Consistent Data Replication in Mobile Ad Hoc Networks

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

Jörg Hähner

aus Düsseldorf

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Alejandro P. Buchmann, Ph. D.
Tag der mündlichen Prüfung:	19.10.2006

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2007

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

First of all I would like to thank my advisor Kurt Rothermel for giving me the opportunity to work on this dissertation in his group. I would also like to thank him for his guidance and many fruitful discussions about my work. I would also like to thank Alejandro Buchmann for his work as my co-advisor.

Many thanks go to my exceptional colleagues in the Distributed Systems Group who helped me at countless occasions with valuable suggestions and lots of motivation. In particular I would like to thank Christian Becker, Dominique Dudkowski, Marcus Handte, Pedro José Marrón, and Gregor Schiele for many valuable discussions.

I would also like to thank the Gottlieb Damiler and Karl Benz Foundation in Ladenburg for partially funding my work within the project "Ladenburger Kolleg – Living in a Smart Environment".

Abstract

Mobile ad-hoc networks (MANETs) are used in situations where networks need to be deployed immediately but no network infrastructure is available. If MANET nodes have sensing capabilities, they can capture and communicate the state of their surroundings, including environmental conditions or objects in their proximity. If the sensed state information is propagated to a database to build a consistent model of the real world, a variety of promising context-aware applications becomes possible.

The models and concepts proposed in this dissertation can be applied to cooperatively maintain a model of the state of physical world objects on devices in MANETs. State information may be updated by independent observers either sequentially or concurrently. Applications that read the state of any object from the model multiple times can rely on the guarantee that every successive read operation will read either the same state information or a newer state information that has been reported by an observer after the previously read information. The first contribution of this dissertation formalizes these requirements and defines a novel consistency model called *update-linearizability*. Secondly, it introduces a new class of data replication algorithms that provably guarantees *update-linearizability* in MANETs without using synchronized clocks on any pair of nodes in the system. The presented algorithms allow to execute read and write operations at any time, which provides *high availability* of data. These properties are even maintained in networks that are temporarily partitioned and where nodes are highly mobile. Finally the dissertation provides a proof that all replicas held in the system eventually converge towards the most recent state information of the physical world objects which they represent.

Zusammenfassung

Die schnelle Entwicklung von eingebetteten Systemen, Kommunikations- und Sensortechnologie hat zu vielfältigen Rechnersystemen geführt, mit denen es möglich ist, Eigenschaften der physischen Welt zu überwachen. Je nach Art der verwendeten Sensortechnologie ist es möglich, einfache physikalische Größen, wie beispielsweise die Temperatur, oder auch komplexere Umgebungsinformationen, wie beispielsweise Bilder die von einer Kamera geliefert werden, zu erfassen. Die Reichweite, mit der die eingesetzten Sensoren Änderungen in der physischen Welt erfassen können ist jedoch beschränkt. Eine Möglichkeit um größere Gebiete zu überwachen, ist die Verwendung von mehreren Geräten mit Sensoren an verschiedenen Orten innerhalb des zu beobachtenden Gebietes. Jedes dieser Geräte kann nun lokal die zu messenden Größen erfassen und diese Informationen mit anderen Geräten mittels drahtloser Kommunikation austauschen. Diese verteilt gemessenen Informationen können dann im Folgenden zu einem Modell des gesamten beobachteten Gebietes zusammengefasst und verwaltet werden. Das daraus resultierende Modell der physischen Welt kann dann als Basis für eine Vielzahl von Beobachtungsanwendungen, wie beispielsweise das Planen und Durchführen von Rettungseinsätzen [HRB04] genutzt werden.

In solchen Beobachtungsanwendungen spielt das Konzept der physischen Zeit eine besondere Rolle, da Schlüsse häufig auf der Basis der chronologischen Ordnung, in der Änderungen erfolgen, gezogen werden. Wenn beispielsweise an allen Türen in einem Gebäude Sensoren angebracht sind, mit deren Hilfe festgestellt werden kann, in welche Richtung eine Person sich durch eine Tür bewegt, dann kann eine Anwendung aus der jüngsten Sensorinformation ableiten, in welchem Raum sich die Person befindet. Die Bezeichnung *jüng-*

ste Information bezieht sich dabei direkt auf die chronologische Ordnung der Sensorinformationen. Wenn eine Applikation die Beobachtungen bezüglich der Zeitpunkte ordnen kann, zu denen sie passiert sind, dann kann die Position einer Person korrekt ermittelt werden. Liegt diese Ordnung nicht korrekt vor, dann kommt die Applikation möglicherweise zu falschen Schlüssen.

Neben der korrekten chronologischen Ordnung der Beobachtungen spielt die kooperative Verwaltung von Sensorinformationen auf einer Menge von Geräten eine wichtige Rolle. Diese Geräte können beispielsweise in einem mobilen ad-hoc Netz (MANET), in dem keine dedizierte Kommunikationsinfrastruktur benötigt wird, organisiert sein. In diesem Fall kann es jedoch vorkommen, dass Änderungen des gleichen Objektes von verschiedenen voneinander unabhängigen Geräten erfasst werden. Mit dem Begriff Datenkonsistenz wird allgemein die Widerspruchsfreiheit einer Datenmenge bezeichnet. Bezogen auf die von verschiedenen Geräten erfassten Daten können Widersprüche beispielsweise auftreten, wenn mehrere Kopien desselben Datenobjekts auf verschiedenen Geräten gespeichert werden. Aufgrund der Eigenschaften von MANETs können so genannte strikte Konsistenzanforderungen aus dem Bereich der klassischen Datenbanksysteme nicht durchgesetzt werden. Stattdessen werden in der Regel so genannte schwache Konsistenzmodelle verwendet, um die Widerspruchsfreiheit der Daten zu beschreiben. Hierbei wird durch die verwendeten Verfahren garantiert, dass alle Kopien eines Datenobjektes in einen gemeinsamen Zustand konvergieren. Die Forderung, dass die Kopien innerhalb des Netzes in irgendeinen gemeinsamen Zustand konvergieren, reicht jedoch oft nicht aus. Wenn beispielsweise die Folge von Schreiboperationen auf einem Objekt dazu dient, gemessene Temperaturänderungen an einer Brücke zu speichern, dann ist es wichtig, dass diese Änderungen auch in der Reihenfolge gespeichert werden, in der sie tatsächlich aufgetreten sind, d.h. in chronologischer Ordnung. Wird bei der Ausführung der Operationen die chronologische Ordnung eingehalten, dann kann durch das Lesen des Datenobjekts zum Beispiel festgestellt werden, ob die Temperatur gestiegen oder gefallen ist. Werden die Änderungen der Temperatur nicht in chronologischer Ordnung gespeichert, so ist es nicht möglich diese Schlüsse korrekt zu ziehen. Als nächstes wird ein Konsistenzmodell vorgestellt, welches die chronologische Reihenfolge von Aktualisierungen berück-

sichtigt. Der dann vorgestellte Replikationsalgorithmus garantiert, dass das beschriebene Modell der Datenkonsistenz eingehalten wird.

Konsistenzmodell

In dem im Folgenden vorgestellten Konsistenzmodell [HRB04] können die beteiligten Knoten eine oder mehrere der folgenden Rollen annehmen: Beobachter, Datenbankknoten und Klienten. Die Aufgabe von Beobachtern ist es, den Zustand von Objekten der physischen Welt (kurz Objekte) mittels geeigneter Sensorik zu erfassen. Dazu müssen die Objekte eindeutig identifizierbar sein. Dies kann beispielsweise durch die Verwendung von elektronischen Etiketten (RFID-Tags) wie etwa dem elektronischen Produktcode [epc] realisiert werden. Wenn ein Beobachter eine Zustandsänderung eines Objektes erfasst, dann erzeugt er eine so genannte Aktualisierungsanforderung (*update request*), welche die Identität und den neuen Zustand des entsprechenden Objektes enthält. Diese Aktualisierungsanforderung wird dann vom Beobachter an alle in der Nähe befindlichen Datenbankknoten versandt. Die Datenbankknoten (kurz DB-Knoten) sind im Folgenden dafür verantwortlich, den jeweils aktuellsten Zustand von jedem Objekt im Sinne der chronologischen Ordnung zu verwalten. Zur Beantwortung von Anfragen halten sie eine Kopie des Zustands von jedem Objekt. Somit können Anfragen von jedem DB-Knoten direkt beantwortet werden. Klienten sind Anwendungen, die ausschließlich Leseoperationen auf den erfassten Zustandsinformationen ausführen.

Da es in verteilten Systemen keine exakte globale Zeit gibt, können Beobachtungen verschiedener Beobachter nur mit begrenzter Genauigkeit geordnet werden. Aus diesem Grund wird die so genannte *geschieht-vor* Relation (*occurred-before*) definiert, um Aussagen über die chronologische Ordnung von Operationen machen zu können:

Definition 1 (*geschieht-vor*):
Seien u und u' zwei Aktualisierungsanforderungen. Definitionsgemäß gilt: u *geschieht-vor* u', genau dann wenn $t_{\text{obs}}(u') - t_{\text{obs}}(u) > \delta$ mit $\delta > 0$. Die Zeit $t_{\text{obs}}(u)$ bezeichnet dabei den Zeitpunkt, zu dem die Beobachtung gemacht

wurde, welche in u beschrieben ist.

In dieser Definition ist der Parameter δ ein Systemparameter, der die mögliche Genauigkeit der chronologischen Ordnung von zwei Aktualisierungsanforderungen angibt. Je kleiner δ ist, umso genauer ist die *geschieht-vor* Relation, da nur solche Aktualisierungsanforderungen geordnet werden können, die mindestens δ Zeit auseinander liegen. Wenn zwei Aktualisierungsanforderungen weniger als δ auseinander liegen, dann spricht man von nebenläufigen Anforderungen. Diese Nebenläufigkeit wird als $u \mid\mid u'$ notiert. Das in dieser Arbeit vorgestellte Konsistenzmodell *Update-Linearisierbarkeit* ordnet die Aktualisierungsanforderungen aller Beobachter auf der Basis der *geschieht-vor* Relation. Informell ausgedrückt wird garantiert, dass ein Klient, der einmal den Zustand eines Objektes gelesen hat, bei einer folgenden Leseoperation entweder denselben Zustand liest oder einen aktuelleren im Sinne der *geschieht-vor* Relation. Die formale Definition der Update-Linearisierbarkeit beruht auf der Existenz so genannter Serialisierungen der verteilten Ausführung von Aktualisierungsanforderungen und Leseoperationen und ist wie folgt definiert.

Definition 2 (Update-Linearisierbarkeit):
Eine Ausführung von Leseoperationen und Aktualisierungsanforderungen ist konsistent, wenn es eine Serialisierung S dieser Ausführung gibt, die folgende Bedingungen erfüllt:
(C1) Alle Leseoperationen eines einzelnen Klienten auf einem einzelnen Objekt sind in S gemäß der Programmordnung des Klienten geordnet.
(C2) Für jedes Objekt x und jedes Paar von Aktualisierungsanforderungen $u[x]$ und $u'[x]$ auf x in S gilt: $u'[x]$ ist ein (direkter oder indirekter) Nachfolger von $u[x]$ in S, wenn $u[x]$ *geschieht-vor* $u'[x]$ oder $u[x] \mid\mid u'[x]$ gilt.
(C3) Für jedes Objekt x in der Datenbank hält S die Spezifikation einer einzigen Kopie von x ein.

Die Update-Linearisierbarkeit sichert zu, dass Klienten, die Kopien eines Objektes lesen, die Sicht auf ein einzelnes Objekt erhalten. Die Definition verlangt, dass auf dieser logischen Kopie der Datenbank alle Aktualisierungsanforderungen gemäß der *geschieht-vor* Relation ausgeführt werden. Basierend auf dem vorgestellten formalen Konsistenzmodell beschreiben wir im Fol-

genden nun einen Replikationsalgorithmus, welcher die die Einhaltung der Update-Linearisierbarkeit garantiert. Innerhalb der gesamten Arbeit wird ein weiterer Algorithmus vorgestellt, der die Informationen auf einer Teilmenge aller im Netz befindlichen Knoten repliziert.

Algorithmen und Datenstrikturen

Der im folgenden beschriebene Replikationsalgorithmus [HRB04] dient dazu, den Zustand von jedem beobachteten Objekt auf allen DB-Knoten zu replizieren. Das zugrunde liegende Rechnernetz besteht aus Knoten, die mobil sein können und die Rollen von Beobachtern, DB-Knoten oder Klienten einnehmen. Jeder Knoten, der die Rolle eines Klienten inne hat, nimmt automatisch auch die Rolle eines DB-Knoten wahr. Damit ist es einerseits möglich, dass Klienten ihre Leseoperationen lokal, d.h. ohne Kommunikation über das Netz, ausführen können. Auf der anderen Seite müssen Aktualisierungen an alle DB-Knoten verteilt werden. Die Rolle des Beobachters kann von einem Knoten entweder exklusiv wahrgenommen werden oder mit der Rolle des DB-Knoten kombiniert werden. Der Replikationsalgorithmus ordnet Aktualisierungsanforderungen basierend auf den Zeitpunkten zu denen sie von DB-Knoten in direkter Kommunikationsreichweite der erzeugenden Beobachter empfangen werden. Daher ist der Wert des Parameters aus der Definition der *geschieht-vor* Relation von der maximalen Schwankung (Jitter) der Kommunikationsverzögerung auf einer einzigen Teilstrecke (single-hop) im Netz abhängig. Wird für den Parameter δ die maximalen Schwankung auf einer Teilstrecke gewählt (dargestellt in Abbildung1), so kann anhand der Empfangszeitpunkte die Ordnung der Aktualisierungsanforderungen nach der *geschieht-vor* Relation bestimmt werden. Knoten, welche die Rolle eines DB-Knoten innehaben, benötigen physische Uhren, die jedoch nur der lokalen Zeitmessung dienen und deshalb nicht untereinander synchronisiert sein müssen.

Der Zustand einzelner Objekte wird von Beobachtern in Aktualisierungsanforderungen abgespeichert. Eine Aktualisierungsanforderung ist ein 4-Tupel $(Obj, State, Obs, SN)$ und enthält Informationen über ein Objekt mit der Identität Obj. Der gespeicherte Zustand des Objektes wird als $State$ be-

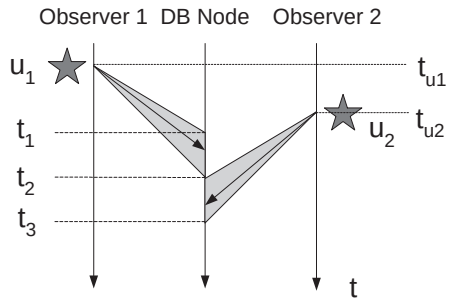


Abbildung 1: Zwei Aktualisierungsanforderungen mit max. Jitter (grau schattiert)

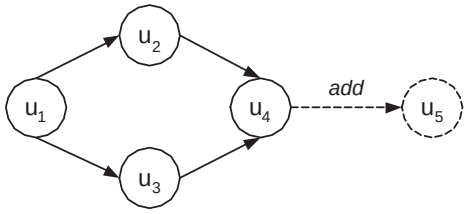


Abbildung 2: Beispiel eines Ordnungsgraphen

zeichnet und wurde von dem Beobachter *Obs* erfasst und versandt. Zu jeder Aktualisierungsanforderung vergibt der Beobachter eine streng monoton wachsende Sequenznummer *SN*. Damit kann die Ordnung von Aktualisierungsanforderungen desselben Beobachters gemäß der *geschieht-vor* Relation anhand der jeweiligen Sequenznummern festgestellt werden. Um die Relation zwischen Aktualisierungsanforderungen von verschiedenen Beobachtern ableiten zu können, müssen zusätzliche Ordnungsinformationen erfasst und verwaltet werden. Dazu wird ein so genannter Ordnungsgraph verwendet. Jeder Knoten in diesem Graph stellt eine lokal bekannte Aktualisierungsanforderung dar. Die gerichteten Kanten zwischen den Knoten repräsentieren Elemente der *geschieht-vor* Relation. Wenn eine Kante (u, u') zwischen zwei Aktualisierungsanforderungen u und u' im Graphen existiert, dann heißt das, dass u vor u' in der *geschieht-vor* Relation liegt.

Abbildung 2 zeigt ein Beispiel eines Ordnungsgraphen, in dem zunächst die Aktualisierungsnachrichten u_1 bis u_4 enthalten sind. Aus diesem Graph lässt sich ablesen, dass u_1 das kleinste Element gemäß der *geschieht-vor* Re-

lation in diesem Graphen ist. Das heißt, bei u_1 handelt es sich in der chronologischen Ordnung um die älteste bekannte Aktualisierungsanforderung. Die Aktualisierungen u_2 und u_3 sind in der chronologischen Ordnung jünger als u_1, können aber untereinander nicht verglichen werden. Die Anforderung u4 ist gemäß der dargestellten Ordnung jünger als u_2 und u_3 und damit auch jünger als u_1. Zusätzlich zum Ordnungsgraphen sind die Operationen *add* und *join* definiert, die es ermöglichen, Ordnungsgraphen zu verarbeiten und aus ihnen Schlüsse über die *geschieht-vor* Relation zwischen Aktualisierungsanforderungen unterschiedlicher Beobachter zu ziehen. Die *add* Operation dient dazu, eine neue Aktualisierungsanforderung in den Graphen einzuordnen, d.h. einen neuen Knoten und Kanten in einen bestehenden Graphen einzufügen. Damit die so erzeugte Ordnungsinformation konsistent mit der Beobachtungsreihenfolge in der physischen Welt ist, wird die add Operation nur angewandt, wenn die beiden folgenden Bedingungen erfüllt sind. Basierend auf der Diskussion des Parameters δ, muss die Aktualisierungsanforderung auf einem DB-Knoten direkt von einem Beobachter empfangen werden. Zusätzlich darf in der Zeitspanne der Länge δ vor dem Empfang der Aktualisierungsnachricht keiner weiten Aktualisierungsnachricht von einem anderen Beobachter für dasselbe Objekt empfangen worden sein. In Abbildung 2 wird die Anwendung der *add* Operation anhand der Aktualisierungsanforderung u_5 veranschaulicht. Die *join* Operation dient dazu, Ordnungsinformationen, die von unterschiedlichen DB-Knoten erfasst wurden, konsistent zu vereinigen. Dazu werden die Knoten- und die Kantenmengen der beiden Graphen vereinigt. Sowohl die *add* als auch die *join* Operation enthalten darüber hinaus Optimierungen, welche die Größe der Graphen reduzieren. Die Reduktion garantiert, dass ein Graph maximal einen Knoten pro Beobachter — den jeweils aktuellsten — enthält. Der Replikationsalgorithmus, der im Folgenden vorgestellt wird, garantiert zu jeder Zeit die Einhaltung der Update-Linearisierbarkeit. Ein DB-Knoten, der eine Aktualisierungsanforderung für ein Objekt x direkt von einem Beobachter empfängt, überprüft zunächst die oben beschriebenen Voraussetzungen für die Anwendung der *add* Operation. Wenn möglich, wird die *add* Operation auf den lokal gespeicherten Ordnungsgraphen für das Objekt x angewandt und der bisher gespeicherte Zustand des Objektes mit dem neu empfangenen Zustand

ersetzt. Nachdem dies durchgeführt wurde, wird die Aktualisierungsanforderung zusammen mit einer Kopie des zugehörigen lokalen Ordnungsgraphen an alle direkt benachbarten DB-Knoten verschickt. Empfängt ein DB-Knoten eine Aktualisierungsanforderung für ein Objekt x zusammen mit einem Ordnungsgraphen wird im Algorithmus zunächst das Resultat der *join* Operation — angewandt auf den empfangenen und den lokal gespeicherten Graphen — gespeichert. Damit erhält der DB-Knoten neue Ordnungsinformationen. Als nächstes muss der Algorithmus entscheiden, ob die empfangene Aktualisierungsanforderung gespeichert, und damit der bisher gespeicherte Zustand des Objekts x überschrieben werden soll. Anders formuliert muss nun überprüft werden, ob der empfangene Objektzustand — enthalten in der Aktualisierungsanforderung — aktueller ist als der bisher gespeicherte. Dazu müssen im Algorithmus drei Fälle unterschieden werden. Wenn das Objekt x bisher unbekannt war, d.h. es wurde bisher keine Aktualisierungsanforderung für x empfangen, dann kann der Zustand angenommen werden. In dem Fall, in dem sowohl der momentan gespeicherte Zustand als auch der empfangene Zustand von x von dem selben Beobachter erfasst wurde, kann über einen Vergleich der Sequenznummern in den dazu gehörigen Aktualisierungsanforderungen entschieden werden, welcher von beiden Zuständen aktueller ist. Im dritten Fall, wenn die empfangene und die bereits gespeicherte Aktualisierungsanforderung von unterschiedlichen Beobachtern stammen, wird der Ordnungsgraph von x verwendet, um über die Speicherung der empfangenen Informationen zu entscheiden.

