan Seskar. Towards Large-Scale Mobile Network Emulation Through Spatial Switching on a Wireless Grid. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Experimental Approaches to Wireless Network Design and Analysis (E-WIND '05)*, pages 46–51, Philadelphia, PA, USA, August 22–26 2005.

[RMK+96] Yakov Rekhter, Robert G Moskowitz, Daniel Karrenberg, Geert Jan de Groot, and Eliot Lear. *Address Allocation for Private Internets. RFC 1918*, February 1996.

[RSO+05] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols. In *Proceedings of the Wireless Communications and Networking Conference (WCNC 2005)*, volume 3, pages 1664–1669, March 13–17 2005.

[Rub00] Alessandro Rubini. Gearheads Only: Virtual Network Interfaces. *Linux Magazine*, April 2000.

[Rus99] David A. Rusling. The Linux Kernel, 25 January 1999. Review, Version 0.8-3, Linux Documentation Project.

[Rus08] Rusty Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, July 2008.

[SB96] Claudia Schmidt and Roland Bless. QoS Monitoring in High Performance Environments. In *Proceedings of the 4th International IFIP Workshop on Quality of Service (IWQOS'96)*, pages 199–207, Paris, France, March 6–8 1996.

[SB02] Volkmar Sieh and Kerstin Buchacker. UMLinux — A Versatile SWIFI Tool. In *Proceedings of the Fourth European Dependable Computing Conference*, volume LNCS 2485, pages 159–171, Toulouse, France, October 23–25 2002. Springer-Verlag.

[SBBD03] Sagar Sanghani, Timothy X Brown, Shweta Bhandare, and Sheetalkumar Doshi. EWANT: The Emulated Wireless Ad Hoc Network Testbed. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2003)*, New Orleans, LA, USA, March 16–20 2003.

[Sch07] Gregor Alexander Schiele. *System Support for Spontaneous Pervasive Computing Environments*. PhD thesis, University of Stuttgart, Institute of Parallel and Distributed Systems, October 4 2007.

[SD04] Ananth I. Sundararaj and Peter A. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. In *Proceedings of the third USENIX Virtual Machine Research and Technology Symposium (VM '04)*, pages 177–190, San Jose, CA, USA, May 6–7 2004.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 196–205, Orlando, FL, USA, June 20–24 1994.

[SFR04] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming, Volume 1: The Sockets Networking API*. Addison-Wesley, 3rd edition, 2004.

[SGD04] Ananth I. Sundararaj, Ashish Gupta, and Peter A. Dinda. Dynamic Topology Adaptation of Virtual Networks of Virtual Machines. In *Proceedings of the Seventh Workshop on Languages, Compilers and Run-time Support for Scalable Systems (LCR 2004)*, October 2004.

[SHR05] Illya Stepanov, Daniel Herrscher, and Kurt Rothermel. On the impact of radio propagation models on manet simulation results. In *Proceedings of the Seventh IFIP International Conference on Mobile and Wireless Communication Networks (MWCN 2005)*, Marrakech, Morocco, September 19–21 2005.

[SKKK03] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. *Linux Netlink as an IP Services Protocol. RFC 3549*, July 2003.

[SKS00] Mark Stemm, Randy H. Katz, and Srinivasan Seshan. A Network Measurement Architecture for Adaptive Applications. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, volume 1, pages 285–294, Tel Aviv, Israel, March 26–30 2000.

[Smo00] Miquel van Smoorenburg. Linux serial console. Source Code Documentation, Linux Kernel 2.4.18, 2000.

[SN05] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*, chapter 1 Introduction to Virtual Machines. Morgan Kaufmann, July 12 2005.

[SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference (ALS 2001)*, pages 165–172, Oakland, CA, USA, November 5–11 2001.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In

Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07), pages 275–287, Lisbon, Portugal, March 21–23 2007.

[SR02] Riccardo Scandariato and Fulvio Risso. Advanced VPN support on FreeBSD systems. In *Proceedings of the 2nd European BSD Conference (BSDCon Europe 2002)*, Amsterdam, the Netherlands, November 15–17 2002.

[SSK97] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 137–146, Monterey, CA, USA, December 8–11 1997.

[Ste09] Illya Stepanov. *Using Geographic Models in the Simulation of Mobile Communication*. PhD thesis, University of Stuttgart, Institute of Parallel und Distributed Systems, January 13 2009.

[Str05] Geoffrey Strongin. Trusted computing using AMD "Pacifica" and "Presidio" secure virtual machine technology. *Information Security Technical Report*, 10(2):120–132, January 2005.

[SU01] Rob Simmonds and Brian Unger. Towards Scalable Network Emulation. In *Proceedings of SPIE: Scalability and Traffic Control in IP Networks*, volume 4526, pages 252–262, Denver, Colorado, USA, August 22–24 2001.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 1–14, Boston, Massachusetts, USA, June 25–30 2001.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[TM99] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*

(OSDI 99), pages 117–130, New Orleans, LA, USA, February 22–25 1999.

[TMB01] Mineo Takai, Jay Martin, and Rajive Bagrodia. Effects of Wireless Physical Layer Modeling in Mobile Ad Hoc Networks. In *Proceedings of the 2nd ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc'01)*, pages 87–94, Long Beach, CA, USA, October 4–5 2001.

[TW+01] Linus Torvalds, G. W. Wettstein, et al. Oops tracing. Source Code Documentation, Linux Kernel 2.4.18, 2001.