Leistungsbewertung

Im Folgenden werden ausgewählte Ergebnisse von Simulationsexperimenten präsentiert, welche die Leistungsfähigkeit des Replikationsalgorithmus veranschaulichen. Hierzu sollen insbesondere zwei wichtige Metriken, die Aktualisierungslatenz und die so genannte Lückengröße zwischen zwei auf einer physischen Kopie durchgeführten Aktualisierungen, betrachtet werden. Die Aktualisierungslatenz beschreibt die Zeit, die zwischen dem Versenden einer Aktualisierungsanforderung durch einen Beobachter und dem Empfang derselben Anforderung auf einem DB-Knoten vergeht. Dabei werden nur sol-

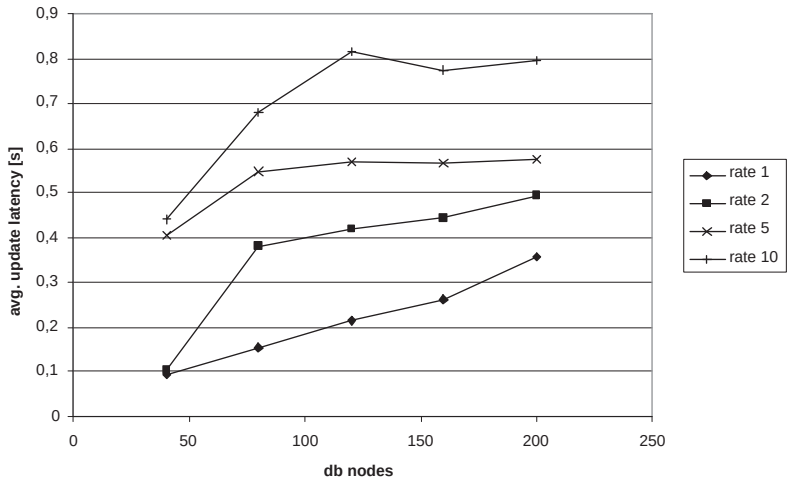


Abbildung 3: Durchschnittliche Latenz bei der Aktualisierung über der Anzahl der DB-Knoten

che Anforderungen betrachtet, die tatsächlich zur Ausführung kommen, d.h. in diesen Fällen kann eindeutig entschieden werden, dass die empfangene Anforderung einen aktuelleren Zustand enthält als der momentan auf dem DB-Knoten gespeicherte Zustand. Zur Laufzeit kann es zu Situationen kommen, in denen Nachrichten nicht bei allen DB-Knoten ankommen. Dieser Fall tritt beispielsweise ein, wenn das Netz durch die Bewegung mobiler Knoten zeitweise partitioniert wird. So kann es dazu kommen, dass einem DB-Knoten eine Aktualisierungsanforderung nicht ausführt, obwohl der darin enthaltene Zustand aktueller im Sinne der *geschieht-vor* Relation ist als der momentan gespeicherte. In diesen Fällen entsteht eine so genannte Lücke, d.h. der DB-Knoten hätte mit vollständigem Wissen einen aktuelleren Zustand gespeichert. Die Lückengröße bezeichnet wie viele solcher Aktualisierungsanforderungen hintereinander auf einem DB-Knoten nicht zur Ausführung kamen und ist damit eine Metrik für die Aktualität der auf DB-Knoten gespeicherten Informationen.

Die Messungen der beschriebenen Metriken wurden im Simulator ns2 [ns2] durchgefhrt. In den durchgeführten Simulationen wird eine Fläche von 500 x

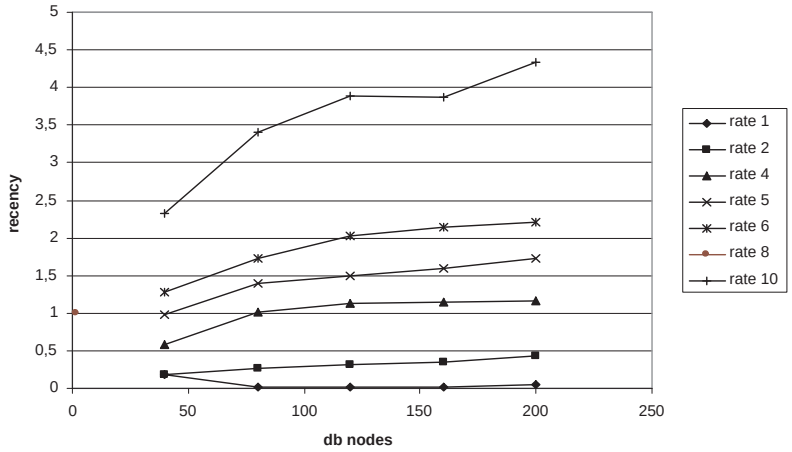


Abbildung 4: Durchschnittliche Lückengröße über der Anzahl der DB-Knoten.

500 m^2 zugrunde gelegt, auf der insgesamt 100 Beobachter platziert sind. Als Sendereichweite wird 100 m und als Datenübertragungsrate wird 1 Mbit/s angenommen. Die Anzahl der DB-Knoten wird in verschiedenen Experimenten zwischen 40 und 200 variiert. Diese Knoten bewegten sich nach dem so genannten *random waypoint* Mobilitätsmodell [BMJ+98] mit einer Geschwindigkeit zwischen 1 und 2 m/s. Die Rate, mit der Aktualisierungsanforderungen erzeugt werden, variiert zwischen einer und zehn Aktualisierungen pro Sekunde (systemweit).

Beispielhaft zeigt Abbildung 3 Ergebnisse für die durchschnittliche Aktualisierungslatenz über der Anzahl der DB Knoten. Jede einzelne Kurve stellt dabei die Ergebnisse für verschiedene Aktualisierungsraten dar. Bei großen Aktualisierungsraten lässt sich ein starker Anstieg der Latenz, bedingt durch die steigende Netzlast, beobachten. In Abbildung 4 ist beispielhaft die durchschnittliche Lückengröße in den entsprechenden Szenarien dargestellt. Mit steigender Aktualisierungsrate und damit steigender Netzlast lässt sich beobachten, dass die Größe der Lücken zwischen Paaren von akzeptierten Aktualisierungen wächst.

Zusammenfassung

Mit den Methoden und Verfahren, die in dieser Arbeit vorgestellt werden, lassen sich Modelle der physischen Welt auf den Knoten von mobilen ad-hoc Netzen verwalten. Das vorgestellte Konsistenzmodell berücksichtigt dabei die chronologische Ordnung zwischen den verschiedenen Aktualisierungen eines beobachteten Objektes. Insbesondere wird garantiert, dass eine Leseoperation niemals einen Wert als Ergebnis liefert, der älter als das Ergebnis einer früheren Leseoperation auf dem selben Objekt ist. Die präsentierten Algorithmen verwenden zum Ableiten der chronologischen Ordnung keine synchronisierten Uhren und garantieren das Konsistenzmodell. Die hierzu notwendige Annahme erfordert, dass die Schwankung (*jitter*) der Kommunikationsverzögerung auf einer einzelnen Teilstrecke im Netz nach oben beschränkt ist.

Die Leistungsbewertung zeigt, dass die Algorithmen sich in einer Vielzahl von Szenarien anwenden lassen. In Szenarien, bei denen die Netzlast insgesamt niedrig ist, ist die Aktualität der gelesenen Informationen nahe an den global aktuellsten Werten. Bei höherer Netzlast sinkt die Aktualität der gelesenen Informationen ab. Es ist jedoch erkennbar, dass die Datenobjekte selbst bei hoher Last regelmäßig aktualisiert werden.

Basierend auf den vorgestellten Konzepten und Algorithmen lassen sich Systeme entwickeln, die

- den Zustand der physischen Welt erfassen und speichern,
- die chronologische Ordnung zwischen Zustandswechseln von Objekten erfassen,
- keine Netzinfrastruktur benötigen und
- keine synchronisierten Uhren benötigen.

Anwendungen in solchen Systemen können die verwalteten Informationen dazu verwenden, um auf Änderungen in der physischen Welt zu reagieren und ihre Funktionalität darauf hin anzupassen.

Chapter 1

Introduction

1.1 Motivation

The rapid development of embedded systems, (wireless) communication and sensor technologies has lead to a manifold class of small computing devices which can be used to unobtrusively monitor properties of their physical surroundings. Depending on the sensor technology the properties that may be monitored range from basic physical quantities in the surroundings, e.g. temperature, to complex environmental information, for example, images captured by a camera. However, all sensor technologies have in common that their sensing range is limited. One solution to enable the monitoring of spatial areas which are larger than the sensing range of an individual sensor is to network multiple devices with sensors at different locations. Each individual device can then monitor local properties and exchange this information with other devices in order to create a model of the physical surroundings from the union of all sensors in the system. This model of the physical surroundings is an important basis for different classes of applications that need to respond to changes in the physical world. It is, for example, important in a rescue scenario [HRB04] where a group of fire fighters needs to cooperatively extinguish a fire in a large spatial area. Other application examples include tracking the position of people or objects, e.g., [WJH97, MHR01], adaptive home and office automation, e.g., [HL01], the structural surveil-

lance of buildings in order to automatically detect damage, e.g., [MLM+05], or driver assistance systems based on information gathered in networks that span multiple vehicles, e.g., [MLM+05].

In such monitoring applications, the concept of *physical time* is fundamental for reasoning about the coherence of changes to objects in the physical world. Given that, for example, all doors in a building are equipped with sensors that detect individuals passing through. If these sensors also detect the direction in which people walk through a door, the collected information can be used to retrieve the position of persons in the building by looking at which sensor reported most recently about them. The question which sensor reported "most recently" directly refers to the order of observations according to physical time. If an application is able to order the observations of sensors according to the time at which they occurred, it can correctly derive the position of a person. If, however, that order is not preserved in the system, the application may come to a false conclusion.

Besides the correct ordering, the *cooperative management of information* across multiple devices is important since many applications need to reason about the situation in their complete operational environment and not only in their vicinity which may be monitored with local sensors. Consider, for example, the group of fire fighters in the rescue mission who have to coordinate their individual tasks in order to achieve the superior goal of fighting a forest fire. Sub-tasks of the workers include securing a part of the operational area. Here, they benefit from information describing the temperature at different locations in the sub-area they are responsible for. Additionally, they need to monitor the state of equipment, such as water pumps, in order to react to failure of equipment. Beyond the individual sub-tasks, groups of workers need information about the state of equipment and coworkers in surrounding areas. This is essential to react promptly, if the situation in other parts of the operational environment becomes critical, e.g., when the direction of the wind changes in the case of a forest fire and a close-by coworker needs support. To be able to access the necessary information, it needs to be collected by and shared between the workers. In that sense, the fire fighters and their equipment may be augmented by appropriate sensors, computing

devices, and communication technology that allow to collect the information at face and to distribute it among the workers giving each one the necessary information to reason about what to do.

One important communication paradigm for networking devices in such future systems is the paradigm of *mobile ad-hoc networks* (MANETs) [Sto02] where devices can communicate spontaneously with each other without using a pre-installed communication infrastructure such as a mobile phone network like GSM or UMTS. Using infrastructure-less systems like MANETs bears two major advantages. First, MANETs may operate in locations where no communication infrastructure is available or has been destroyed. Secondly, they support the management of information that has local relevance [BBH02] such as information from nearby co-workers in the rescue scenario by design since close-by devices exchange information directly. Additionally, information can be distributed over larger distances than the communication range of a single device using multi-hop communication.

In summary it can be concluded that data models which contain information about observations made in the physical world will provide an important basis for applications in order to adapt their behavior to changes of the model and thus to changes in the physical world. Due to the limited sensing range of individual sensors multiple devices will be used to make observations in larger spatial areas. Using MANETs to network these devices suits the requirement of many monitoring applications well, since the monitoring area of interest can easily be covered with a MANET even if a pre-installed networking infrastructure is not available.

1.2 Problem Statement and Contributions

From the previous scenario it can be derived that it is necessary to *consistently* manage the information about the state of physical world objects. In particular this involves to maintain the order of changes according to physical time which becomes particularly difficult, if the observations are sensed by *independent devices*. Therefore it is required to develop appropriate mechanisms to order physical world observations in a distributed system with

respect to physical time. In the case of independent devices these observations are causally unrelated since the observations occur outside the system. In the described rescue scenario this happens, for example, if the temperature at a certain location is first measured by one fire fighter and then later on by one of his co-workers. Both measure the temperature at the same location. However, the reportings of the temperature values need to be ordered according to physical time to derive whether there was an increase or decrease of temperature at the observed location.

The results of this dissertation can be used to build and maintain a model of the physical world which describes the state of (real) objects in a given spatial area on the devices in a MANET. The objects are monitored by mobile devices equipped with sensors (so-called observers) which are located in the spatial area. The state information of objects is maintained cooperatively on mobile devices which assemble a MANET. State information may be updated by independent observers either sequentially or concurrently. Applications that read the state of any object from the model multiple times can rely on the guarantee that every successive read operation will read either the same or a more recent state information that has been reported by an observer after the previously read information. The first contribution of this dissertation formalizes these requirements and defines a novel consistency model called *update-linearizability*. Secondly, it introduces a new class of data replication algorithms that provably guarantees *update-linearizability* in MANETs without using synchronized clocks on any pair of nodes in the system. The presented algorithms allow to execute read and write operations at any time, which provides *high availability* of data. These properties are even maintained in networks that are temporarily partitioned and where nodes are highly mobile. Finally the dissertation provides a proof that all replicas held in the system eventually converge towards the most recent state information of the physical world objects which they represent.

Classification

The models and algorithms which are presented in this dissertation are in the intersection of three areas of research: the chronological ordering of events

in distributed systems, data replication, and data consistency. One of the first approaches that comes to mind for deriving the chronological ordering between pairs of events is the use of synchronized physical clocks and timestamps. However, the effort needed to maintain the synchronicity of potentially all clocks in the system is high. This makes traditional time synchronization algorithms like NTP [Mil94], which originated in wired networks, unattractive for energy-constrained wireless networks like wireless sensor networks (WSN) [ER03]. There are, however, clock synchronization schemes especially designed for wireless ad-hoc and sensor networks, e.g., [Röm01, EGE02], that take limitations of resources like bandwidth and energy into account. Unfortunately, the maximum time synchronization error between any pair of nodes in the system depends on two potentially unknown system parameters: the maximum number of hops between nodes (the network diameter), and the duration of disconnection between nodes, e.g., the time network partitions prevail. Therefore, we introduce a novel approach that allows to derive chronological ordering between events that occur in the physical world with constant accuracy. In this approach the chronological ordering is derived from the order in which messages are received via a single hop in the network.

Increasing the availability of data and the overall system performance are two of the major reasons for using *replication techniques*. Depending on the particular replication method used, it is possible to continue operation in the presence of server failures or network partitioning. If there exists only one physical copy of a logical data item (non-replicated case) each access to a logical object corresponds to one access to a physical object. If, however, multiple physical copies of a logical object exist (replicated case), operations on a logical object can, in general, map to different sets of operations on the underlying physical objects. The read-one-write-all approach [BG84], for example, may map a read operation on a logical object to any one of the physical copies while write operations are executed on all physical copies. Obviously, the order in which operations on the physical copies of a data object are executed impacts the result of the operations on logical objects. *Consistency models* are therefore used to specify the correct behavior of accessing replicated data objects. In the area of (distributed) databases, numerous replication manage-

ment algorithms have been proposed to achieve one-copy consistency (e.g., see [BHG87]). The goal of these approaches is to make the system behave as if only one copy existed. However, applying these algorithms in MANETs, where network partitioning may occur frequently [HDMR04], would result in reduced availability of data, while also incurring unacceptably high communication overhead and latencies. For many applications the availability of data is more important than strong consistency, as in the scenario described above. This fact has lead to the definition of weaker consistency levels in various areas, such as directory systems, e.g., [Nee93], distributed file systems, e.g., [Sat93], or database replication, e.g., [DGH+87, KC00]. However, the consistency models used in these areas do not consider the temporal ordering of operations. Instead, they use, for example, any order as long as it is the same for all copies [KC00]. In this dissertation we define a novel consistency model which provides chronological ordering guarantees between operations on a data object. The model belongs to the class of weak consistency models and thus the replication algorithms presented in this dissertation achieve high availability of data even in networks that change dynamically over time or that are temporarily partitioned.

Outline of the Dissertation

The remainder of this dissertation is structured as follows. Chapter 2 gives a formal definition of the novel consistency model *update-linearizability* along with the system model used for the remainder of this work. Chapter 3 first provides an overview of the approach followed to derive chronological ordering. Next, two important data structures, namely the *state record* and the *ordering graph*, that are necessary for storing state information about physical objects and temporal ordering information in the system are presented. Chapter 4 describes two data replication algorithms that provably guarantee *update-linearizability* and use the data structures presented earlier. Chapter 5 presents simulation results that compare the performance of the replication algorithms. Chapter 6 discusses the related work. The dissertation closes with a summary of the presented contributions and an outlook to possible future work.

Chapter 2

Consistency and System Model

A formal model that defines the correct ordering of events that occur in the physical world is important since it defines a correctness criteria for algorithms. Thus, algorithms that guarantee a particular consistency model provide valuable guarantees to application programmers. Therefore, we first define the consistency model called *update-linearizability* [HRB04, HBM04] in this chapter. Since the basis for the ordering of events is physical time in the context of this work, the definition of consistency is preceded by a formal definition of chronological ordering. Following the definition of *update-linearizability*, we introduce the system model for which the algorithms in the remainder of this work are developed.

2.1 Consistency Model

The goal of the consistency model presented next is to order events in the physical world according to the physical time at which they occurred. As illustrated in Figure 2.1, the model defines the roles of perceivable objects (objects for short), observers, clients, and database nodes (DB nodes).

Objects are physical world objects that can be uniquely identified by appropriate sensor technology. Observers use sensors to identify and monitor the state of perceivable objects in their vicinity. They create so-called *update requests* whenever the state of an object in their vicinity changes significantly.

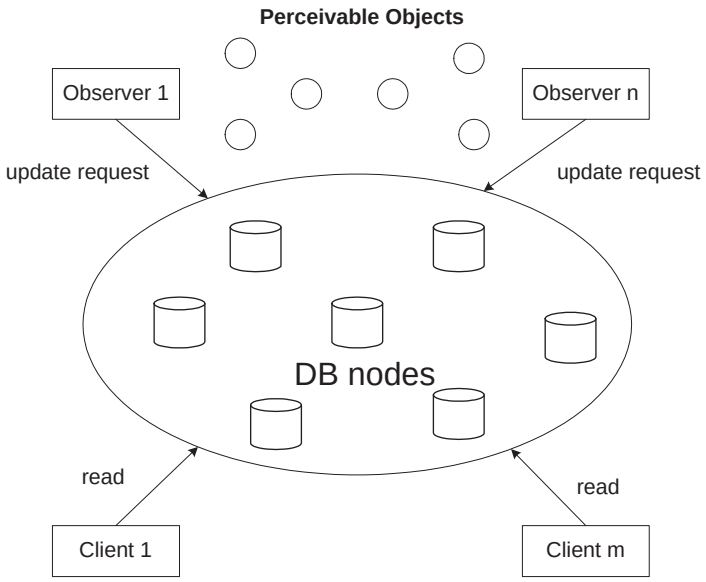


Figure 2.1: Illustration of the different roles

Whether a state change is significant or not depends on the particular application running in the system. Each observer has a unique identifier by assumption. Update requests contain the the identifier and the new state of the observed object. Clients are applications that read state information of objects. This information is, for example, used to adapt the application behavior or to provide services to other nodes in the network. Finally, DB nodes are responsible for storing object state information. Given that perceivable objects are considered to be physical objects, the state information stored in DB nodes is only updated upon changes detected by observers. Client nodes may solely read the information provided.

The goal of the consistency model presented here is to prevent that clients acquire state information that is older than what they have previously read. This means that every state change of a physical world object which is read by a client occurred in the order in which it was read. The opposite implication, i.e., that every state change which occurred in the system can be read by a client, is not guaranteed by the consistency model as a trade-off to less stringent assumptions about the underlying system model and high

availability of the data.

2.1.1 Chronological Ordering

Due to the lack of exact global time in distributed systems, the chronological order of update-requests being created by different observers can be captured only with limited accuracy. We define the *occurred-before* relationship for update-requests, which is a relaxation of exact chronological ordering, as follows.

Definition 1 (occurred-before)**.** *Let u and u' be two update requests. Then u occurred-before ($<$) u' iff $\mathrm{t_{obs}}(u') - \mathrm{t_{obs}}(u) > \delta$, where $\delta > 0$ and $\mathrm{t_{obs}}(u)$ denotes the physical time at which the state change leading to the generation of u occurred in the physical world.*

The parameter δ defines how accurate the chronological ordering of state changes can be captured in a given system. It is important for applications because it defines the minimum temporal distance between any pair of state changes which is necessary to determine their correct chronological ordering. If neither $u < u'$ nor $u' < u$, then u and u' are said to be *concurrent*, denoted as $u \| u'$. For concurrent update requests it cannot be guaranteed that correct chronological ordering is captured. The magnitude of δ depends on the system mechanisms used to determine the order of update requests. The approach used in this work will be introduced Section 3.1.

2.1.2 Definition of Consistency

The formal definition of *update-linearizability* comprises the idea that for an execution of operations there exists a totally ordered serialization against a single logical image of all DB objects and each client sees a view of the objects that is consistent with the logical image. It guarantees that updates are only performed in the *occurred-before* order within the serialization. Furthermore, the definition guarantees that read operations of a single client

for a single object are ordered according to each client's program order. As a consequence, *update-linearizability* guarantees that *a client never reads a value that is older than any value it has read before for the same object.* The formal definition of the consistency model is given as:

Definition 2 (update-linearizability)**.** *An execution of the read and update operations issued by clients and observers is said to be update-linearizable if there exists some serialization S of this execution that satisfies the following conditions:*

(C1) *All read operations of a single client on a single object in S are ordered according to the program order of the client.*

(C2) *For each object x and each pair of update requests $u[x]$ and $u'[x]$ on x in S: $u'[x]$ is a (direct or indirect) successor of $u[x]$ in S if $u[x] < u'[x]$ or $u[x] \parallel u'[x]$.*

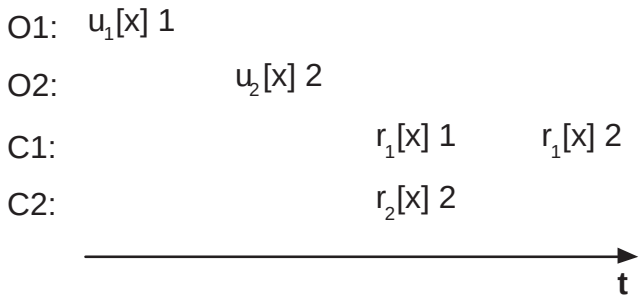
(C3) *For each object x in the database S meets the specification of a single copy of x.*

Once a client c has executed a read r_1 that returned the result of an update request u_1 on a specific object x, conditions (C1) and (C2) guarantee that the next read operation r_2 of c on x returns at least the same result as r_1 or some result written by an update request u_2, with $u_1 < u_2$ or $u_1 \| u_2$. Additionally, condition (C3) guarantees that each read operation returns the value of the update operation, which precedes the read operation in the serialization.

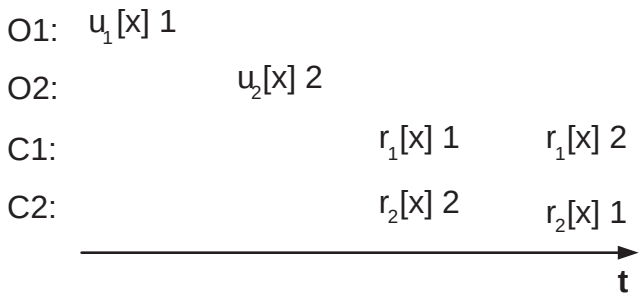
2.1.3 Examples of Executions

Figure 2.2 shows examples of valid and invalid executions according to Definition 2. The notation $u_2[x]1$ is used to describe an update request for object x, that is created by observer O2 and that writes the state 1. $r_1[y]2$ describes a read operation of client C1 that reads object y and returns state 2. The time axis runs from left to right.

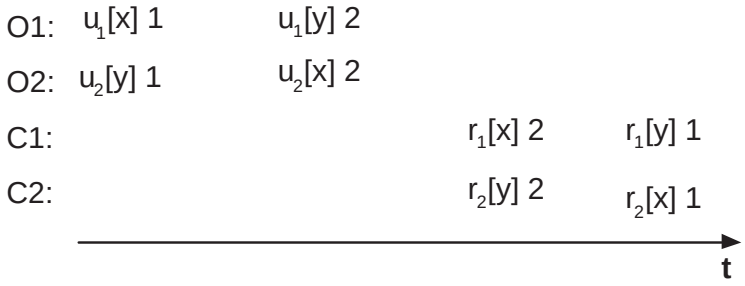
The execution in Figure 2.2(a) is correct according to Definition 2. One serialization that satisfies all conditions of Definition 2 is



(a) Valid example with one object



(b) Invalid example with one object



(c) Valid example with two objects

Figure 2.2: Example executions

$$S_a = (u_1[x], r_1[x]1, u_2[x]2, r_1[x]2, r_2[x]2).$$

Condition (C1) of Definition 2 is maintained in S_a, because the read operations of client C1 are executed according to the program order: $r_1[x]1$ precedes $r_1[x]2$. Since client C2 executes only one read operation, its program order cannot be violated. At the same time, $u_1[x]1$ precedes $u_2[x]2$ in S_a. This means that condition (C2) of Definition 2 is fulfilled, because $u_1[x]1 < u_2[x]2$ according to Definition 1 holds in the given example. Finally, condition (C3) of Definition 2 holds, because each read operation returns the value of the update request by which it is preceded in S_a.

In contrast to the previous example, Figure 2.2(b) shows an invalid execution. There exists no serialization of the execution that satisfies all conditions of Definition 2. For example, the serialization

$$S_b = (u_1[x]1, r_1[x]1, r_2[x]1, u_2[x]2, r_1[x]2, r_2[x]2)$$

satisfies condition (C2), because the update requests are ordered according to the physical time at which they were created by the observers. Under the assumption that (C3) holds, i.e., each read operation returns the value that has been written by the preceding update operation in the serialization, it can be concluded that the program order of client C2 is violated (see S_b where $r_2[x]1$ precedes $r_2[x]2$). Consequently, if condition (C1) holds, condition (C3) is violated. Given the partial serialization

$$S'_b = (r_1[x]1, r_1[x]2, r_2[x]2, r_2[x]1)$$

of all read operations that fulfills condition (C1) and starts with the program of client C1, there exists no way of inserting the update operations into S'_b without violating the other two conditions. Update $u_1[x]1$ needs to be inserted before the first read operation in S'_b in order to fulfil condition (C3) for $r_1[x]1$. Additionally, $u_2[x]2$ has to be inserted before $r_1[x]2$ in order to fulfil condition (C3) for that read operation, which results in the following serialization

$$S_b'' = (u_1[x]1, r_1[x]1, u_2[x]2, r_1[x]2, r_2[x]2, r_2[x]1).$$

While condition (C2) is fulfilled for the update operations, condition (C3) is violated for $r_2[x]1$ (which returns 1 as the result), since the preceding update operation in S_b'' has written the value 2 to x. All permutations of the read operations that fulfil condition (C1) and start with the program of client C2 will require that $u_2[x]2$ comes first in the serialization in order to fulfil condition (C3). The consequence of this is, that condition (C2) will be violated.

The example in Figure 2.2(c) is a valid execution with two objects, because update-linearizability is an object-local property and both clients read each object only once. This means that condition (C1) will always be fulfilled. A valid serialization for this execution is

$$S_c = (u_1[x]1, r_1[x]1, u_2[x]2, r_2[x]2, u_2[y]1, r_1[y]1, u_1[y]2, r_2[y]2)$$

which first orders all operations for object x and then all operations for object y according to Definition 2.

2.1.4 Using Update-Linearizability

From the perspective of a programmer the concept of update-linearizability is different from classical strict consistency models such as one-copy consistency [BHG87].

Consider a program that monitors the state of an object, e.g. the temperature of an object in a fire-fighting application. The task of the program is to send a notification to another process if a temperature threshold is exceeded. Let $state_{below}$ be a value below and $state_{above}$ a value above a given threshold. Definition 2 guarantees that if the monitoring process reads a sequence $state_{below}$ then $state_{above}$ there was a state change from below the threshold to above the threshold. The same holds respectively for reading $state_{above}$ first and then $state_{below}$. This means that the monitored state of

the object crossed the threshold in the observed direction, if the monitoring process sends a notification. The opposite implication, i.e., that each state change of the object causes a notification, is not guaranteed by the consistency model. Computations that involve reading multiple objects may be regarded as concurrent clients where one object is read by each client, because update-linearizability is a local property for each object.

2.2 System Model

The system in which the algorithms presented in this work are assumed to run consists of a set of nodes $\mathcal{N}$. Each node is a programmable autonomous system with at least a CPU, memory, and a bi-directional wireless communication interface. Nodes are initially deployed on a spatial area, the operational environment, and may be either mobile or stationary. Each node can be in one or more of the following roles as depicted in Figure 2.1: observer, client, and database node (DB node).

Besides the nodes in the network, other (physical) objects may be located in the operational environment. These *perceivable objects* (objects) are not nodes of the network and therefore do not receive or send packets. It is assumed that these objects may be uniquely identified using appropriate sensor technology, such as radio frequency identification tags (RFID tags, e.g. [epc]). Furthermore, the state of these objects is monitored by observers over time to detect state changes.

2.2.1 Communication between Nodes

Any node is able to exchange data packets directly with another node via a *single hop*, if the spatial distance between them is less than r_{tx}. The parameter r_{tx} depends on the particular communication technology used and effects of radio propagation, such as the reflection or absorption of radio waves in the operational environment. Given a particular node $\mathrm{n}_i \in \mathcal{N}$ and a time t, the set of neighbors $\mathit{neighbors}(\mathrm{n}_i, t) \subseteq \mathcal{N}$ is defined as the set of all nodes with whom n_i is able to communicate with directly at time t using the

`send`(m) primitive. We assume that the `send` primitive delivers a message m to all neighbors of the sender with best-effort semantics.

A node may communicate with other nodes than its neighbors using a multi-hop routing algorithm with `uni-send`(*destination*, m). The primitive `uni-send` delivers the message m to a receiver with the unique address *destination* with best-effort semantics.

A node may communicate with a group of other nodes using a multi-hop multi-cast algorithm using `multi-send`(*dest-group*, m). The primitive sends a message to a group of receivers (multi-cast group) specified by a group address *dest-group* with best-effort semantics. Nodes may become part of the multi-cast group at any time by using `multi-attach`(*dest-group*) and leave the group by using `multi-detach`(*dest-group*). The well-known group `DBGRP` represents the group of all DB nodes.

2.2.2 Observation Jitter

The approach for deriving the chronological ordering between update requests is introduced in Section 3.1. It is influenced by two parameters of the system: the jitter[1] δ_{comm} of the end to end communication delay between two nodes in the network on a single hop using `send` and the jitter δ_{sens} of the time it takes to observe a state change by means of sensors. The sum of both terms is called *observation jitter* or δ_{obs}.

For the communication technology used, it is assumed that δ_{comm} is bounded and known. This can, for example, be achieved by using broadcast communication on layer two of the protocol stack, where packets are not repeated upon communication failures, sent within a bounded randomization interval, and are queued in a finite interface queue. These requirements are, for example, fulfilled by the specification of the IEEE 802.11 Standard (distributed coordination function, [Boa97]) and the MAC implementation provided with TinyOS [HSW+00], an operating system designed for wireless sensor network platforms such as the Berkeley Motes [mot].

[1] Delay jitter is defined to be the maximum *difference* between delays experienced by any two packets [ZK91].

Regarding the jitter imposed by sensor technology, we assume within this work that $\delta_{\text{sens}} = 0$, i.e., it always takes the same time to retrieve and process data from the sensor hardware.

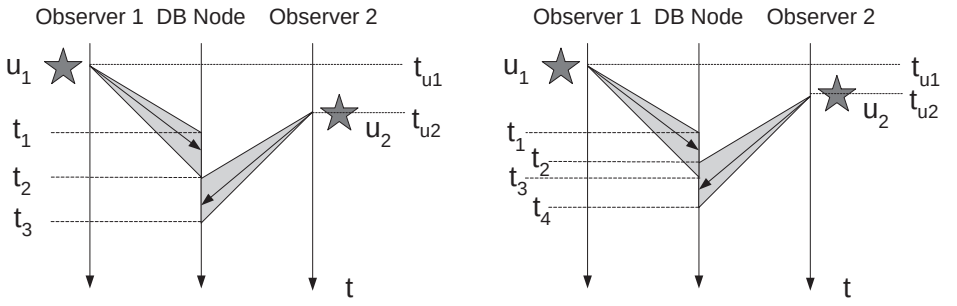
Chapter 3

Basic Concepts and Data Structures

In this chapter, we introduce the basic concepts and the data structures that are used within this dissertation for the development of the replication algorithms presented in Chapter 4. The two main data structures are the state record and the ordering graph. The state record reflects all information necessary to represent an update request in the system. The ordering graph is used to maintain information about the chronological ordering of update requests created by different observers.

3.1 Basic Concepts for Deriving Chronological Ordering

A frequently used approach for deriving the chronological ordering between pairs of update requests is the use of synchronized physical clocks. Here, timestamps are associated with the events by the sensors that captured the data. The chronological ordering and the time elapsed between a pair of events can simply be determined by comparing their associated timestamps. This, however, requires that the clocks of the nodes in the network are synchronized. This can be achieved by using infrastructures that provide accurate time information, for example, the Global Positioning System (GPS), or



(a) Unambiguous ordering possible (b) Unambiguous ordering not guaranteed

Figure 3.1: Ordering of two packets at the receiver, with δ_{comm} (shaded grey)

by using clock synchronization schemes. The GPS infrastructure, however, is not available in indoor scenarios since it requires line of sight communication with the GPS satellites. Clock synchronization can be either be performed proactively (often periodically), e.g., [Mil94] or reactively using posteriori synchronization, e.g., [Röm01]. Clock synchronization schemes especially designed for wireless ad-hoc and sensor networks, e.g., [Röm01, EGE02], take limitations of resources like bandwidth and energy into account. However, the maximum time synchronization error between any pair of nodes in the system depends on two potentially unknown system parameters: the maximum number of hops between nodes (the network diameter), and the duration of disconnection between nodes, e.g., the time network partitions prevail. While the former may be critical in large networks, the latter plays a significant role in sparsely connected networks with mobile nodes where events that need to be correlated at some point in time occurred earlier on in two distinct network partitions. Due to the practically inevitable clock drift, the clocks in distinct network partitions will run at different speeds even if a periodic synchronization of the clocks within each partition is carried out. This leads to an offset between clocks that grows with the duration of the disconnection periods.

In order to achieve ordering accuracy which is independent from the network diameter and the age of information, we propose a novel concept for determining the chronological ordering between update requests. Our approach is based on the order in which update requests are *received* by nodes

in the network via a single-hop. The concept is based upon the assumption that the jitter δ_{comm} of the communication delay on a single-hop in the network has an upper bound. Figure 3.1 shows two examples with two observer nodes and one DB node each. In both cases, the two observers send an update request for an object at times t_{u1} and t_{u2}, where $t_{u1} < t_{u2}$. The communication between the observers and the DB node is direct, i.e. no intermediate hops are used. The shaded triangles indicate the maximum delay jitter δ_{comm} that the individual packets may experience. By the assumptions taken in Chapter 2 this jitter is bounded.

In Figure 3.1(a) the difference $\delta_{\text{comm}} = t_2 - t_1$ is equal to the difference between t_{u1} and t_{u2} at which the update requests u_1 and u_2 were created. As a result of that, the order in which the packets sent by the observers is reflected in the order in which the messages are received. The example in Figure 3.1(b) shows a situation where the difference $t_{u2} - t_{u1}$ between the sending of the two update request messages is less than δ_{comm}. As a result of that, the message containing u_1 may arrive after t_2 (but before t_3) and the message containing u_2 may arrive earlier than t_3 (but not before t_2). Since the difference $t_2 - t_3$ is negative, the order in which the messages arrive may be different from the order in which they were sent as a cause of the possible jitter.

In general, when $t_{u2} - t_{u1} \geq \delta_{\text{comm}}$, the order in which the messages were sent can be unambiguously derived from the order in which they are received at the DB node. In all other cases, when $t_{u2} - t_{u1} < \delta_{\text{comm}}$, that order may not be maintained in all cases, depending on the effective jitter the messages have experienced in the system. In contrast to timestamp-based approaches, however, our approach cannot be used to calculate the temporal difference between update requests.

Experiments presented in Chapter 5 show that δ_{comm} is reasonably small using standard technology. The values measured for δ_{comm} using Berkeley Motes are below 1ms. If synchronized (physical) clocks are used the possible ordering accuracy depends on the accuracy of the underlying clock synchronization algorithm. In the case of the time synchronization algorithm for MANETs presented in [Röm01] the *inaccuracy increases linearly* with both

the number of hops a message traverses in the network and the age of the information. Measurements on standard PCs presented in [Röm01] show inaccuracies of approximately 2.2 ms and 3.75 ms for information that is 500 and 900 seconds old.

3.2 Basic Concepts for Data Replication

The replication algorithms in Chapter 4 are based on the previously presented approach of deriving the chronological ordering of update requests by the order in which they are received. This section describes the basic concept used to replicate object state information in the system. Figure 3.2 outlines the process of replicating the state information contained in an update request on each DB node in the system. Initially, a perceivable object is sensed by an observer which captures the *id* and the current state of the object (phase 1 in Figure 3.2). Next, the observer creates a so-called state record for the observation it has just made. The state record contains information about the object that has been observed and the observer that has made the observation. After creation, the state record is forwarded to all DB nodes in the single-hop communication range of the observer using the observer-node algorithm (phase 2). Based on the ordering concept introduced in the previous section, DB nodes will now decide on accepting or rejecting the information. If the information is accepted, the DB node creates new chronological ordering information between the new update request and update requests that have been previously processed by the node. The ordering information is then stored in the so-called ordering graph using the *add* function of the ordering graph. Finally, the DB node sends the update request – which now contains the state record and the ordering graph – to other DB nodes in the system using the node-node algorithm (phase 3). A DB node receiving this information can now use the *join* and *occurredBefore* functions of the ordering graph to derive the chronological order between the received information and the locally stored information for the given object. The remainder of this chapter describes the basic data structures used in the replication algorithms.

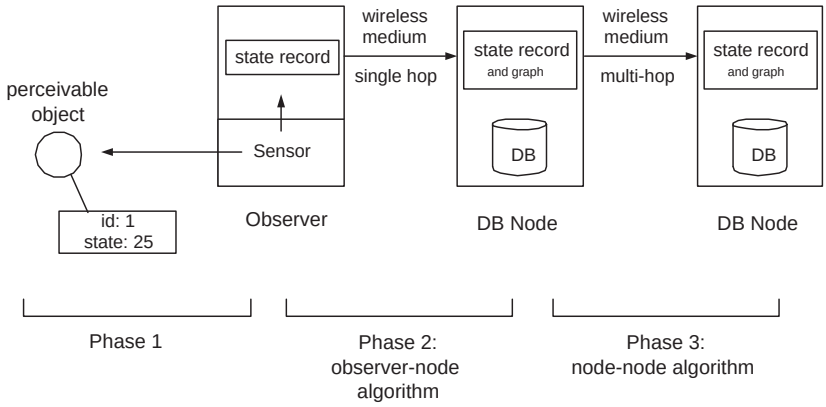


Figure 3.2: Overview of the update operation of the replication algorithms

3.3 State Record

A state record is used to collect all necessary information to reflect an update request in the system and is defined as a 4-tuple $(obj, state, obs, sn)$. In this tuple, *obj* is the unique identity of the perceivable object concerned. The state of the perceivable object associated with the state record is stored in the *state* component of the tuple. The identity of the observer which created the state record is stored in *obs*. Additionally, a unique and strictly monotonic increasing sequence number is created by *obs* and is stored in *sn*. Each observer uses a single counter for the sequence numbers independent of the number of objects it observes.

The notation $\mathrm{u}[x]_{sn}^{obs}$ is used to denote a state record created for object x by observer *obs* with the sequence number *sn*. Similarly, $\mathrm{db}[x]_{sn}^{obs}$ is used to denote a state record which is stored in the local database of a DB node. The set of all state records is denoted by $\mathcal{R}$.

3.4 Ordering Graph

In cases where two state records are created by the same observer, the order in which the state changes of the respective object occurred can be determined by comparing the sequence numbers of the records. Whenever the order

between two state records created by independent observers must be derived, additional ordering information is necessary. The purpose of the ordering graph is to provide a data structure for storing such information.

Definition 3 (ordering graph)**.** *Let* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *denote an ordering graph. The set of vertices* V *contains state records. The set of directed edges* E *reflects the ordering relationship between state records, such that when an edge* $(\mathrm{u}[x]_i^o, \mathrm{u}[y]_k^p)$ *exists in* E *it holds that* $\mathrm{u}[x]_i^o < \mathrm{u}[y]_k^p$.

The set of all ordering graphs is denoted as $\mathcal{G}$. $\mathcal{V}$ and $\mathcal{E}$ denote the sets of vertices and edges, respectively.

Note, that by construction the ordering graph is a directed *acyclic* graph. A cycle in the graph would contradict the definition of the ordering graph, because each of the graph vertices in the cycle would contain an observation which transitively occurred before itself.

3.4.1 Adding Ordering Information

In order to add new ordering information to an ordering graph, we define the operation add : $\mathcal{G} \times \mathcal{R} \rightarrow \mathcal{G}$ as shown in Algorithm 1.

Algorithm 1 The *add* function

1: `function` add($G_{\text{in}}, \mathrm{u}[x]_{sn}^{Obs}$) `returns` $\mathcal{G}$
2: $V' \leftarrow V_{\text{in}} \cup \{\mathrm{u}[x]_{sn}^{Obs}\}$
3: $E' \leftarrow E_{\text{in}} \cup \{(u, \mathrm{u}[x]_{sn}^{Obs}) \mid u \in V_{\text{in}} \setminus \{\mathrm{u}[x]_{sn}^{Obs}\}\}$
4: $G' \leftarrow$ `reduce`(V', E')
5: `return`(G')

Given an ordering graph G and a state record $\mathrm{u}[x]_{sn}^{obs}$, the *add* operation inserts the state record into the set of vertices. Additionally, edges from all vertices previously contained in G to the new vertex $\mathrm{u}[x]_{sn}^{obs}$ are added to the set of edges. The *reduce*[1] function described next is then used to reduce the size of the resulting graph. In order to show the correctness of the *add* function, we need to proof the following claim.

[1]Whenever *reduce* is used in an algorithm, one of the presented functions for reducing an ordering graph may be used.

Correctness

Claim: Given that a graph G is correct according to Definition 3. If the *add* function is applied by DB nodes to G after receiving an update request directly from an observer, the resulting graph G' will not contain invalid ordering information. We take the assumption that the node using the *add* function at time t_r has not received an update request from a different observer in the time period $[t_r - \delta_{\text{comm}}, t_r]$. This assumption is guaranteed by the replication algorithms in the next chapter.

Proof. Without loss of generality we assume that a DB node n_i receives an update request containing a state record $\text{u}[x]_{sn}^{Obs}$ from observer Obs. First, the function adds the new vertex $\text{u}[x]_{sn}^{Obs}$ to the graph in line 2. This is a vertex created from an update request and therefore it is valid to add it to the graph. By the assumptions taken, all update requests in $V \setminus \{\text{u}[x]_{sn}^{Obs}\}$ *occurred-before* $\text{u}[x]_{sn}^{Obs}$ (the update request was received more than δ_{comm} after the last update request from a different observer). Therefore, *add* inserts edges from all vertices previously contained in the graph ($V \setminus \{\text{u}[x]_{sn}^{Obs}\}$). The correctness of the *reduce* functions is discussed in Section 3.4.2. □

3.4.2 Reducing the Ordering Graph

Every time the *add* operation is used, the amount of storage needed for the graph grows. In order to keep the size of the ordering graph reasonably small, it is desirable to remove as much information as possible while still being able to determine the chronological ordering of state records. The typical pair of records, whose ordering needs to be determined, is one record that has recently been created by an observer and another record that is the currently the most recent information about an object stored on a DB node. This means, that older vertices stored in the graph are less likely to be needed for deciding on the ordering of state records. In general, we distinguish between two classes of information removal. First, it is possible to remove only such information from the graph that does not cause any loss of ordering information (lossless reduction). Secondly, we follow approaches that do

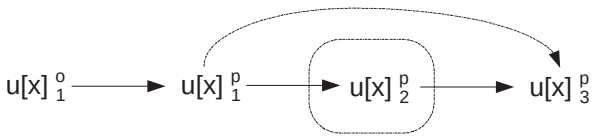


Figure 3.3: Example of using *lossless-reduce*

remove ordering information from the graph (lossy reduction). However, the focus here is to remove old vertices (and all in- and out-going edges). The approaches differ in how much ordering information is lost as well as their computational complexity and the memory complexity of the resulting graph. We refine the approach taken in [Eic04]. Additional approaches that select a sub-graph for transmission without reducing the local data structure are discussed in Section 4.3 (p. 98). Experiments presented in Chapter 5 show that the negative effect of using lossy reduction is low. In the given scenarios (Figure 5.8, p. 121), for example, the percentage of rejected update requests due to unknown ordering is in the order of 3 %.