[VBV+05] Nitin H. Vaidya, Jennifer Bernhard, V. V. Veeravalli, P. R. Kumar, and R. K. Iyer. Illinois Wireless Wind Tunnel: A Testbed for Experimental Evaluation of Wireless Networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Experimental Approaches to Wireless Network Design and Analysis (E-WIND '05)*, pages 64–69, Philadelphia, PA, USA, August 22–26 2005.

[VYW+02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 9–11 2002.

[Wac08] Arno Rüdiger Wacker. *Key Distribution Schemes for Resource-Constrained Devices in Wireless Sensor Networks*. PhD thesis, University of Stuttgart, Institute of Parallel und Distributed Systems, May 29 2008.

[Wal02] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 181–194, Boston, MA, USA, December 9–11 2002.

[WHL99] G. Wölfle, R. Hoppe, and F. M. Landstorfer. A Fast and Enhanced Ray Optical Propagation Model for Indoor and Urban Scenarios, Based on an Intelligent Preprocessing of the Database. In *Proceedings of the 10th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Osaka, Japan, September 1999.

[WLG02] Brian White, Jay Lepreau, and Shashi Guruprasad. Lowering the Barrier to Wireless and Mobile Experimentation. In *Proceedings of*

the First Workshop on Hot Topics in Networks (Hotnets-I), Princeton, New Jersey, USA, October 28–29 2002.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 255–270, Boston, MA, December 2002.

[WPR+02] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, and Marc Bechler. *Linux Netzwerkarchitektur: Design und Implementierung von Netzwerkprotokollen im Linux-Kern*. Addison-Wesley, 2002.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, USA, December 9–11 2002.

[XHR04] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, volume 4, pages 2514–2524, Hong Kong, March 7–11 2004.

[Yan04] Zhenxiang Yang. Entwicklung eines Verfahrens zur Emulation der Medienzugriffssteuerung in Wireless LAN. Diplomarbeit Nr. 2167, Universität Stuttgart, Abteilung Verteilte Systeme, 2004.

[YED+03] Ken Yocum, Ethan Eade, Julius Degesys, David Becker, Jeff Chase, and Amin Vahdat. Toward Scaling Network Emulation using Topology Partitioning. In *Proceedings of the 11th IEEE / ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2003)*, pages 242–245, Orlando, FL, USA, October 12–15 2003.

[YWV+02] Ken Yocum, Kevin Walsh, Amin Vahdat, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the SIGCOMM 2002 Poster Abstracts*, page 28, Pittsburgh, PA, USA, August 19–23 2002.

[ZBG98] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulations (PADS '98)*, pages 154–161, Banff, Canada, May 1998.

[Zec03] Marko Zec. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 137–150, San Antonio, Texas, June 9–14 2003.

[ZJTB03] Junlan Zhou, Zhengrong Ji, Mineo Takai, and Rajive Bagrodia. Maya: a Multi-Paradigm Network Modeling Framework for Emulating Distributed Applications. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation (PADS'03)*, pages 163–170, San Diego, CA, USA, June 10–13 2003.

[ZJTB04] Junlan Zhou, Zhengrong Ji, Mineo Takai, and Rajive Bagrodia. Maya: Integrating hybrid network modeling to the physical world. *ACM Transactions on Modeling and Computer Simulation*, 14(2):149–169, 2004.

[ZL04] Marcia Zangrilli and Bruce B. Lowekamp. Using Passive Traces of Application Traffic in a Network Monitoring System. In *Proceedings of the Thirteenth IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, pages 77–86, Honolulu, HI, USA, June 4–6 2004.

[ZM03] Marko Zec and Miljenko Mikuc. Real-Time IP Network Simulation at Gigabit Data Rates. In *Proceedings of the 7th International Conference on Telecommunications (ConTEL 2003)*, pages 235–242, Zagreb, Croatia, June 11–13 2003.

[ZM04] Marko Zec and Miljenko Mikuc. Operating System Support for Integrated Network Emulation in IMUNES. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (2004 OASIS)*, pages 3–12, Boston, MA, October 9–13 2004.

[ZN02a] Pei Zheng and Lionel M. Ni. EMPOWER: A Scalable Framework for Network Emulation. In *Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*, pages 185–192, Vancouver, B.C., Canada, August 18–21 2002.

[ZN02b] Pei Zheng and Lionel M. Ni. EMWIN: emulating a mobile wireless network using a wired network. In *Proceedings of the 5th ACM International Workshop on Wireless Mobile Multimedia (WoWMoM '02)*, pages 64–71, Atlanta, Georgia, USA, 28 September 2002.

[ZN02c] Pei Zheng and Lionel M. Ni. Experiences in Building a Scalable Distributed Network Emulation System. In *Proceedings of 2002 International Conference on Parallel and Distributed Systems (ICPADS 2002)*, Taipei, Taiwan, ROC, December 17–20 2002.

[ZN03a] Pei Zheng and Lionel M. Ni. EMPOWER: A Network Emulator for Wireless and Wired Networks. In *Proceedings of the Conference on Computer Communications (INFOCOM 2003)*, volume 3, pages 1933–1942, San Francisco, March 30 – April 3 2003.

[ZN03b] Pei Zheng and Lionel M. Ni. Test and Evaluation of Wide Area Networks Using Emulator Cluster. In *Proceedings of the IEEE International Conference on Communications (ICC'03)*, volume 1, pages 281–285, May 11–15 2003.

[ZN04] Pei Zheng and Lionel M. Ni. EMPOWER: A Cluster Architecture Supporting Network Emulation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):617–629, July 2004.