Lossless Reduction

The first approach is to only remove information from the graph that does not cause the loss of any ordering information. To do so, the *lossless-reduce* function makes use of the sequence numbers inserted by observers into each state record[2]. The idea is to remove such vertices from the graph that have only in- and out-going edges from and to vertices which contain state records created by the same observer. The formal definition of the function *lossless-reduce* is given in Algorithm 2.

Figure 3.3 shows a graph were the *lossless-reduce* function removes the vertex $u[x]_2^p$. This vertex has only in- and out-going edges to vertices created by the same observer (which is p) and does not have the highest sequence number. Vertex $u[x]_3^p$ must not be removed, because it is the vertex with the highest sequence number in the graph. The vertex $u[x]_1^p$ must not be

[2]Recall that the sequence numbers issued by each observer are strictly monotonic increasing.

removed, because there exists an in-going edge originating at $\mathrm{u}[x]_1^o$ which was created by a different observer. The edge $(\mathrm{u}[x]_1^p, \mathrm{u}[x]_3^p)$ is inserted by *addTransitive* to reflect the relationship between the two vertices.

Algorithm 2 The *lossless-reduce* function

```
1: function lossless-reduce(G_in) returns 𝒢
2: V_d ← ∅
3: {select vertices for removal}
4: for all v ∈ V_in do
5:    V_to ← selectTo(V_in, E_in, v)
6:    V_from ← selectFrom(V_in, E_in, v)
7:    V_ft ← V_from ∪ V_to
8:    remove ← true
9:    for all v' ∈ V_ft \ {v} do
10:      if v'.obs ≠ v.obs then
11:         remove ← false
12:      end if
13:   end for
14:   if remove then
15:      V_d ← V_d ∪ {v}
16:   end if
17: end for
18: {do not remove highest sequence numbers}
19: for all v ∈ V_d do
20:   if v.sn = high(V_d, v.obs) then
21:      V_d ← V_d \ v
22:   end if
23: end for
24: return addTransitive(G_in, V_d)
```

In Algorithm 2, the functions selectTo(V, E, v) is used to select all vertices from V that have an out-going edge in E leading to vertex v (the predecessors of v in E). Similarly, the function selectFrom(V, E, v) selects all vertices in V that have in-going edges in G originating at v (the successors of v in G). The function purgeEdges(V, E) removes all edges (v_1, v_2) from E that include at

least one vertex which is element of V.

Using these functions, *lossless-reduce*, iterates over all vertices v in G_{in} (lines 4-17) and selects the predecessors and successors for each vertex and stores them in V_{ft} (lines 5-7). If all vertices $v' \in V_{\text{ft}} \setminus \{v\}$ were created by the same observer, vertex v is stored in V_{d} for removal (lines 9-16). However, it is necessary to ensure that not all vertices created by an observer are removed from the graph. To do so, the algorithm iterates over all vertices collected in V_{d} and ensures that for each observer the vertex with the highest sequence number is kept in the graph (by removing it from V_{d}, lines 19-23). For that purpose, the function *high* used in line 20 returns for a set of vertices V and an observer *obs* the highest sequence number for *obs* contained in any vertex in V_{d}.

The function *addTransitive* given in Algorithm 3 and used in line 24 calculates transitive edges in the given graph that span vertices selected for removal. This prevents that ordering information is lost purely to the removal of vertices. Finally, *lossless-reduce* returns a graph where all vertices selected in V_{d} are removed and the transitive edges are added.

Correctness of *lossless-reduce*

Claim: Given a correct graph G_{in} according to Definition 3, the function lossless-reduce(G_{in})

- does not create false ordering information,
- does not remove ordering information that is contained in G_{in}, and
- the resulting graph has the same number or less vertices than G_{in}.

In particular the most recent vertex of each observer that can be found in G_{in} will remain in the resulting graph.

Proof. We take the assumption that the function *addTransitive* is correct (discussed next). In general, removing edges and vertices from a correct ordering graph will result in a correct graph, because the definition of an

ordering graph (Definition 3) does not require the equivalence between the existence between an edge in the graph and the actual ordering of the corresponding observations. Only if a particular edge exists, it is required that the ordering of the observation is correct. Since the function only removes vertices it does not create false ordering information.

The algorithm selects those vertices $v = \mathrm{u}[x]_i^o$ for removal that have only predecessors and successors created by the same observer o (lines 4-17 in Algorithm 2). The ordering information between those vertices and their predecessors and successors can be derived by comparing their sequence numbers, since observers use strictly monotonic increasing sequence numbers by assumption. Only when a vertex created by an observer o has no successors and its predecessor was also created by o, it must not be removed, because it is the most recent vertex for o. This is guaranteed by the algorithm in lines 19-23. Thus, the vertices in V_d can be removed without losing ordering information. □

Algorithm 3 Function to add transitive edges

```
1: function addTransitive(G_in, V_d) returns G
2: {collect border nodes and add trans. edges}
3: E_add ← ∅
4: V_dd ← ∅
5: for all v ∈ V_d do
6:    V_dd ← V_dd ∪ {v}
7:    V_af ← selectFrom(V_in \ V_dd, E_in ∪ E_add, v)
8:    V_at ← selectTo(V_in \ V_dd, E_in ∪ E_add, v)
9:    for all v_f ∈ V_af do
10:      for all v_t ∈ V_at do
11:         E_add ← E_add ∪ {(v_t, v_f)}
12:      end for
13:   end for
14: end for
15: E_purged ← purgeEdges(V_d, E_in ∪ E_add)
16: return G(V_in, E_purged)
```

Correctness of *addTransitive*

Claim: The function *addTransitive*($G_{\text{in}}, V_{\text{d}}$) adds only edges which are correct for G_{in} according to Definition 3. V_{d} contains vertices that are going to be removed from the graph. In particular, for the vertices that remain in the resulting graph it holds: if a path existed between a pair of such vertices in G_{in}, there also exist a path between the same pair in the resulting graph.

Proof. The function *addTransitive* iterates all vertices that are selected for removal in V_{d}. For each of these vertices $v \in V_{\text{d}}$ it selects the predecessors and successors from the set of all vertices except those elements from V_{d} that have already been examined ($V_{\text{in}} \setminus V_{\text{dd}}$). In lines 9-13 the function adds a new edge for each pair of predecessor and successor. Since there exists a path in G_{in} between those pairs (going through a vertex in V_{d}), it is valid to include these edges.

If two (or more) successive vertices in G_{in} are selected for removal, they are by-passed sequentially by removing one after the other (independent of the order in which they are removed). Unnecessary edges are removed in line 15 by calling purgeEdges($V_{\text{d}}, E_{\text{in}} \cup E_{\text{add}}$).

□

Lossy-*k*-Reduce

The second approach to reducing an ordering graph explicitly allows to remove ordering information from the graph. Since the ordering graph is sent over the network this is done to reduce the amount of memory needed for the ordering graph. The basic idea is to only keep the k most recent vertices of each observer in the graph and remove all other vertices. Similar to the *lossless-reduce* function, transitive edges are added to span the removed edges. The formal definition of the *lossy-k-reduce* function in presented in Algorithm 4.

In order to show the correctness of the *lossy-k-reduce* function we need to support the following claim.

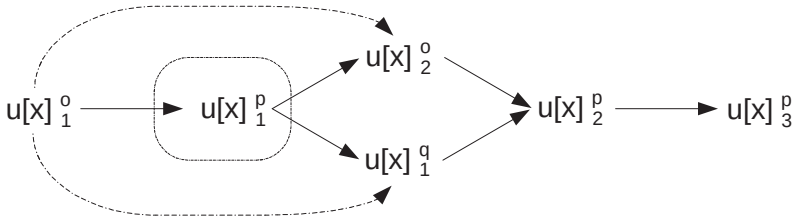


Figure 3.4: Example of using *lossy-k-reduce* with $k = 2$

Claim: Given a correct graph G_{in}, the function lossy-k-reduce(G_{in}) returns a correct graph with at most k vertices per observer. The vertices for each observer that remain in the resulting graph are the most recent ones for each observer in G_{in}, i.e., the ones with the highest sequence numbers. Additionally, the function does not introduce false ordering information.

Proof. By assumption *addTransitive* is correct. The algorithm first calculates the set of all observers represented in the given graph by using the function *getObservers* in line 4. Next, it iterates over all observers and selects for each observer the (at most) k vertices with the highest sequence numbers by using *getHighestSN* in line 6. The vertices selected for removal in V_{d} are those which have not been selected by *getHighestSN*. This means that at least the most recent vertex of each observer remains in the graph. Finally, the algorithm calculates the set of transitive edges using *addTransitive* which has shown to be correct in the previous section. The algorithm returns a graph where all vertices in V_{d} are removed and the transitive edges in E_{add} are added. Since *lossy-k-reduce* neither adds vertices nor edges (except in *addTransitive*) it does not create false ordering information. □

An example for the use of *lossy-k-reduce* with $k = 2$ is depicted in Figure 3.4. Here, the vertex $\text{u}[x]_1^p$ is selected for removal, since there exist three vertices created by observer p in the graph. The function *addTransitive* adds two new edges to the graph pointing from $\text{u}[x]_1^o$ to $\text{u}[x]_2^o$ and $\text{u}[x]_1^q$, because there existed a path from $\text{u}[x]_1^o$ to the two latter vertices going through the removed vertex in the original graph.

Algorithm 4 The *lossy-k-reduce* function

1: `function` lossy-k-reduce(G_{in}, k) `returns` $\mathcal{G}$
2: $V_{\text{d}} \leftarrow \emptyset$
3: {select vertices for removal}
4: $V_{\text{obs}} \leftarrow \text{getObservers}(V_{\text{in}})$
5: **for all** $o \in V_{\text{obs}}$ **do**
6: $V_{\text{k}} \leftarrow \text{getHighestSN}(V_{\text{in}}, o, k)$
7: $V_{\text{all}} \leftarrow \text{getAll}(V_{\text{in}}, o)$
8: $V_{\text{d}} \leftarrow V_{\text{d}} \cup (V_{\text{all}} \setminus V_{\text{k}})$
9: **end for**
10: `return` addTransitive($G_{\text{in}}, V_{\text{d}}$)

3.4.3 Joining Ordering Information

Whenever two DB nodes exchange state records in the replication algorithms described in Chapter 4, they need to also exchange ordering information. A DB node receiving the ordering information stored at another DB node therefore needs to consistently integrate that information into the ordering information locally known. This integration is done by using the join : $\mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ function which is defined in Algorithm 5.

Algorithm 5 The *join* function

1: `function` join(G_1, G_2) `returns` $\mathcal{G}$
2: $E = E_1 \cup E_2$
3: $V = V_1 \cup V_2$
4: $G = \texttt{reduce}(V, E)$
5: `return`(G)

The two ordering graphs which are passed to the *join* function are merged by calculating the set union of both the two sets of vertices and the two sets of edges. Again, the *reduce* function is used to reduce the size of the ordering graph.

Correctness

Claim: Given that G_1 and G_2 are correct graphs, join(G_1, G_2) does not create false ordering information. If the *reduce* function that is applied in line 4 does not remove any vertices or edges, the result of *join* will at least contain the ordering information that is contained in either G_1, G_2, or both graphs.

Proof. By applying the set union to edges and vertices the new graph returned by the join(G_1, G_2) function contains all ordering information contained in G_1 and G_2. Any additional ordering information that is contained in the result is the valid transitive concatenation of ordering information from G_1 and G_2. This is caused if G_1 and G_2 have common vertices.

□

3.4.4 Determining the Ordering of Update Requests

DB nodes have to decide whether or not to accept received update requests. Consider the case where the local DB includes state record $\mathrm{db}[x]_i^o$ and update request $\mathrm{u}[x]_j^p$ is received. To preserve consistency, $\mathrm{u}[x]_j^p$ may only be accepted if the update request that wrote $\mathrm{db}[x]_i^o$ occurred-before $\mathrm{u}[x]_j^p$ or both requests are concurrent. If both requests are from the same observer ($o = p$) the update request can be accepted if $j > i$. If both update requests come from different observers, the ordering graph G_{in} passed to *occurredBefore* has to be evaluated to decide whether the update has to be accepted: $\mathrm{u}[x]_j^p$ has to be accepted if *occurredBefore*$(G_{\mathrm{in}}, \mathrm{db}[x]_i^o, \mathrm{u}[x]_j^p)$ evaluates to `true`. If the predicate evaluates to `false`, the ordering of the two vertices is either `unknown` or $\mathrm{u}[x]_j^p < \mathrm{db}[x]_i^o$. The predicate *occurredBefore* is defined in Algorithm 6. The algorithm is described next as part of a constructive proof, which is followed by an example for using the algorithm.

Correctness of occurredBefore

Claim: occurredBefore$(G_{\mathrm{in}}, \mathrm{u}[x]_i^o, \mathrm{u}[y]_j^p)$ == `true` implies that $\mathrm{u}[x]_i^o < \mathrm{u}[y]_j^p$ (*occurred-before*). By assumption, the graph G_{in} is a correct ordering graph.

Algorithm 6 The *occurredBefore* predicate on ordering graphs

```
1: function occurredBefore(G_in, u[x]_i^o, u[x]_j^p) returns boolean
2: if o = p then
3:   if j > i then
4:     return true
5:   else
6:     return false
7:   end if
8: else
9:   v_from ← getVertexGE(V_in, o, i)
10:  v_to ← getVertexLE(V_in, p, j)
11:  if (v_from ≠ null) ∧ (v_to ≠ null) then
12:    if hasPath(E_in, v_from, v_to) then
13:      return true
14:    else
15:      return false
16:    end if
17:  else
18:    return false
19:  end if
20: end if
```

Proof. The *occurredBefore* predicate first compares the identities of the observers o and p. If these identities are equal, the predicate evaluates to `true`, given that the sequence number i is less than j. If $i < j$ holds, it can be derived that $\mathrm{u}[x]_i^o < \mathrm{u}[y]_j^p$, by the assumption that sequence numbers of observers increase strictly monotonic.

If the vertices $\mathrm{u}[x]_i^o$ and $\mathrm{u}[x]_j^p$ were created by different observers ($o \neq p$), the graph is used to determine the ordering between the given vertices (lines 9-19). First, the function getVertexGE(V_{in}, o, i) retrieves the vertex v_{from} from V_{in} that was created by observer o and has the smallest sequence number available in the graph which is greater or equal to i. Similarly, getVertexLE(V_{in}, p, j) returns the vertex v_{to} that was created by observer p and has the highest sequence number which is less or equal to j. Both functions, *getVertexLE* and *getVertexGE* return `null` to indicate that no vertex with the given observer *id* was found.

If $\mathrm{u}[x]_i^o$ is a vertex in the graph, it will be selected as v_{from} by *getVertexGE*. If not, some vertex $\mathrm{u}[x]_{i+m}^o$ with $m > 0$ will be selected, if it exists. In this case it holds that $\mathrm{u}[x]_i^o < \mathrm{u}[x]_{i+m}^o$ because of the assignment of the sequence numbers. If no vertex $\mathrm{u}[x]_{i+m}^o$ with $m > 0$ exists *getVertexGE* returns `null` and therefore *occurredBefore* returns `false`. A similar argumentation holds when getVertexLE(V_{in}, p, j) is applied to find a vertex v_{to} in the graph. If $\mathrm{u}[x]_j^p$ is in the graph, it will be selected. Otherwise, either `null` or some vertex $\mathrm{u}[x]_{j-n}^p$ with $n > 0$ is returned. In the latter case it can be concluded that $\mathrm{u}[x]_{j-n}^p < \mathrm{u}[x]_j^p$ by comparing the sequence numbers $j - n$ and j.

Next, the algorithm examines, if a path from v_{from} to v_{to} exists. If such a path is found, it can be concluded that $v_{\mathrm{from}} < v_{\mathrm{to}}$, because the *occurredBefore* relation is transitive. Following the previous discussion about using *getVertesGE* and *getVertexLE* it can also be concluded that $\mathrm{u}[x]_i^o < \mathrm{u}[x]_j^p$. Therefore, it is valid to return `true`. □

Example for occurredBefore

If, for example, the predicate occurredBefore($G, \mathrm{u}[x]_1^o, \mathrm{u}[x]_3^p$) is evaluated using the graph depicted in Figure 3.5 it returns `true`. Since the vertex $\mathrm{u}[x]_1^o$

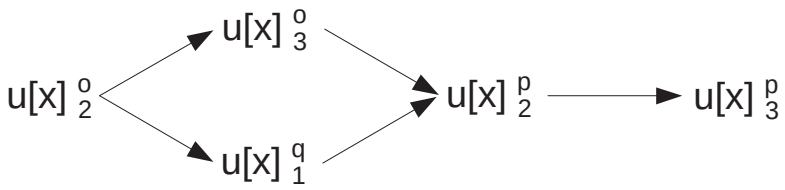


Figure 3.5: Example graph G for using *occurredBefore*

is not in the graph, getVertexGE($V_{\text{in}}, o, 1$) returns the vertex u$[x]_2^o$, which is then used when searching for a path in line 11 of the algorithm. Since an appropriate path is found, it can be concluded that u$[x]_1^o$ *occurred-before* u$[x]_3^p$. The evaluation of the predicate occurredBefore(G, u$[x]_3^o$, u$[x]_1^q$) using the same graph will return `false`, because there exists no path between the update requests u$[x]_3^o$ and u$[x]_1^q$ in the graph.

3.4.5 Ordering the State of Distinct Objects

In some applications it is not only important to have access to the most recent state of each object. Additionally, it may be beneficial to compare the chronological ordering between the state records of two or more distinct objects. If an application,for example, is monitoring the temperature of two objects, it may be necessary to reason about which object's temperature has crossed a threshold first. To do so, only minor changes to the data structures used in the above algorithms are necessary.

Instead of using one ordering graph per object, it is possible to use one single ordering graph in which the ordering information for all objects are maintained. The *add*, *join*, the *losslessReduce*, and the *occurredBefore* operations remain unchanged. The *lossy-k-reduce* operation needs to be modified in such way that the k most recent vertices for each observer and each object are kept in the ordering graph. The *add* operation used in the observer-node algorithm (Alg. 7, p. 72) can be safely applied to a single ordering graph, since the new update request being added to the graph is more recent than all update requests in the graph independent from the object that is concerned.

Function	Complexity	Remark
add	$\mathcal{O}(\lvert V_{\text{in}}\rvert^3)$	$\mathcal{O}(\lvert V_{\text{in}}\rvert^2)$ when using *lossy-k-reduce*
lossless-reduce	$\mathcal{O}(\lvert V_{\text{in}}\rvert^3)$	
lossy-k-reduce	$\mathcal{O}(\lvert V_{\text{in}}\rvert^2)$	
join	$\mathcal{O}(\lvert V_{\text{in}}\rvert^3)$	$\mathcal{O}(\lvert V_{\text{in}}\rvert^2)$ when using *lossy-k-reduce*
occurredBefore	$\mathcal{O}(\lvert V_{\text{in}}\rvert + \lvert E_{\text{in}}\rvert)$	

Table 3.1: Computational complexities of the graph operations

3.4.6 Complexity of the Operations

Both, the space and the computational complexities of the graph operations depend on the data structures used to internally represent a graph. Here, we will assume that the ordering graph $G = (V, E)$ is stored as a list of vertices and a list of adjacent vertices for each vertex (adjacency list). Internally, the vertices are numbered with natural numbers between $\{0 \dots |V| - 1\}$. Therefore, we assume that there exists a bijective function index : $V \rightarrow \{0 \dots |V| - 1\}$ that assigns a natural number from $\{0 \dots |V| - 1\}$ to each vertex of a graph (the so-called vertex numbers). Additionally, there exists the inverse function index^{-1} : $\{0 \dots |V| - 1\} \rightarrow V$. Table 3.1 summarizes the *worst-case* computational complexity of the graph operations based on the assumptions that the graph is represented as a list of vertices and a adjacency list for each vertex.

Implementation Issues

In the following we discuss the implementation issues of the ordering graph which are necessary to derive the space and the computational complexity of the graph operations.

We assume that the index and the index^{-1} functions are implemented by using hash tables with the vertices as keys and the corresponding vertex numbers as values and vice versa. The adjacency list for each vertex is implemented as a linked list of vertex numbers.

Space Complexity of an Ordering Graph

Each vertex of the graph has a constant length defined by the size of the object and observer *id*s and the sequence number. Note that the state associated with each update request is not stored in the graph, because it is not needed to determine the ordering between state records. Therefore, the space required to store the array of vertices is in $\mathcal{O}(|V|)$. The size of the hash map is determined by the number of buckets m used in the hash map. Each bucket points to a pair of vertex and vertex-numbers. Thus the space complexity of the hash map used to implement the index-function is in $\mathcal{O}(m + |V|)$. The precise magnitude of m depends on the actual implementation od the hash map used. The worst case space complexity for one adjacency list is $\mathcal{O}(|V|)$ if the list of successors of a vertex v includes all vertices in V. Thus, the space needed to store all edges is at most $\mathcal{O}(|V|^2)$.

The previous discussion leads to a total *worst case space complexity* of the ordering graph which is in $\mathcal{O}(|V|^2 + m)$.

Computational Complexity of *addTransitive*

The *addTransitive* function presented in Algorithm 3 iterates all elements contained in V_d. Within this loop, the function first calculates the result of the functions selectFrom$(V_\mathrm{in} \setminus V_\mathrm{d}, E_\mathrm{in}, v)$ and selectTo$(V_\mathrm{in} \setminus V_\mathrm{d}, E_\mathrm{in}, v)$ for the given vertex v in V_d. The complexity of *selectFrom* depends linearly on the number of elements in the adjacency list of vertex v. For computing the result of the *selectTo* function, however, it is at most necessary to search all adjacency lists, i.e., its complexity is in $\mathcal{O}(|E_\mathrm{in}|)$. The complexity of the nested `for`-loops in lines 9-13 is in $\mathcal{O}(|V_\mathrm{af}| * |V_\mathrm{at}|)$ which, in general, is in $\mathcal{O}(|V_\mathrm{in}|^2)$. However, it holds that $|V_\mathrm{af}| + |V_\mathrm{at}| \leq |V_\mathrm{in}|$ since the graph is acyclic by construction.

In general, the complexity of the *addTransitive* function is in $\mathcal{O}(|V_\mathrm{in}|^3)$. If, however the number of vertices that are removed from the graph is constant (as in the *lossy-k-reduce* function), the complexity is in $\mathcal{O}(|V_\mathrm{in}|^2)$.

Computational Complexity of *lossless-reduce*

The `for`-loop in lines 4-17 of Algorithm 2 has a computational complexity in $\mathcal{O}(|V_{\text{in}}|*(|E_{\text{in}}|+|V_{\text{in}}|))$. The first term $|V_{\text{in}}|$ is caused by the `for`-loop starting in line 4 which iterates over all elements in V_{in}. The term $(|E_{\text{in}}| + |V_{\text{in}}|)$ is caused by the operations inside the loop. For each vertex in V_{in} the functions *selectFrom* and *selectTo* are executed. This leads to a complexity in $\mathcal{O}(|E_{\text{in}}|)$ as discussed previously. The second term $|V_{\text{in}}|$ in the complexity goes back to the nested `for`-loop in lines 9-13 which iterates over a subset of all vertices.

The `for`-loop in lines 19-23 executes in linear time with the size of the set V_{d} since the *high*-function can be pre-computed. The complexity of *addTransitive* is in $\mathcal{O}(|V_{\text{in}}|^3)$ here, since V_{d} may contain any fraction of the vertices in G_{in}. The complexity of *purgeEdges* is in $\mathcal{O}(|E_{\text{in}}|)$.

In summary, it can be concluded that the computational complexity of the *lossless-reduce*-function is in $\mathcal{O}(|V_{\text{in}}|^3)$.

Computational Complexity of *lossy-k-reduce*

The *getObservers*(V_{in}) function in line 4 of Algorithm 4 is linear in time with the size of V_{in}, if the list of all vertices is searched once. Therefore, the number of observers is in $\mathcal{O}(|V_{\text{in}}|)$. This defines how many iterations are done in the `for`-loop in lines 5-9. Inside that loop, the functions *getHighestSN*(V_{in}, o, k) and *getAll*(V_{in}, o) can be done in linear time depending on the number of vertices in V_{in} (optimizations permit to execute both functions in a single iteration). The computation in line 8 is also linear and depends on the size of the set V_{k}. Thus, the complexity of the complete `for`-loop is in $\mathcal{O}(|V_{\text{in}}|^2)$.

Under the assumption that the *lossy-k-reduce* function is used on graphs which contain at most $2*k$ vertices of the same observer, the size of the set V_{d} is limited to at most k elements which limits the complexity of the *addTransitive* function to $\mathcal{O}(|V_{\text{in}}|^2)$ as discussed in Section 3.4.6. The complexity of *purgeEdges* is in $\mathcal{O}(|E_{\text{in}}|)$.

Consequently, it can be concluded that the computational complexity of the *lossy-k-reduce*-function is in $\mathcal{O}(|V_{\text{in}}|^2)$.

Computational Complexity of *add* and *join*

Both, the complexities of the *add* and the *join* function, are dominated by the *reduce* function. This leads to an overall computational complexity of $\mathcal{O}(|V_{\text{in}}|^3)$ or $\mathcal{O}(|V_{\text{in}}|^2)$ for both functions depending on which reduce function is used.

Without using a *reduce*-function the complexity of the *join* function is in $\mathcal{O}(\max(|E_1|, |E_2|, |V_1|, |V_2|))$ for the computation of the set unions in lines 2 and 3 in Algorithm 5.

For the *add*-function without the application of a *reduce*-function the computational complexity is in $\mathcal{O}(|V_{\text{in}}|)$ for adding a new edge originating at each vertex in line 3 of Algorithm 1.

Computational Complexity of *occurredBefore*

The `if`-branch in lines 2-7 of Algorithm 6 is calculated in constant time, because it only consists of sequential comparisons and conditional branching.

In the `else`-branch of the algorithm, the functions $getVertexGE(V_{\text{in}}, o, i)$ and $getVertexLE(V_{\text{in}}, p, j)$ are both executed in linear time which depends on the size of the set V_{in}. Again, optimizations are possible in order to search the set only once for both functions. The function $hasPath(E_{\text{in}}, v_{\text{from}}, v_{\text{to}})$ in line 12 can be calculated using a breadth-first search in the graph in $\mathcal{O}(|V_{\text{in}}| + |E_{\text{in}}|)$ [Tur96].

In summary, the overall computational complexity of the *occurredBefore* predicate is therefore in $\mathcal{O}(|V_{\text{in}}| + |E_{\text{in}}|)$.

Chapter 4

Replication Algorithms

Given the consistency model discussed in Chapter 2, other issues need to be considered when designing replication algorithms that guarantee the consistency of the data in the first place: the availability of data for access and the cost of data access.

One major goal of data replication is to increase the *access availability* of the data [SS05]. In order to prevent data access failures caused by node or communication failures, multiple physical copies of the same logical data object are placed on a number of nodes in the system. How many copies to create and where to place them are therefore important considerations that have to be taken into account as properties of the concrete system model.

The *cost of data access* depends on how the operations defined on the data are implemented. The cost for executing one kind of operation, e.g. read operations, is in general not independent from the implementation of the other operations, such as update operations. When designing a particular replication algorithm a trade-off between the cost of the operations has to be considered in the context of the requirements of applications which operate on the replicated data.

Within this chapter, we present two replication algorithms that guarantee the consistency model presented in Chapter 2. The first algorithm performs a *full replication* on all nodes in the network that may read data. There are several advantages that support this approach:

- All read operations are executed locally at no communication cost.
- Reading is always possible.

This behavior is advantageous for applications where the ratio between read and update operations is high. Additionally, clients are able to read data even when a node is temporarily disconnected, for example in a partitioned network. However, these advantages come at a price. First, the cost for updating an object requires that update operations are sent and applied to all nodes in the network. Secondly, each node storing physical copies of the objects needs enough local storage in order to host the data.

The second replication algorithm presented in this chapter replicates the data objects only on a subset of nodes in the system in order to balance the ratio between the cost for read and update operations. On one hand, using this algorithm means that both, read and update operations, are in general remote operations that cause communication cost. On the other hand, not every node requires the memory capacity to store a copy of each data object.

4.1 Algorithm 1: Full Replication

The algorithm presented next fully replicates the state information of objects on all nodes that may perform read operations on the state information of objects. In terms of the system model presented in Chapter 2, all nodes that are in the role of a client are also in the role of a DB node and vice versa. Nodes in the role of an observer may either be solely in this role or the role of an observer may be combined with the other two roles on a single node. The possible combinations of roles per node are shown in Table 4.1. The algorithm creates one ordering graph for each information object.

4.1.1 Update Operations

An update operation is separated into three phases as depicted in Figure 4.1. In the first phase an observer detects a state change of a physical world object.

Table 4.1: Possible combinations of roles on a node for algorithm 1.

	Node Role		
	Observer	DB node	Client
Node type A	✓		
Node type B		✓	✓
Node type C	✓	✓	✓

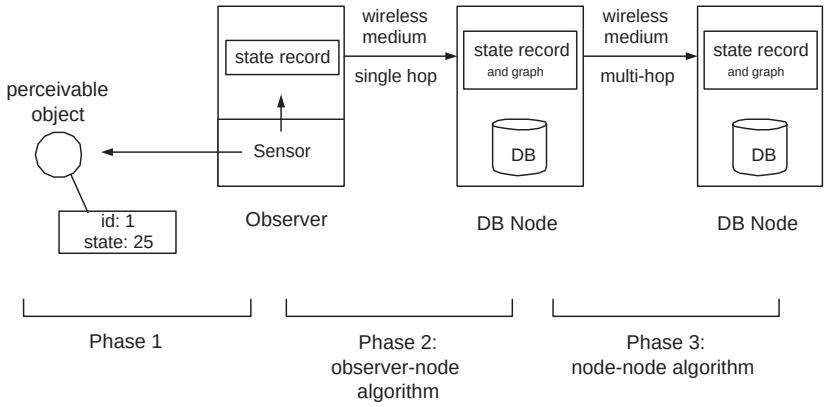


Figure 4.1: Overview of the update operation of Algorithm 1

Following, the observer may pre-process the captured data before it creates a state record. In the second phase, the observer sends a message that contains the state record to all nodes in single-hop distance using a layer 2 broadcast message. Given that a DB node receiving the message at time t_{rx} has not previously received another state record from a different observer in the time interval $[t_{\mathrm{rx}} - \delta_{\mathrm{obs}}, t_{\mathrm{rx}}]$ (line 2, Algorithm 7), it can now be concluded that this message contains a state record with the most recent state information about the particular object (see Figure 3.1, p. 48). Therefore, the DB node accepts the update request and writes the state record contained in the message to the local copy of the database replacing any old value. Additionally, the DB node updates the ordering graph of the corresponding object by inserting the new state record into the graph using the *add* function (see Algorithm 1, p. 52). If the condition in line 2 of the algorithm does not hold the received update request is discarded. In this case the chronological order of the two observations leading to the generation of the two update requests cannot be determined unambiguously. The algorithm used in the second phase is depicted in Algorithm 7.

Algorithm 7 Observer-node Algorithm (phase 2)

Require: incoming message sent by an observer

1: `on receive`$(\mathrm{u}[x]^{Obs}_{sn})$:
2: **if** $t_{\mathrm{rx}} - t_{\mathrm{last}}(x) > \delta$ **then**
3: {Update database and ordering graph}
4: $\mathrm{db}[x]^{O}_{S} \leftarrow \mathrm{u}[x]^{Obs}_{sn}$
5: $G_x = \mathtt{add}(G_x, \mathrm{u}[x]^{Obs}_{sn})$
6: `send`$(\mathrm{message}(\mathrm{u}[x]^{Obs}_{sn}, G_x))$
7: **end if**

In the third phase the DB node is responsible for synchronizing other DB nodes which are not in the communication range of the observer that created the new state record. For this phase the algorithm uses a flooding-based protocol which sends the new state record and the corresponding ordering graph to all other DB nodes in the system using the `send` primitive. A DB node receiving a message from another DB node executes Algorithm 8 in order to decide about accepting the state record.

The algorithm executed when receiving a message containing a state record $\mathrm{u}[x]^{Obs}_{sn}$ and an ordering graph G'_{x} first merges the ordering graph G'_{x} with the locally stored ordering graph G_{x} (line 3, Algorithm 8). Then a check is performed whether there already exists a state record for object x in the local database. If not (case N-1 in Algorithm 8) the received state record is stored. If a state record for object x has previously been stored in the local database the algorithm has to evaluate which one of the two state records — the one stored locally or the one just received — is more recent. If both state records were created by the same observer a comparison of the sequence numbers is sufficient for this decision (case N-2-1, lines 10-22). If the two state records were created by different observers then the decision to accept the received state record has to be made using the ordering graph. In particular, the *occurredBefore* predicate on the corresponding ordering graph has to be evaluated to make the decision (case N-2-2, lines 23-34).

In all situations where the algorithm accepts the received state record (cases N-1, N-2-1-1, and N-2-2-1), the newly accepted state record and the corresponding ordering graph are sent to all single-hop neighbors using the `send` primitive. If the algorithm does not accept a particular state record, this record is either older or the same than what the DB node has already stored in its database, or the information in the ordering graph is not sufficient to derive that the received record is younger than what is stored. Note that the state record and the ordering graph are also forwarded when the state record has not been accepted but the ordering graph has changed. This allows DB nodes to propagate new ordering information.

4.1.2 Read Operations

Read operations are done on the local copy of the objects. Therefore, a read operation on an object returns the state written by the last update operation that has been accepted locally.

Algorithm 8 Node-node algorithm (phase 3)

Require: incoming message sent by DB node

1: `on receive`$(\mathrm{u}[x]^{Obs}_{sn}, G'_x)$:
2: $G^{\mathrm{Old}}_x \leftarrow G_x$
3: $G_x \leftarrow$ `join`(G_x, G'_x) {join ordering graphs}
4: **if** $\mathrm{db}[x]^O_S = empty$ **then**
5: {case N-1: no state record in DB for object x}
6: $\mathrm{db}[x]^O_S \leftarrow \mathrm{u}[x]^{Obs}_{sn}$
7: `send`$(\mathrm{message}(\mathrm{u}[x]^{Obs}_{sn}, G_x))$
8: **else**
9: {case N-2: object x stored in DB: $\mathrm{db}[x]^O_S$}
10: **if** $Obs = O$ **then**
11: {case N-2-1: db record and update request from the same observer}
12: **if** $sn > S$ **then**
13: {case N-2-1-1: update request has higher sequence number }
14: $\mathrm{db}[x]^O_S \leftarrow \mathrm{u}[x]^{Obs}_{sn}$
15: `send`$(\mathrm{message}(\mathrm{u}[x]^{Obs}_{sn}, G_x))$
16: **else**
17: {case N-2-1-2: db record is more recent or the same}
18: **if** $G_x \neq G^{Old}_x$ **then**
19: `send`$(\mathrm{message}(\mathrm{db}[x]^O_S, G_x))$
20: **end if**
21: **end if**
22: **else**
23: { case N-2-2: different observers }
24: **if** `occurredBefore`$(G_x, \mathrm{db}[x]^O_S, \mathrm{u}[x]^{Obs}_{sn})$ **then**
25: { case N-2-2-1: }
26: $\mathrm{db}[x]^O_S \leftarrow \mathrm{u}[x]^{Obs}_{sn}$
27: `send`$(\mathrm{message}(\mathrm{u}[x]^{Obs}_{sn}, G_x))$
28: **else**
29: {case N-2-2-2: update request not accepted}
30: **if** $G_x \neq G^{Old}_x$ **then**
31: `send`$(\mathrm{message}(\mathrm{u}[x]^{Obs}_{sn}, G_x))$
32: **end if**
33: **end if**
34: **end if**
35: **end if**

4.1.3 Correctness

In this section, we first show that our algorithm is safe, i.e., we achieve update-linearizability according to Definition 2. Then, we show that our algorithm is live, i.e., every DB copy of an object eventually converges to the most recently propagated state if communication failures are not permanent.

In the proofs the reduction of the ordering graph does not need to be considered explicitly, because the *reduce*-functions presented in Section 3.4.2 (p. 53) guarantee that at least the most recent vertex of each observer remains in the ordering graph. Older ordering information that may be removed from graphs by a *reduce*-function can always be related to a newer vertex of the same observer in the graph that has a higher observer sequence number. These sequence numbers increase strictly monotonic locally on each observer.

Safety

First, we show that Definition 2 is fulfilled for a single copy. Let $S_{x,n}$ denote the sequence of update and read operations executed on the copy of object x stored on node n.

Claim: $S_{x,n}$ meets Definition 2.

Proof. First, we have to show that all read operations in $S_{x,n}$ are performed in the client's program order. Since each read operation in $S_{x,n}$ is requested by means of a local (synchronous) DB call, it is guaranteed that the execution order of reads corresponds to the program order.

Next, we show that all update operations in $S_{x,n}$ are performed in *occurred-before* order. More precisely, we show that once node n has accepted $\mathrm{u}[x]_j^k$, it will never accept an update $\mathrm{u}[x]_i^m$ if $\mathrm{u}[x]_i^m < \mathrm{u}[x]_j^k$.

For the observer-node protocol, we assumed that δ_{obs} is defined by the maximum communication delay jitter of a single-hop communication link as described in Chapter 2. Therefore, it is guaranteed that $\mathrm{u}[x]_i^m$ is not accepted after $\mathrm{u}[x]_j^k$ at any node. Since nodes perform update requests issued by observers in the order of their arrival, the observer-node protocol preserves the *occurred-before* order.

For the node-node protocol we have to consider two cases. If $k = m$, then $\mathrm{u}[x]_i^m$ and $\mathrm{u}[x]_j^k$ were created by the same observer. In this case, our sequence numbering scheme ensures that $\mathrm{u}[x]_i^m$ is not accepted once $\mathrm{u}[x]_j^k$ has been accepted as $i < j$ (case N-2-1-2, Algorithm 8). If $k \neq m$, then $\mathrm{u}[x]_i^m$ will only be accepted if the local ordering graph includes a path from $\mathrm{u}[x]_j^k$ to $\mathrm{u}[x]_i^m$ (case N-2-2-2). However, since no node receives $\mathrm{u}[x]_j^k$ before $\mathrm{u}[x]_i^m$, no ordering graph will ever include such a path. Consequently, the node-node protocol also preserves the *occurred-before* order. □

To show that our algorithm also fulfills Definition 2 for all copies of an object, we have to consider serializations of the read and write operations performed on all copies of a given object.

Claim: For each object x, there exists a serialization S_x of all read and update operations on x that fulfills Definition 2.

Proof. Let $SS_x = \{S_{x,n} \mid n \text{ is a DB node}\}$. Each $S_{x,n}$ in SS_x can be divided into segments, one for each update operation in $S_{x,n}$ and the succeeding read operations. Without loss of generality, let $S_{x,n}$ include the following sequence: $\cdots u[x]_k; r[x]_{k+1}; \cdots; r[x]_{k+m}; u[x]_{k+m+1}; \cdots$. Then the segment $seg(u[x]_k)$ of $u[x]_k$ is defined to be $u[x]_k; r[x]_{k+1}; \cdots; r[x]_{k+m}$ $(k \geq 0)$. S_x can be constructed by merging the segments of sequences in SS_x according to the *occurred-before* order. In other words, for any two segments $seg(u[x])$ and $seg(u'[x])$ in SS_x $seg(u[x])$ must have *occurred-before* $seg(u'[x])$ in S_x if $u[x] < u'[x]$, and in any order if $u[x]||u'[x]$. This is possible, because the occurred-before relation is a partial order. □

Liveness

In this section we show that the DB copies of an object converge to the most recently observed state if communication failures are not permanent.

Claim: For each object x all available copies of x eventually receive and accept an update operation $u[x]$, where $\neg\exists u_k[x] : u[x] < u_k[x]$.

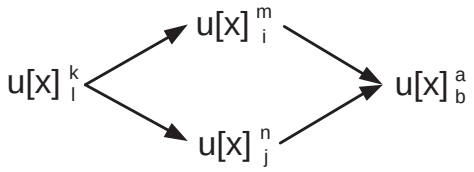


Figure 4.2: Example graph after the ambiguity in the ordering has been resolved by the algorithm.

Proof. We start with the assumption that all update requests reach every DB node with a finite latency. Furthermore we assume that there exists only one observer m that updates a particular object x.

Let $\mathrm{u}[x]_j^m$ be the most recent update request that has been created by m for x. If all previous update requests $\mathrm{u}[x]_i^m$ with $1 \leq i < j$ have been delivered to all DB nodes before $\mathrm{u}[x]_j^m$, the DB nodes will accept $\mathrm{u}[x]_j^m$ because j is currently the highest sequence number of m in the system (Algorithm 8, case N-2-1-1). In those cases where any update request $\mathrm{u}[x]_i^m$ with $i < j$ is delayed and therefore delivered to a DB node after $\mathrm{u}[x]_j^m$, the update request $\mathrm{u}[x]_j^m$ has already been accepted and $\mathrm{u}[x]_i^m$ will be dropped (Algorithm 8, case N-2-1-2).

Next, we discuss the case when more than one observer exist that create update requests for object x. Without loss of generality, assume that there exists a pair of observers m and n, and $\mathrm{u}[x]_i^m$ and $\mathrm{u}[x]_j^n$ are the latest updates transmitted by m and n with $\mathrm{u}[x]_i^m < \mathrm{u}[x]_j^n$. If at least one DB node receives $\mathrm{u}[x]_j^n$ from n after having learned about $\mathrm{u}[x]_i^m$, this node *add*s edge $(\mathrm{u}[x]_i^m, \mathrm{u}[x]_j^n)$ to its local ordering graph and forwards it. Therefore, each node eventually accepts $\mathrm{u}[x]_j^n$.

In the presence of communication failures, however, messages may be lost and therefore a DB node that has edge $(\mathrm{u}[x]_i^m, \mathrm{u}[x]_j^n)$ in its ordering graph may not exist. Then, a portion of the nodes may end up with $\mathrm{db}[x]_i^m$ and another one with $\mathrm{db}[x]_j^n$, and each node's ordering graph G_x eventually includes vertices $\mathrm{u}[x]_i^m$ and $\mathrm{u}[x]_j^n$ without an ordering relationship defined between them.

Without loss of generality, we assume that update requests $\mathrm{u}[x]_l^k < \mathrm{u}[x]_i^m < \mathrm{u}[x]_j^n$ are forwarded and that all nodes accept $\mathrm{u}[x]_l^k$. Further we assume that node n_1 accepts $\mathrm{u}[x]_i^m$ and misses $\mathrm{u}[x]_j^n$ while node n_2 misses $\mathrm{u}[x]_i^m$ and accepts $\mathrm{u}[x]_j^n$. This condition, where different nodes hold different state information for the same object, is resolved with the next update request $\mathrm{u}[x]_b^a$ which will be ordered as the youngest update by the node receiving it from j and adding it to its graph. After this update, the previously unordered update requests $\mathrm{u}[x]_i^m$ and $\mathrm{u}[x]_j^n$ are ordered as preceding $\mathrm{u}[x]_b^a$. This situation is depicted in the Graph in Figure 4.2. See Section 4.4 for the discussion on periodic update request.

□

4.2 Algorithm 2: Partial Replication

In contrast to the first algorithm, the replication algorithm presented next does not replicate the information on all nodes [HBMR06]. Instead, a sub-set of all nodes is chosen to become DB nodes. Each of these nodes holds a copy of all objects. As in the first algorithm, observers send a state record for each observation they make to DB nodes in single-hop distance. If no DB node is available in single-hop distance to an observer, a client is elected by the observer to become a temporary DB node (TDB node). TDB nodes have to at least maintain ordering graphs to support the creation and maintenance of ordering information in the system. For the sake of simplicity we assume in the following that they also store the most recent state for each object. Therefore, TDB nodes have the same tasks as DB nodes except that they may change their role back to only being a client under certain conditions described later. Clients read the state of objects by sending unicast messages to a DB node. A client may choose a DB node that is, for example, closest to it. If the communication between the client and its DB node fails, the client may choose a new DB node to read from using the server selection algorithm. The client is conceptually composed of an application that reads object information and a database sub-system that handles remote communication with DB nodes if necessary.

4.2.1 Server Selection Algorithm

Whenever a client needs to (re-)select a DB node to read from, it uses the server selection algorithm. We first show the basic server selection approach followed by implementation aspects. Then we outline some optimizations that can be easily included into the basic approach.

Basic Server Selection

To guarantee update-linearizability the new DB node has to provide state information to the client not older than what the client has read so far. More formally, it has to fulfill the following criteria. Let $O_c = \{x_1, x_2, \cdots, x_n\}$ be

the set of objects that the client has read at least once. For every object $x_k \in O_c$ the client maintains the tuple (k, Obs_k, sn_k) that contains the object identifier, the observer identifier and the sequence number of the observer whose update request it has read last. The set of all tuples is called *client read snapshot*. The client may use a DB node if for every object $x_k \in O_c$ the following holds: the object information last read by the client must be the same or must have *occurred-before* the state information of the DB node's copy of the state information. Thus, the correctness of the server selection is based on the correctness of the *occurred-before* operation. Note that a client that has not performed a read before may be served by an arbitrary DB node, because clients have an empty read snapshot on initialization.

Implementation Aspects

To find an appropriate DB node, the client sends a message that contains its client read snapshot and a client sequence number[1] to one or more DB nodes. This may be achieved by, for example, sending a multicast message m to all DB nodes using `multi-send`(DBGRP, m). In turn, each receiving DB node checks whether it fulfills the consistency criteria needed to serve the client. This is achieved by evaluating the *occurred-before* predicate for each entry of the client read snapshot and the corresponding state record the DB node has stored. A DB node only sends a reply message, if it is able to serve the client according to the criteria above. The reply message sent from the DB node to the client contains the client sequence number from the request message in order to match the reply and the read snapshot. If the client receives multiple replies, it may, for example, choose the DB node whose reply was received first for future read operations.

The client may not find a subsequent server to read from for two reasons: due to communication failures or because the DB node that served the client previously crashed. In the former case the client periodically retries server selection until the connectivity has been restored and an appropriate DB node

[1] Client sequence numbers are strictly monotonic increasing at each client and are used to associate, for example, read and read-reply messages. Sequence numbers as part of update requests are those maintained by observers.

can be found. In the latter case a DB node that is able to serve the client may not exist, because the client read snapshot may not match the current state of any DB node. In this case the client also periodically repeats the server selection procedure, because eventually some DB node will update its copy of the database accordingly and the client will receive a positive reply to its next server selection See Section 4.4 for the discussion on periodic update request.

Optimizations

The basic approach may be optimized by allowing clients to read different objects from different servers. This may be for example advantageous if the read snapshot of a client is large. Given that, for example, a client has read objects x, y, and z, there may be no server that suits the client's needs by providing recent versions of all objects. There may, however, be servers that can serve the client consistently with a subset of these objects. To use this optimization the implementation outlined before needs to be modified in such way that each server reports to the client which subset of the objects contained in the read snapshot it can serve. The client may then select multiple servers to read from.

Some applications may not require to read all objects consistently at all times. Therefore, the application developer may explicitly specify when the consistency requirements for a particular object can be released. On releasing an object, it is simply removed from the client read snapshot. As a consequence the requirements for finding servers to read from are weakened.

Example for the Server Selection

Figure 4.3 shows an example of the message sequence when a client sends a request to two available DB nodes. As show in Figure 4.4, DB node 1 is not able to fulfill the clients request, because the values for objects y and z which the client has read are more recent than what is stored in the local database of DB node 1. In contrast to that, DB node 2 is able to fulfill the request of the client, because all values stored in the local database of DB node 2

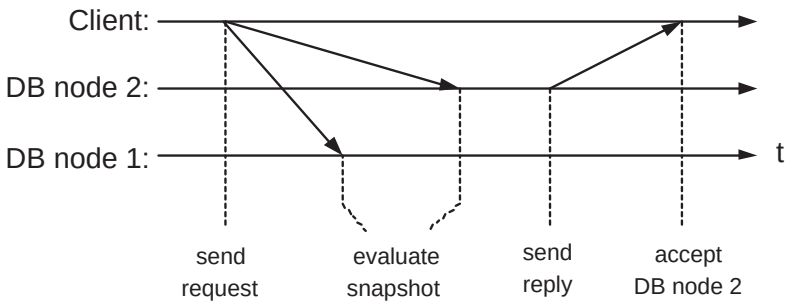
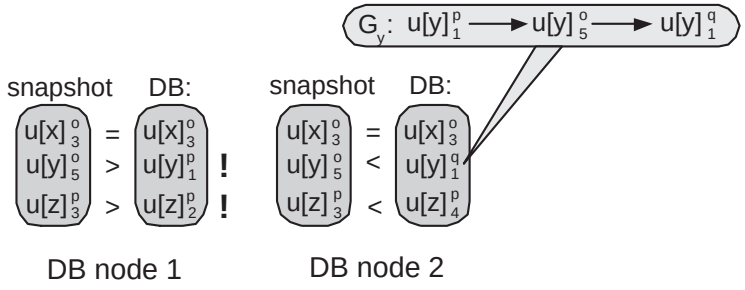


Figure 4.3: Example of the message sequence when selecting a server



$$H_{\text{global}} = (\ldots, \mathrm{u}[y]_1^p, \mathrm{u}[x]_3^o, \mathrm{u}[y]_4^o, \mathrm{u}[x]_5^o, \mathrm{u}[z]_2^p, \mathrm{u}[y]_1^q, \mathrm{u}[z]_3^p, \mathrm{u}[x]_2^q, \mathrm{u}[z]_4^p, \ldots)$$

Figure 4.4: Examples of DB nodes evaluating a client read snapshot

are either the same values or more recent than what the client has last read. These conclusions can be reached by either comparing sequence numbers of state records (if both records were created by the same observer) or by evaluating the *occurredBefore* predicate on the ordering graph corresponding to the object. H_{global} in Figure 4.4 shows one possible (global) order in which the observations could have taken place.

4.2.2 Dynamic Replica Allocation Algorithm

The update algorithm derives ordering information between update requests based on the assumption that at least one DB node is in the transmission range of each observer. If at least one client is in the (single-hop) transmission range of an observer, the replica allocation algorithm ensures that one of the clients present becomes a temporary DB node. Therefore, observers periodically send single-hop messages to indicate their presence, which may be piggybacked on update messages. Every DB node in the transmission range of the observer replies with a *DB node available* message. If at least one such message is received by the observer, it concludes that at least one DB node is present and available for processing update requests. The number of reply messages sent by DB nodes may be reduced by a scheme similar to counter-based flooding [NTCS99]. If no DB node replies, the observer concludes that there is no DB node in its communication range. Then, the observer sends an election message in order to elect one of its neighbor client nodes as a new TDB node. Each available client node replies with a message in order to advertise its presence. This message includes some appropriate parameters as election criteria, such as the amount of memory or energy available at the client node. The observer then addresses the best suited client node to become a TDB node, for example, the one with the largest amount of memory available. The new TDB node creates a local database and starts replicating object information by joining the group of DB nodes and accepting update messages. To speed up the replication progress the TDB node may request the ordering graphs from another DB node.

In mobile networks, the dynamic replica allocation algorithm will in general lead to an increasing number of replicas (TDB nodes) over time. In

order to keep the number of replicas low, a TDB node may change its role and drop its copy of the database under one of the following conditions: when no observer is in the single-hop communication range of the TDB node or whenever there is more than one (T)DB node in the communication range of an observer. The first condition can be detected by the TDB node, if no observer announces its presence for a given time. In the second case TDB nodes may negotiate which ones – for example the ones with the smallest amount of memory available – drop their copy in order to have one copy remaining in the communication range of the observer. If a read request arrives after the TDB node has dropped its copy, the client treats this as a DB node failure (cf. read algorithm).

4.2.3 Update Operations

The phases of the algorithm are similar to those of the algorithm fully replicating the data as depicted in Figure 4.1. When an observer senses a state change of a perceivable object it sends a state record to all its neighbors. All DB nodes among the neighbors will then act according to Algorithm 7, except that the call of the `send` primitive in line 6 is replaced with `multi-send`(`DBGRP`, message(u$[x]_{sn}^{Obs}, G_x$)). With this change, the message is sent to all DB nodes in the system which are assumed to have joined the (well-known) multi-cast group `DBGRP`.

A DB node receiving a state record (from another DB node) via a multi-cast message executes the algorithm shown in Algorithm 9. The algorithm is similar to Algorithm 8. However, here incoming messages are not relayed by the receiving DB node. The dissemination of the state records in the system is handled by the underlying multi-cast algorithm.

4.2.4 Read Operations

In comparison to the first replication algorithm the partial replication algorithm requires that read operations need to be executed on remote DB nodes. In condition (C1) of *update-linearizability* it is required that all read operations of a client on a single object are executed in the program order of the

Algorithm 9 Algorithm for processing incoming update requests

Require: incoming message sent by a DB node

1: `on receive`$(\mathrm{u}[x]^{Obs}_{sn}, G'_x)$:
2: {join local and received ordering graphs}
3: $G_x \leftarrow$ `join`(G_x, G'_x)
4: **if** $\mathrm{db}[x]^O_S = empty$ **then**
5: { case N-1: no state record in DB for object x }
6: $\mathrm{db}[x]^O_S \leftarrow \mathrm{u}[x]^{Obs}_{sn}$
7: **else**
8: { case N-2: object x stored in DB: $\mathrm{db}[x]^O_S$ }
9: **if** $Obs = O$ **then**
10: {case N-2-1: db and incoming record are from same observer}
11: **if** $sn > S$ **then**
12: {case N-2-1-1: update request has higher sequence number}
13: $\mathrm{db}[x]^O_S \leftarrow \mathrm{u}[x]^{Obs}_{sn}$
14: **end if**
15: **else**
16: {case N-2-2: different observers}
17: **if** `occurredBefore`$(G_x, \mathrm{db}[x]^O_S, \mathrm{u}[x]^{Obs}_{sn})$ **then**
18: {case 2-2-1: received record is more recent}
19: $\mathrm{db}[x]^O_S \leftarrow \mathrm{u}[x]^{Obs}_{sn}$
20: **end if**
21: **end if**
22: **end if**

client. Abstractly, this can be achieved by providing FIFO communication between the client and the DB node in both directions of communication.

For the sake of simplicity we first describe the read algorithm where a client may only execute one read operation at a time. This blocking read operation provides FIFO ordering. However, to improve the performance of read operations the client may handle a number of read operations concurrently, either by bundling several read operations in a single read request message or by sending multiple read request messages before a reply has been received. In order to maintain the program order, the client then has to order the incoming read reply messages accordingly. Our approach requires that only the client needs to maintain information about the state of its own read operations. The DB nodes do *not* have to maintain state information regarding read operations clients sent to them.

If a client requests to read an object for which the DB node does not store any state, an error is reported by the DB node. In the following we assume that the client has selected a DB node to read from, for example, by using the server selection algorithm described in Section 4.2.1.

Blocking Read Operation

To read the state of an object the client first sends a read request message to the DB node using the `uni-send` primitive. The message contains the *id* of the object for which the client wants to read the state and a *client sequence number*. Next, the client blocks until it receives a read reply message from the DB node or until a timeout occurs. If the client does not receive a read response message after a timeout period, it may retransmit the read request up to a maximum number of retries. If all retries fail, it may use the server selection algorithm to select a new DB node.

When the DB node receives the read request, it sends a read response to the client which contains the state record of the requested object and the client sequence number from the request. When the client receives a read response, it uses the client sequence number to detect duplicate messages. Next, it updates its read snapshot with the observer *id* and observer sequence

number from the state record contained in the response. Finally, the state of the object is delivered to the application.

Concurrent Read Operations

One advantage of the blocking read operation is the fact that it is easy to implement. However, one disadvantage is the aggregated latency of multiple read operations executed sequentially: each read operation has a latency of at least the round-trip time (RTT) of the communication between client and DB node. In order to reduce the latency of successive read operations, the client may send multiple read requests concurrently. This means that the client is split into two concurrent threads of control: one contains the application itself and the other thread contains the database sub-system. Figure 4.5 shows two programs that may benefit from read operations that are executed concurrently. We assume that the program is executed until the first dependency on a read operation that is not satisfied with a read reply is reached. If such a dependency is reached, the processes must be synchronized by using, e.g., mechanisms similar to future objects in remote procedure calls, e.g., [Cha89], which block on accessing a result which has not been delivered yet.

The first example in Figure 4.5(a) starts by reading object x. Based on a (local) condition the program reads either object y or z next. If the blocking read operation is used when executing this program, the execution time would be at least two times the RTT of the communication link between client and DB node. However, when the read operations are started concurrently the execution time can be reduced to a value between RTT and 2∗RTT, because execution of the program would only block if $value1$ and $value2$ are not yet delivered and needed for processing (since the variable *condition* is a local variable with no dependencies to database objects).

The program in Figure 4.5(b) shows an example that reads the same object x periodically in order to detect changes of the object's state (e.g., by calculating the difference). If the blocking read operation is used for this program, it would only be possible to detect changes with a frequency of less than $\frac{1}{RTT}$ (the sleep function would be omitted then). If the concurrent read

```
bool condition {local variable}
value1 = read(x)
if condition then
   value2 = read(y)
else
   value2 = read(z)
end if
{process value1 and value2}
```

(a) Example where explicit combination of read operations is difficult

```
int diff = 0
while diff == 0 do
   x_1 = read(x)
   sleep t_min
   x_2 = read(x)
   diff = x_2 − x_1
end while
{signal change diff}
```

(b) Example program where the same object may be read concurrently

Figure 4.5: Example programs relevant for remote read operations

operation is used it is possible to detect changes that have a higher frequency. If the jitter of the (multi-hop) one-trip time in the network was 0 it would be possible to detect changes with a frequency of $\frac{1}{t_{\min}}$.

Implementing Concurrent Read Operation

The first implementation of concurrent read operations can be done explicitly by the application programmer by bundling these read operations into a *batched read* operation by passing a list of object *id*s to the database sub-system. As a result, the database sub-system then sends a single read request to the DB node listing all object *id*s that the client wants to read. The DB node then sends a single reply message to the client.

A second approach to increase concurrency of read operations from a single client is to send successive read requests before the reply messages to the previous requests have been received. However, if the client issues two concurrent operations reading the *same* object, the program order of the client may be changed if a reordering of messages occurs in the network[2]. If both requests for the same object were sent concurrently, either the requests, the replies, or both could be reordered in the network. Additionally, an update may be accepted on the DB node between the processing of the two read requests. Finally, any read request or read reply message may be lost due to (temporary) communication failures.

[2]Recall that read requests and replies may be sent over multiple hops in the network.

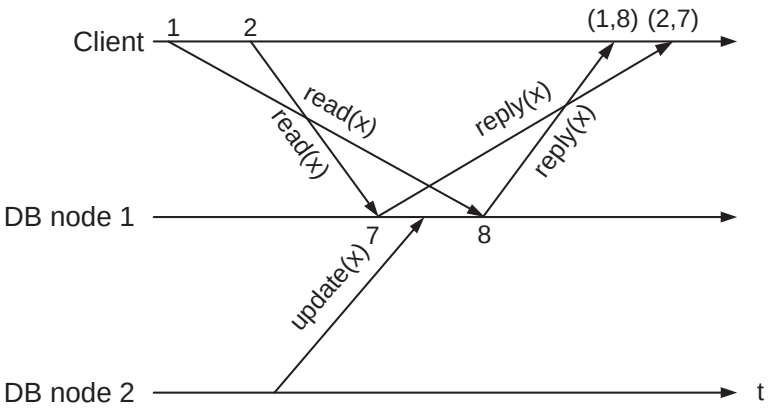
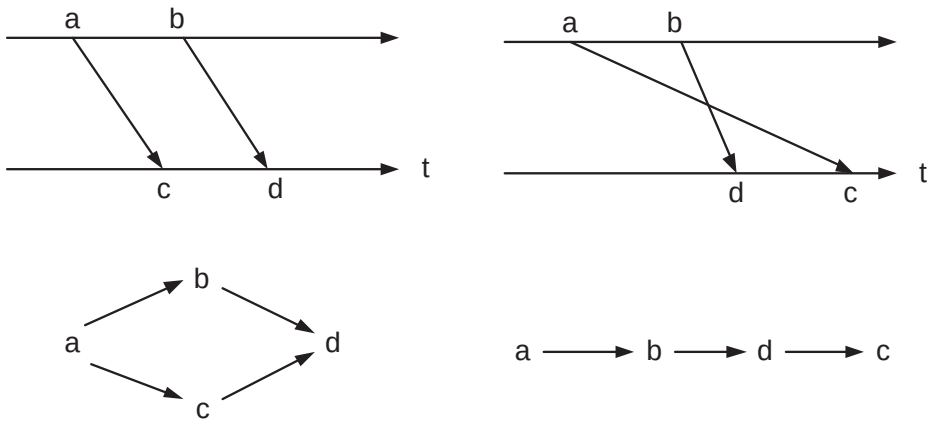


Figure 4.6: Possible re-ordering when reading the same object

An example for this situation is given in Figure 4.6 where object x is concurrently read twice by the application. In the time interval between processing the two read requests, the DB node accepts an update for object x. This means that the first reply message sent (associated with the second request) contains the "old" value (prior to the update) for x and the second reply message sent contains the "new" value (after the update) for x. Additionally, the order in which the reply messages are received is again changed.

To solve this problem we need to verify that two read requests were not reordered in the network before they are received by a DB node. This means that the event of sending the second read request is causally unrelated to the processing of the first read request on the DB node. Figure 4.7 depicts the two relevant cases where messages are delivered in order and where messages are reordered. In the figure, events a and b represent the sending of read requests and events c and d represent the processing of the read requests. In Figure 4.7 (a) the events b and c are causally unrelated which indicates that the messages were delivered in order. In contrary, in Figure 4.7 (b) there is a causal dependency between events b and c from which the reordering can be derived.

The causal independence between events b and c can be detected by using *server sequence numbers* issued by the DB node in addition to the client



(a) Events b and c are causally unrelated (b) Event b causally precedes event c

Figure 4.7: Causal dependencies: events a and b are read requests, c and d are read replies

sequence numbers. Each DB node therefore assigns strictly monotonic sequence numbers to each read request (from any client) it processes. Formally, the client receives a pair of sequence numbers $(sn_{\mathrm{c}}, sn_{\mathrm{s}})$ with each read reply. This pair of sequence numbers can be interpreted *similar* to a timestamp of a vector clock [Mat89].

In the example in Figure 4.6 the client assigns the sequence numbers 1 and 2 to the read requests in the order they are sent. When receiving the requests the DB node assigns server sequence numbers, e.g., 7 and 8, to the requests. The read replies sent back to the client will therefore contain pairs of sequence numbers $(1, 8)$ and $(2, 7)$.

The client now compares two received tuples by doing a component-wise comparison, i.e.

$$(sn_{\mathrm{c}}, sn_{\mathrm{s}}) < (sn'_{\mathrm{c}}, sn'_{\mathrm{s}}) \Leftrightarrow (sn_{\mathrm{c}} < sn'_{\mathrm{c}}) \wedge (sn_{\mathrm{s}} < sn'_{\mathrm{s}}) \tag{4.1}$$

Given that a client first receives a read reply message m_{rp1} that contains the tuple of sequence numbers $(sn_{\mathrm{c1}}, sn_{\mathrm{s1}})$ and then a read reply message m_{rp2} that contains the tuple of sequence numbers $(sn_{\mathrm{c2}}, sn_{\mathrm{s2}})$ we derive three cases that have to be distinguished:

1. *Reads are correct and in order*: If $(sn_{c1}, sn_{s1}) < (sn_{c2}, sn_{s2})$ holds, the client can conclude that the request and reply messages were sent in order. This means that the read operations were processed and received in the program order of the client.

2. *Reads are correct and out of order*: If $(sn_{c2}, sn_{s2}) < (sn_{c1}, sn_{s1})$ holds, the client can conclude that the request messages were received by the DB node in order, but the reply messages were received out of order by the client.

3. *Read violation of the first read*: If both tuples are unrelated, the two read request messages were reordered in the network and have *not* been processed in the program order of the client.

Next, we discuss how read reply messages are processed at the client. We assume that the client executes a *sequence* of k concurrent read operations $S = (r_1, \ldots, r_k)$. The sequence is concurrent, if the read request of r_k has been sent by the client before the read reply of r_1 has been received by the client.

Since the consistency model (cf. Definition 2, p. 40) is an object-local model, the order in which read operations of different objects are processed on the DB node is not critical for the correctness. However, if two or more read operations are reading the same object, the ordering of the corresponding requests and responses is important to maintain the program order of the client. Without loss of generality, let S be a finite concurrent sequence of n read operations that read only one object x (for example the program depicted in Figure 4.5(b)). This means that the client receives at most n reply messages (since messages may be lost) which may be delivered to the database sub-system in any permutation. The spectrum of processing the reply messages and consistently deliver data to the application now ranges from using the first reply only to waiting until all reply messages have been received[3]. Each of these approaches is correct with respect to Definition 2,

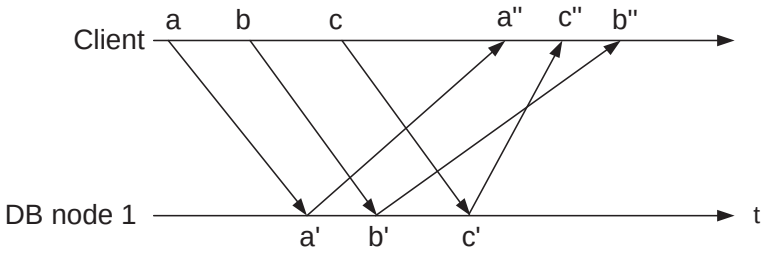
[3]Since our system is based on best-effort communication, the latter approach is combined with a timeout: wait until all replies have been received or the timeout expires

however they make a trade-off between latency and the granularity of data access. In the following we describe three approaches of the spectrum:

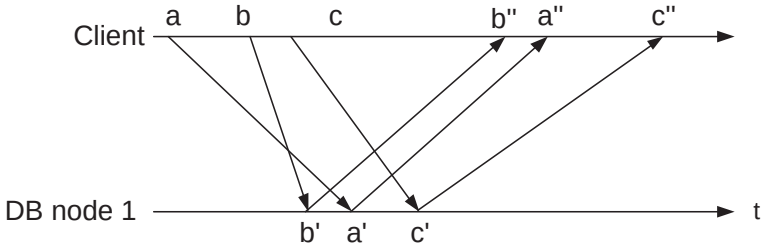
1. *Deliver the first read reply*: The first reply that the client receives is delivered for all read operations. This approach has low latencies with a trade-off to a coarse granularity of results .

2. *Process every reply as it arrives*: The first read reply that arrives at the database sub-system of the client with the tuple $(c_{i(1)}, s_{j(1)})$ of client and server sequence numbers is delivered straight away to satisfy the read operation with the sequence number $c_{i(1)}$. In the following, the database sub-system maintains a state of which tuple $(c_{i(k)}, s_{j(k)})$ has been delivered to which read operation. k denotes the k-th reply message that has been received. The functions $i(k)$ and $j(k)$ return the client and the server sequence numbers of the k-th reply message, respectively.

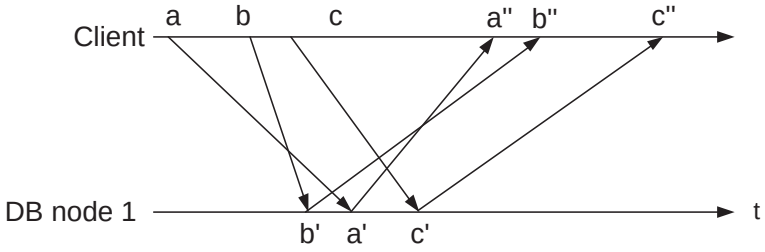
   For all other reply messages that include $(c_{i(k)}, s_{j(k)})$ with $2 \leq k \leq n$, the database sub-system compares the newly received tuple of sequence numbers with those already delivered to the application according to the relation defined for tuples of sequence numbers. If $(c_{i(k)}, s_{j(k)})$ is correct and in order with all previously delivered tuples, the incoming reply will be delivered for the read operation with the client sequence number $c_{i(k)}$. If any reply message with a client sequence number between $c_{i(k-1)}$ and $c_{i(k)}$ was not delivered, i.e., $c_{i(k)} - c_{i(k-1)} > 0$, the result for the reply $(c_{i(k)}, s_{j(k)})$ will be delivered for these missing read requests as well. If the received reply violates the read order, it will be dropped. If the last reply of the sequence is not received after a timeout, the last read operation will return the result of the last reply that has been received correctly. An example of a situation where a reply is received out of order is illustrated in Figure 4.8(a). The DB node receives the read requests in FIFO order. At the client, however, the reply messages b'' and c'' are received out of order. In this case, the DB sub-system delivers the reply c'' as a result to the read request b and c while b'' is dropped on arrival. In Figure 4.8(b) the read requests are



(a) The requests are delivered in FIFO order, the replies are delivered out of order: When the client receives reply c'', it delivers this reply to requests b and c. The reply b'' is dropped.



(b) The requests are delivered out of order, the replies are in FIFO order: The DB node replies in the order in which it receives the requests. The client accepts b'' as a reply to request a and b and drops a''. Reply c'' is accepted for c.



(c) The requests and replies are delivered out of order: The client will accept a'' as the reply to a. Reply b'' is rejected, since it violates the read order. Reply c'' is then used for answering reads requests b and c.

Figure 4.8: Examples for using the "process every reply on arrival" strategy.

delivered out of order, while the replies are delivered in FIFO order. Here, the reply b'' is accepted to answer the requests a and b. The reply a'' is then dropped. In Figure 4.8(c) the requests a and b and the corresponding replies are delivered out of order. In this example, reply a'' will be used to answer request a. Reply b'', however, is dropped, since it violates the read order. Instead, reply c'' is used to answer the read requests b and c.

3. *Deliver when all replies are received*: When all reply messages are received the database sub-system sorts them according to the server sequence numbers, i.e., the order in which the requests were processed by the server. The reply with the smallest server sequence number will then be delivered to the first read operation in the sequence, the reply with the next server sequence number will be used as a result to the second read operation and so on. By waiting until all replies are received the database sub-system can match the program order of the client with the processing order of the DB node and thus satisfy the consistency model.

Continuous Read Operations

If an application on a client needs to read the same object x multiple times, a continuous read operation `readcont`(x, listen_x) may be used by the programmer. With a *continuous read* the client registers at the DB node to be notified about all changes made on the state of object x. Every time the DB node receives an update for x it sends a read reply message containing the new state to the client as depicted in the message sequence shown in Figure 4.9. On receiving a read reply associated with a continuous read operation the database sub-system uses the call-back $listen_x$ to notify the application of the new state. To preserve the correct ordering of multiple and duplicated read replies the database sub-system uses the server sequence numbers. Both, duplicated replies and replies received out of order are not delivered to the application.

While the read operations described earlier do not require that the DB

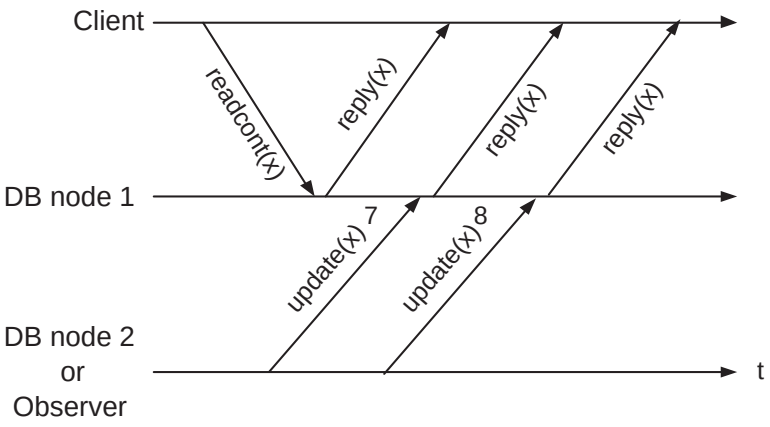


Figure 4.9: Example of the message sequence when using a continuous read operation

node keep any state for its clients, the continuous read operation requires that the DB node keeps track of all continuous read operations that clients have registered. To avoid unnecessary state and message overhead we use a lease mechanism which requires that the client periodically refreshes the continuous read at the DB node. If the DB node does not receive a refresh message for some time it removes the continuous read from its list and does not send any read replies on changes any more.

Comparison of the Read Operations

The blocking read operation is easy to implement and to use. However, for each successive read operation in an application program, the execution time of the application grows linearly with the number of read operations. The *batched* concurrent read operations ease this disadvantage in performance. However, these require that the application programmer explicitly combines read operations that may be sent in a single read request message. The concurrent read operations can provide higher performance than the blocking read operation. Additionally, they do not require the application programmer to explicitly specify concurrency as it is required using the batched read operations. Nevertheless, the implementation of the database

Table 4.2: Comparison of the read operations with respect to statefull DB nodes, programming support required, complexity of the database sub-system, and latency

	state at DB node	explicit use by programmer	complexity of DB sub-system	latency
blocking	no	no	low	high
batched	no	yes	medium	low
concurrent	no	no	high	low
continuous	yes	yes	medium	low

sub-system supporting concurrent read operations is more complex than for the approaches previously described. For many applications that repeatedly read the same object, the continuous read operation constitutes an alternative that may reduce the overall message overhead, because less read requests have to be sent to the DB node. However, the implementation requires that state information is stored at the DB node for continuous reads used by any client. Table 4.2 summarizes the comparison of the different read operations.

4.2.5 Correctness

In this section we outline why the algorithm for partial replication is correct according to the definition of update-linearizability. For a discussion of the safety and the liveness (convergence) of the update algorithm we refer to Section 4.1.3 where all nodes are updated. Here we will focus on showing the correctness of the read algorithms.

Correctness of the Read Algorithm

For all read operations we can assume that the DB node that serves a client is correct with respect to the consistency model. In order to show that the read operations are correct, we need to verify that the replies are delivered according to the order in which the requests were sent (the program order

of the client). The blocking read operation is correct since only one read operation is allowed to be pending at any time. Duplicate reply messages are eliminated by using client sequence numbers.

In the class of concurrent read operations, the correctness is based on the comparison of client and server sequence numbers. The strategy, which delivers the first read reply for a sequence of requests, is correct, since any one of the read replies is used to answer all requests of a sequence. This is correct with respect to Definition 2, since using only one reply does not impose any ordering ambiguities.

The correctness of the approach that processes every reply on arrival can be shown by examining all classes of ordering problems: requests and replies are delivered in FIFO order (case I), requests are in FIFO order and replies are out of order (case II), requests are out of order and replies are in FIFO order (case III), and requests and replies are out of order (case IV). The latter three cases are depicted in Figure 4.8. In case I the program order of the client is respected. This situation is detected, when all tuples of sequence numbers are received in increasing order according to the definition in Equation 4.1. If this holds, the requests are processed and the replies are delivered in the order in which they were sent by the client (the program order). In case II (Figure 4.8(a)) the requests are processed in the program order of the client. However, the replies are delivered out of order. In this case, a read operation is answered by a reply which belongs to a request which is later in the sequence of all requests. This is correct, because the reply to answer the request is the same as the reply used to answer the succeeding request. In case III (Figure 4.8(b)) the requests are not processed in the client's program order. On receiving the replies the client drops one of the replies which has been processed out of order and uses the other to answer both requests, which is correct. In case IV (Figure 4.8(c)) the request and the reply are delivered out of order. When the second reply arrives, this is detected by comparing the tuples of sequence numbers. The second reply is dropped and is answered by the next correct reply.

The approach "deliver when all replies are received" is correct, because it matches the order in which requests are processed by the DB node to the

program order by sorting the replies according to the sequence numbers of the DB node. Replies which are not received by the client are substituted by consecutive replies. The continuous read operation is correct, because the replies are processed in increasing order of the DB node sequence numbers. Duplicate and out of order replies are dropped.

4.3 Transmission of the Ordering Graph

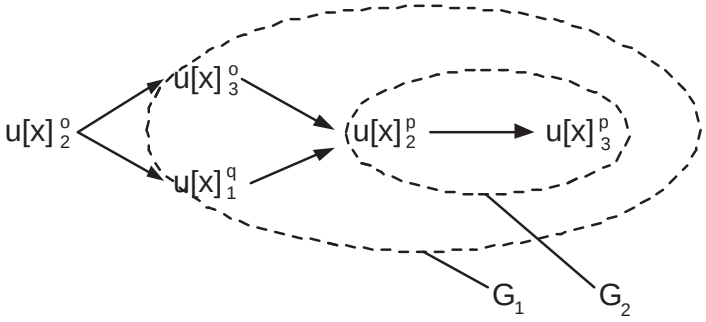
The presented replication algorithms need to exchange ordering information among DB nodes. This is necessary to synchronize the ordering information available in the system. One approach is to transmit the complete ordering graph. In addition to the reduction functions on a local ordering graph presented in Section 3.4.2 (p. 53), it is also possible to select subgraphs for exchange without reducing the local ordering graph and thus save bandwidth. This is in general possible, if the subgraph that is transmitted has vertices which are also contained in the graph maintained at the receiver. The subgraph G_1 depicted in Figure 4.10(a), for example, connects to the graph G_3 shown in Figure 4.10(b), because G_1 and G_3 share common vertices. If the subgraph G_1 is joined with graph G_3, the resulting graph (depicted in Figure 4.10(c)) contains valuable additional ordering information. However, subgraph G_2 does not connect to G_3, because the two graphs do not contain common vertices. This means that, if only subgraph G_2 was transmitted, the graph $G_4 = \text{join}(G_1, G_3)$ would result in a disjoint graph that misses ordering information. In the following, we will present three classes of possible approaches for exchanging ordering information based on transmitting subgraphs.

The presented approaches are based on the idea of selecting a subgraph for transmission by traversing the graph in reverse order, i.e., by following the edges in the graph in the opposite direction. For the representation of the graph this means that in addition to the edges also the *reverse edges* need to be stored. We distinguish between three approaches that, starting at the most recently added vertex, extract subgraphs in reverse order for the exchange between DB nodes:

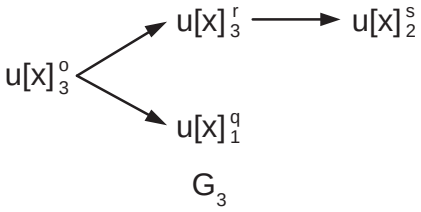
- depth-first based subgraph extraction,
- breadth-first based subgraph extraction, and
- negotiation-based subgraph extraction.

The first two approaches use traditional graph traversal algorithms, i.e., depth-first and breadth-first search along the reverse edges of the graph (see, for example, [HS78]). The number of vertices selected for the subgraph is limited to k. The parameter k describes the trade-off between the amount of data that is transmitted and the probability of successfully synchronized ordering graphs (under the assumption that synchronization is possible at all). In preparation for the subgraph extraction all transitive edges are removed from the ordering graph, the so-called transitive reduction [AGU72] of a graph. This has to be done in order to avoid situations where edges exist in the graph that lead from "old" to very recent vertices.

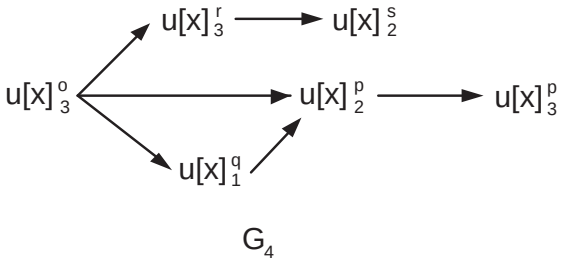
The third approach is based on the negotiation between pairs of DB nodes [HBR03] in four phases. A DB node N_1 that has accepted a new update request $\mathrm{u}[x]_n^o$ will advertise it by sending a message that contains the triple (x, o, n), the *id* x of the corresponding object, the *id* o of the observer that created it, and the sequence number n. The DB node N_2 receiving the triple will then check whether it has already accepted the corresponding update request or its current database entry is more recent than the offer. If both conditions are *not* fulfilled, DB node N_2 will collect all vertices from its ordering graph G_2 that do not have any successors in the graph in V_{new}, i.e., vertices that do not have any out-going edges. These vertices represent the most recent update requests known on N_2. The set V_{new} is then sent to N_1 which, in turn, selects a subgraph $G_{\mathrm{sub}} \subseteq G_1$ that contains all vertices $(V_{\mathrm{new}} \cap V_1)$ (or vertices from the same observer with a higher sequence number), the newly received update request $\mathrm{u}[x]_n^o$, and those vertices that are on any path between a vertex in $V_{\mathrm{new}} \cap V_1$ and $\mathrm{u}[x]_n^o$. If the subgraph G_{sub} is non-empty, it is transmitted to N_2 together with the update request $\mathrm{u}[x]_n^o$. If G_{sub} is empty, the negotiation failed and the graphs are synchronized by one of the other approaches that have been described.



(a) Graph available to be sent for synchronization



(b) Graph to be synchronized



(c) Result of joining G_1 and G_3

Figure 4.10: Example of using only a subgraph for synchronization

4.4 Removal of Objects from the Database

If the event that an object under observation leaves the entire system cannot be detected, the database may contain obsolete state records. If the system tracks people in a building, for example, the state record associated with a person should be automatically removed after he or she left the building. For this purpose, we adopt a soft state approach, which can be easily integrated into the above algorithms. Each state record is associated with a time to live (TTL) greater than the rate of state changes. Observers are responsible for refreshing the TTL by sending new state records for every object in their observation range, i.e., within a TTL period at least one state record must be issued. Nodes refresh the TTL timer of a state record whenever they accept an update for that record. If the timer expires, the corresponding state record is removed.

Chapter 5

Performance Analysis

In this chapter we examine the performance of the algorithms presented in the previous chapter. First, we define a set of metrics that are used to formally describe the performance in different dimensions. The next section describes the parameters of the system that impact the performance of the algorithms and which are varied in the following experiments. The chapter is then concluded by the presentation and the discussion of the results obtained in simulation experiments.

5.1 Methodology

5.1.1 Performance Metrics

In order to quantify the performance of the proposed algorithm the following quantities are measured during the experiments: the observation jitter, the read and update latency, the read and update success ratio, and the recency of the obtained read results with respect to the (idealized) global state.

Observation Jitter

The observation jitter $\delta_{\text{obs}} = \delta_{\text{comm}} + \delta_{\text{sens}}$ as defined in Definition 1 is a critical parameter that describes the best possible ordering accuracy for two

update requests in a given system. It is given as the sum of the jitter in the single-hop communication delay (δ_{comm}) and the jitter of the time it takes to sense the state of a physical object (δ_{sens}). In the experiments δ_{comm} is measured. The term δ_{sens} depends on the particular sensor technology used and is beyond the scope of the experiments.

Latency of Operations

Both, the latency of read and update operations are taken into account. The *read latency* denotes the time difference between starting a read operation on a client node and reception of the result for the read operation. Failed read operations are not taken into account. When Algorithm 1 (full replication) is used, the latency of read operations is close to 0, since no remote communication is used in the execution of a read operation. However, when Algorithm 2 is used, remote communication is used and thus the latency of read operations may be significant.

The *update latency* is defined at a node as the time difference between sending an update request at an observer and accepting it at a DB node. For example, consider an update request that is passed from an observer to node n_1 first and then from n_1 to node n_2. Assume further that it takes time t_1 for processing and sending the request from the observer to n_1 and time t_2 for the communication between n_1 and n_2. The update latency accounted for that update request will be t_1 at node n_1 and $t_1 + t_2$ at node n_2. The update latency is only taken into account for those request which are accepted by a DB node.

Update and Read Success Ratio

The success ratio in general describes the ratio between the number of successfully finished operations per node and the total number of operations that should have been completed on a given node. In the experiments, we distinguish between the *read success ratio* and the *update success ratio* for read operations and update operations. In the best case, a client node should execute all read operations it has requested and a DB node should execute all

update requests. If Algorithm 1 is used, the read success ratio is considered to be 1.0, since it is assumed that local read operations do not fail.

Update and Read Recency

The *update recency* for an object x compares an accepted update request at a DB node with the globally most recent update request available for x at the time of acceptance. Consider, for example, that an object has been updated 10 times by updates $u_1, \ldots, u_{10}$. The recency of the update is 0 if a DB node has stored u_{10}, 1 if the DB node has stored u_9, 2 if the DB node has stored u_8, and so on.

The *read recency* is defined analogously for the recency of read results. In this case read results are compared to the globally most recent update request for the given object at the time they are received by the client. The distinction between update and read recency is important for the partial replication algorithm. Here, read reply messages may be in transit between DB node and client while new update request are created in the system.

Message Overhead

For the full replication algorithm, the message overhead is measured in messages per update and node. Using this algorithm, messages are only sent when updates are created. When the partial replication algorithm is used, however, periodic message overhead, which is independent from particular updates, needs to be taken into account. Here, the message overhead is measured in messages sent per node and second on layer 2.

5.1.2 System Parameters

The performance metrics are influenced by a variety of system parameters, which are described next. The system parameters are varied in the experiments to investigate how they impact the overall performance of the system.

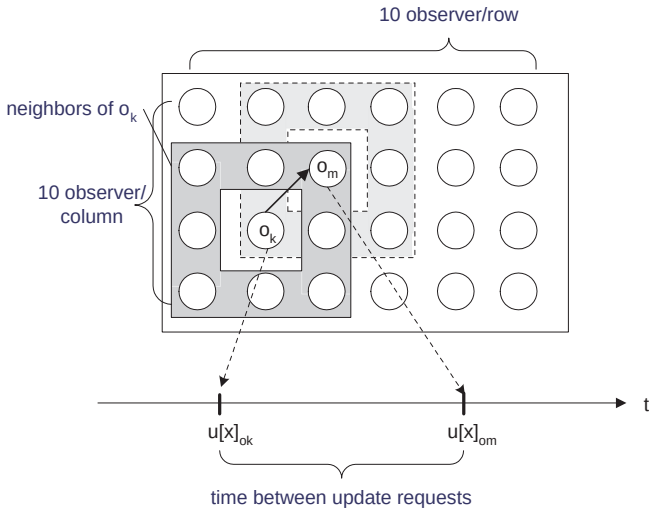


Figure 5.1: Selection of observers for update requests

Load Model

The load imposed on the system during the experiments is characterized by two parameters: the rate at which *read operations* and *update operations* are performed. Both, the read and the update rate are given in terms of a system-wide rate.

For the read operations, clients are selected randomly with a given *read frequency*. The observers for the update operations are chosen in order to simulate a random movement of the object under observation. The observers are arranged in a regular grid and remain stationary. Initially the observer that issues an update request is chosen at random. The next update request is then issued by an observer which is a neighbor of the preceding observer on the grid of observers. This process of selecting new observers and issuing update requests is repeated until the end of the simulation with a given read frequency. Figure 5.1 illustrates the selection of observers. Using this object mobility model stresses the replication algorithm, because objects are observed by distinct observers on every observation. This enforces that the ordering graph is used in most cases to decide on accepting or rejecting an update request.

Node Mobility and Density

The number of nodes in the network and their mobility pattern influence the topology of the network and therefore also the performance of read and update operations. Depending on the mobility of nodes, the network may, for example, become partitioned [HDMR04]. For this reason, an update operation that is created in one partition becomes available only to DB nodes in the same partition. If the spatial node density in the network is high, instead of network partitioning, broadcast storms [NTCS99] may lead to the collision of messages in the network. As a consequence, update or read messages may be lost, impacting the performance. In the evaluation the performance of networks where all nodes are stationary is compared with networks where nodes follow the random-waypoint mobility pattern.

Stationary Nodes: In the stationary scenarios nodes are initially deployed randomly on a given spatial area with uniform distribution. All nodes remain stationary until the end of the experiment.

Random-Waypoint Mobility: In the random-waypoint mobility model [BMJ+98] all nodes are initially placed at random positions (x, y) on the given spatial area. Each node then selects a random destination (x', y') to which it moves with a constant speed v. The speed v is chosen randomly from an interval $[v_{\min}, v_{\max}]$ with uniform distribution. After reaching its destination a node pauses for a time t_{pause} which is selected randomly from an interval $[t_{\min}, t_{\max}]$ with uniform distribution. After the pause, the node selects a new destination and a new speed and moves to the new destination as described before. The process is repeated until the end of the experiment.

The random-waypoint model has two major drawbacks if used in the unmodified version. First, the average node speed can decrease over time and asymptotically approaches 0 if $v_{\min} = 0$ [YLN03]. Secondly, the spatial distribution of nodes over time decays from its initial uniform distribution to a non-uniform distribution that has its maximum density of nodes in the center of the spatial area and a density of almost 0 close to the border of the area [BRS03]. The mobility pattern used in the experiments were generated using the mobility generator presented in [PBV05]. This mobility generator avoids the above mentioned drawbacks of the random-waypoint model by

generating mobility traces in steady state with a given average node speed and a steady spatial distribution of nodes.

Parameters of the Algorithms

The following gives a summary of the parameters used to configure the replication algorithms.

The reduce function used in all experiments is the *lossy-k-reduce* function with $k = 1$. Out of all proposed *reduce*-variants this function removes the most ordering information from ordering graphs and serves as a lower performance bound. At all times in the experiments the complete ordering graph is transmitted between DB nodes.

In the partial replication algorithm all clients select a server initially using the multicast-based server selection protocol. Read operations are repeated once after a timeout of 3 s. Observers announce themselves every 10 s. If a read operation failed twice clients initiate a server selection.

5.2 Evaluation of the Observation Jitter

The value of $\delta_{\text{obs}} = \delta_{\text{comm}} + \delta_{\text{sens}}$ in Definition 1 depends on the system mechanisms used to determine the order of update requests. The method we propose for the ordering of events is based on the order in which messages are received on the first hop node. This means that the value of δ_{comm} is *constant* given that the jitter of the single-hop communication delay is bounded. This section presents an analytical discussion of δ_{comm} followed by experimental results that strengthen the analytical results in practice.

5.2.1 Delay Jitter: Experimental Evaluation

The goal of the experimental evaluation is to find out the magnitude of δ_{comm} in practice. This section follows and extends the work that has been conducted in [Hor05] to measure the single-hop delay jitter.

The fundamental problem with measuring the communication jitter in a life system is that physical clocks are used. In practice, these clocks are inaccurate, i.e., they have a drift that lets them run faster or slower than the defined physical time [Lom02]. Formally, a physical clock $C_i(t)$ can be described as

$$C_i(t) = a_i(t)t + b_i \tag{5.1}$$

where $a_i(t)$ denotes the drift of clock C_i at physical time t and b_i denotes the skew of C_i at time $t = 0$.

Given a system with two processes P_1 and P_2 that run on two physical nodes and which may communicate directly using messages sent over a wireless communication interface. Process P_1 has access to a physical clock C_1 and process P_2 can access C_2. Let P_1 — representing an observer — send a sequence of k messages to P_2 — representing a DB node — at times $s_1, \cdots, s_k$. The corresponding times at which P_2 receives the messages are denoted as $r_1, \cdots, r_k$. The goal of the experiments is to obtain times $\delta_i = r_i - s_i$ for each of the messages in order to derive the maximum delay jitter δ_{comm}. The

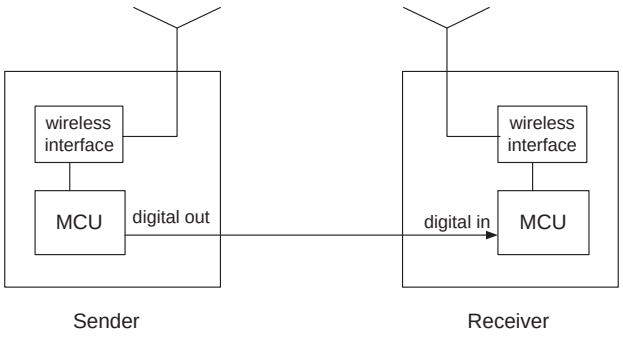


Figure 5.2: Setup for measuring the one-trip time

problem, however, is that the sender can only obtain $C_1(s_i)$, whereas the receiver can only obtain $C_2(r_i)$.

Measuring the One-trip Time

The communication pattern used by the observer nodes is based on a one-way communication. Each observer sends one message for each observation it makes. In order to make measurements for the delay jitter in a similar environment it is desirable to have only one-way communication in the experiments as well.

To be able to make precise measurements an out of band signaling mechanism is used that allows the sender of a message to signal the receiver the start of a transmission. The overall setup of the experiment is depicted in Figure 5.2. A general purpose (one bit) digital output of the sending node is used to signal the receiver. The signal received on the digital input channels is then used to trigger an interrupt handler on the receiver. The task of the interrupt handler is to simply reset the internal physical clock to 0.

After initialization, the sender will periodically send (layer 2) broadcast messages to the receiver immediately after sending a signal through the digital out channel resetting the clock on the receiver. Thus, the receiver can derive the one-trip time by reading its clock when the message arrives. The drift of the receiver's clock can be neglected in these experiments since the transmission of a single messages takes only a few milliseconds and its clock

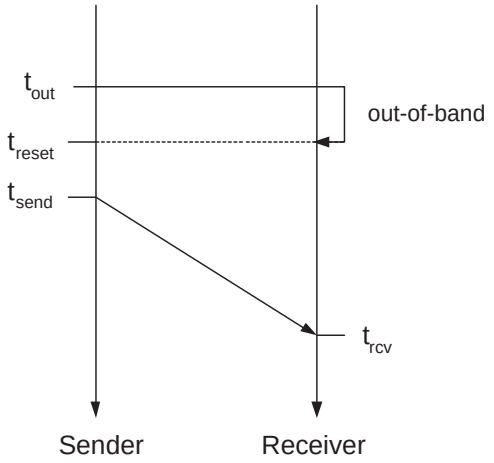


Figure 5.3: Communication to obtain the one-trip time

Parameter	**Value**
number of samples	7389
bit rate	19.2 kbps
message length	36 bytes

Table 5.1: Parameters of the experiments to measure the delay jitter

is reset using the out-of-band mechanism each time before sending a message. Figure 5.3 shows a single message exchange between the two nodes used to obtain the OTT. At time t_{out} puts a signal on the digital out channel. At time t_{reset} the receiver will have executed its interrupt handlers setting its clocks to 0. On receiving a message the receiver simply reads its local clock to obtain the OTT of the message. After obtaining a set of one-trip times, node 2 can calculate the delay jitter as the difference between the maximum and minimum OTT.

Experimental Setup

The experiments were conducted using two MICA-2 nodes [mot] Each experiment consists of 7389 messages being sent at a rate of one message per

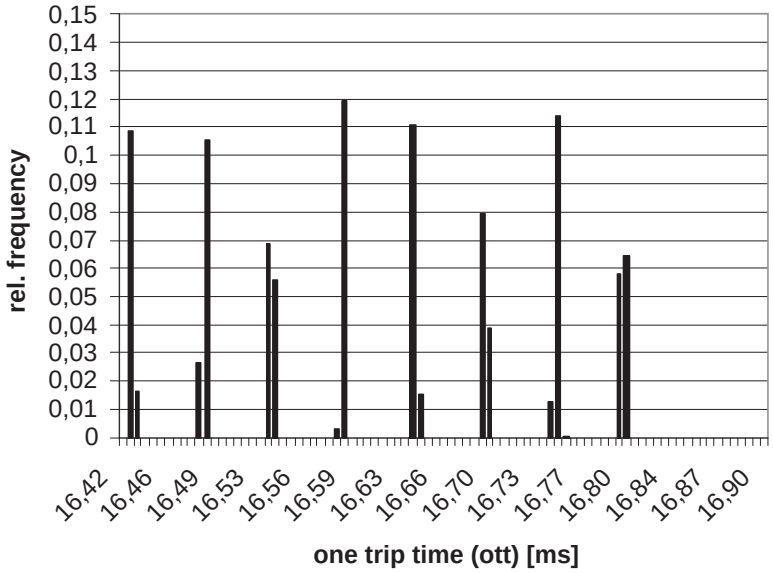


Figure 5.4: Histogram of the measured one-trip times

second to calculate the OTT. For the measurements TinyOS [HSW+00], a common operating system of the MICA-2 nodes, is used. The MAC-protocol implementation is modified in such way that it writes timestamps into messages immediately before serializing them to the channel. This approach avoids that the measurements are impacted by the delay jitter imposed by the randomization mechanisms used in the MAC-protocol. The communication is carried out with a bit rate of 19.2 kbps. Each message has a length of 36 bytes, which consists of 29 bytes payload and 7 bytes message header. Table 5.1 shows a summary of the parameters used.

Results

Figure 5.4 shows a histogram of the measured values of the one-trip times obtained in the experiments. The x-axis denotes time intervals into which the sampled messages latency fit and the y-axis shows the relative frequency of the samples. The OTTs are clustered around eight distinct times. The

difference between neighboring clusters is approximately 50 μs which corresponds to the time it takes to transmit one bit at a bit rate of 19.2 kbps (which is 52.08 μs). Internally, the radio transceiver used on the Motes[1] has a 8 bit shift register. Each time this register has been filled, the transceiver signals an interrupt to the micro controller (MCU) indicating that a byte has been received. However, this byte may contain up to seven bits of the preamble sent over the radio channel and only one bit which belongs to the first byte of the data frame. As a consequence the last byte that is received may contain between one and eight bits of the data frame. If the last byte contains one bit of the data frame, the MCU receives the completed frame seven bit-times later, since the shift register will be filled with seven bits that do not belong to the data frame. The delay will be six bit-times, if the last byte contains two bits of the data frame, and so on. This means that the theoretical lower bound for the jitter in this system is seven bit-times (364 μs). However, the maximum jitter obtained in the experiments is 436.19 μs. The difference of 72 μs goes back to the variance in processing delays[2].

5.2.2 Discussion

The results of the experiments conducted on a hardware platform for WSN applications (Mica Motes [mot]) show values of approximately 0.5 ms for δ_{comm} (excluding the randomization delay caused by the MAC layer). The ordering accuracy which can be achieved by using this system is therefore bound to an observation frequency of 2 kHz.

If synchronized (physical) clocks are used the possible ordering accuracy depends on the accuracy of the underlying clock synchronization algorithm. In the case of the time synchronization algorithm for MANETs presented in [Röm01] the *inaccuracy increases linearly* with both the number of hops a message traverses in the network and the age of the information. Measurements on standard PCs presented in [Röm01] show inaccuracies of approximately 2.2 ms and 3.75 ms for information that is 500 and 900 seconds old.

[1] The radio transceiver is a Chipcon CC1000 [Chi05]

[2] This corresponds to approximately 500 clock cycles of the MCU.

5.3 Evaluating the Full Replication Algorithm

In this section we present the experimental evaluation of the full replication algorithm. The first step is to evaluate the algorithm in an artificial environment without collision, contention, and transmission failures. This is done in order to study its performance without the impact of underlying protocol software and transmission channel. The next set of experiments studies the performance of the algorithm under the influence of a CSMA/CA based protocol on layer 2 of the protocol stack. In this setting various other parameters, such as the node density and the load model, are varied and their effect is evaluated. Finally, we also examine the impact of node mobility and transmission failures on the performance of the protocol.

5.3.1 Evaluation using a Perfect MAC-Layer

The goal of the first set of experiments is to evaluate the performance of the full replication algorithm in an (artificial) environment which allows for collision-free communication at a maximum data rate on MAC-layer without any transmission errors[3]. In this setting, the full replication algorithm can distribute updates to all nodes in the same network partition at the maximum data rate of the channel.

Using this setup, the number of DB nodes is varied. The node mobility for the DB nodes is chosen from the interval $[1.0, 2.0]$ m/s with a mean speed of 1.5 m/s. The pause time is set to 0. The number of observers is set to 100 arranged in a regular and stationary grid as described above. The simulation area is chosen to be 500 x 500 m^2. Each node has a transmission range of 100 m using the free space radio propagation model with no transmission failures and an omnidirectional antenna. Each experiment consists of 500 updates for a single object which are sent at an update rate of 10 updates per second following the load model described above. The properties of the system used for these experiments are summarized in Table 5.2

The examination of the success ratio shown in Figure 5.5 shows that

[3]This is often referred to as *null-MAC*

Parameter	**Value**
Number of database nodes	40, 80, 120, 160, 200
Number of observer nodes	100
Update rate (1/s)	10
Number of updates/experiment	500
Simulation area	500 x 500 m^2
Transmission range	100 m
Mobility model	random-waypoint (steady state)
Movement speed (db nodes)	$[1.0, 2.0]$ m/s
Maximum data rate	1 Mbit/s
Randomization period	[0, 5] ms, uniformly chosen

Table 5.2: Parameters of the initial set of experiments

almost 100 % of all updates were accepted by nodes on average. The fraction of the updates which were not accepted, have not been delivered to the node due to network partitioning. This can be derived from the experimental results, because all updates that were delivered to nodes (using the collision-free MAC) were accepted. In other words, no updates had to be rejected at nodes due to missing ordering information. The figure further shows that the majority of the updates were accepted based on decisions made on the ordering graph. This is caused by the simulation of mobile objects that are sighted by changing observers according to the load model described in Section 5.1.2. The second largest cause for accepting updates are those cases where DB nodes receive updates directly from observer. Finally, the cases where updates are accepted based on the comparison of version numbers, where two consecutive updates from the same observer are received by a DB node (see Algorithm 8, p. 74), occur very seldom (c.f. Table 5.3). These cases can only be found, if no DB nodes are in the transmission range of an observer, i.e., when the observer is in a network partition at the instance of sending an observation.

The results for the scenario with 40 DB nodes show a significantly lower update success ratio as compared to the scenarios with higher node densi-

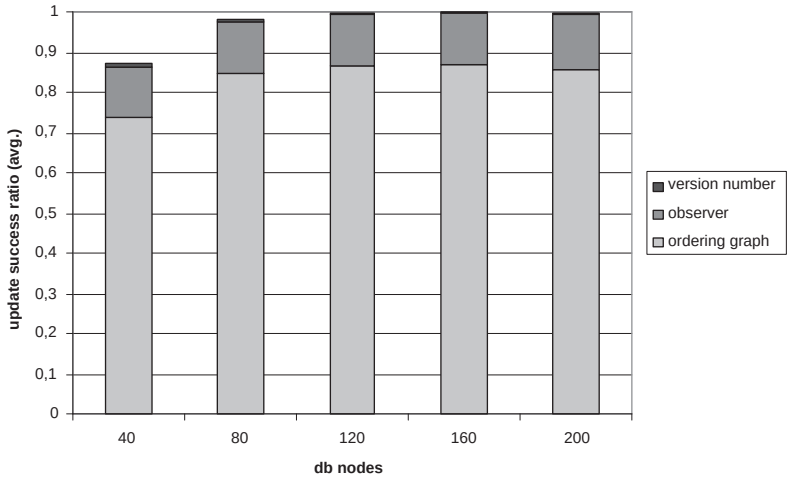


Figure 5.5: Average update success ratio using the collision-free MAC at an update rate of 10/s

DB	**total**		**ord. graph**		**observer**		**version num.**	
nodes	avg.	std.dev.	avg.	std.dev.	avg.	std.dev.	avg.	std.dev
40	0.872	0.144	0.739	0.152	0.122	0.063	0.010	0.005
80	0.983	0.062	0.849	0.071	0.128	0.050	0.004	0.002
120	0.996	0.026	0.864	0.064	0.127	0.060	0.003	0.001
160	0.998	0.002	0.868	0.069	0.129	0.070	0.002	0.001
200	0.996	0.004	0.856	0.074	0.136	0.075	0.002	0.001

Table 5.3: Update success ratio when using the collision-free MAC at an update rate of 10/s (higher is better).

DB	**recency**	
nodes	average	std.dev.
40	0.1462	2.9228
80	0.0171	1.1323
120	0.0039	0.2860
160	0.0013	0.0321
200	0.0037	0.0611

Table 5.4: Result recency when using the collision-free MAC at an update rate of 10/s (lower is better).

ties. This is due to the fact that network partitions occur more frequently in the 40 nodes scenario as compared to the other (higher) node densities. With 120 and more DB nodes, network partitions rarely occur, leading to an update success ratio of, on average, more than 99.5 %. This effect is also reflected in the standard deviation given in Table 5.3 for each average. In the 40 nodes scenario, the standard deviation is significantly larger than in the other scenarios, as some nodes may be isolated from the network partition in which updates are distributed. The results for the recency of results shown in Table 5.4 supports the success figures as the average result recency for 40 DB nodes is observably larger than for the higher densities and decreases to almost 0 as the number of DB nodes increases.

5.3.2 Evaluation using a CSMA/CA MAC Protocol

In the previous section we have presented an evaluation of the full replication algorithm using a collision-free MAC protocol. These experiments show the performance of the algorithm under ideal communication conditions. In this section we present simulation results using a more realistic MAC protocol. For the experiments we have used a MAC protocol based on *carrier-sense multiple access* with *collision avoidance* (CSMA/CA) which is used, for example, in the *distributed coordination function*[4] (DCF) of the IEEE 802.11

[4]The DCF is often called *ad-hoc* mode.

	rate 1/s		rate 5/s		rate 10/s	
DB nodes	average	std. dev.	average	std. dev.	average	std. dev.
40	0.0943 s	0.0971	0.1912 s	0.2901	0,4399 s	1,2798
80	0.1562 s	0.1756	0.5495 s	0.9135	0,6787 s	1,7502
120	0.2152 s	0.2624	0.5680 s	0.8307	0,8164 s	2,2173
160	0.2636 s	0.3124	0.5657 s	0.8321	0,7759 s	1,8863
200	0.3585 s	0.3968	0.5754 s	0.8053	0,7965 s	1,7059

Table 5.5: Average update latencies and standard deviation of accepted update messages using the CSMA/CA MAC protocol.

protocol family [Boa97]. Since the full replication algorithm uses only MAC-layer broadcast messages, the experiments do not employ a *request-to-send* (RTS) *clear-to-send* (CTS) mechanism for a so-called virtual carrier-sense. The parameter set for the experiments includes those shown in Table 5.2. Additionally, the update rate was varied between 1 and 10 updates per second.

Update Latency

The average update latency of accepted updates increases both with the number of DB nodes and the update rate in the system as shown in Figure 5.6 and Table 5.5. The results show a standard deviation which approximately equals the average for low update rates. The large confidence interval given by the standard deviation is caused by the varying number times an update message is forwarded. This means that DB nodes close to the originating observer will experience much lower latencies (small number of hops) than DB nodes which are further away from the observer (large number of hops).

For larger update rates, both, the average latency and its standard deviation, show a sharp increase. Due to the increased overall network load, which is caused by the higher update rate, collisions of messages occur more frequent. However, not all of these updates messages are finally lost, because the underlying flooding protocol may still deliver them on other routes in the network. This assumption is supported by the number of hops update

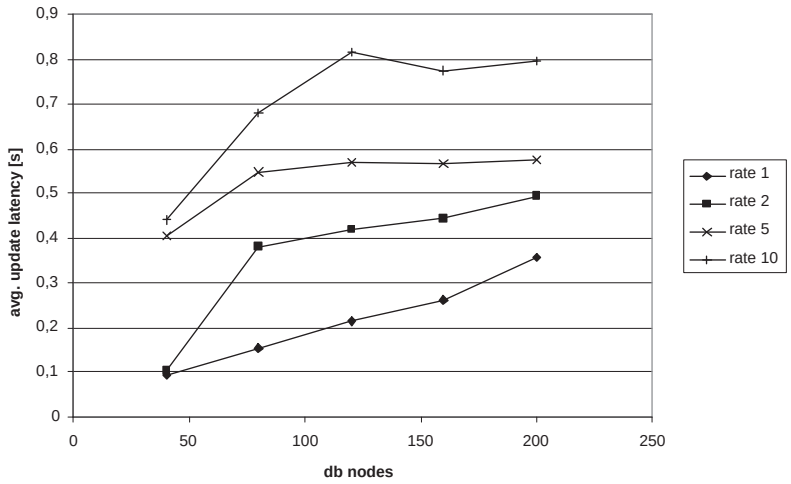


Figure 5.6: Average update latency over number of db nodes using the CSMA/CA MAC protocol

	rate 1/s		**rate 5/s**		**rate 10/s**	
DB nodes	average	std. dev.	average	std. dev.	average	std. dev.
40	3.6897	1.9823	3.5629	2.1352	3.2632	2.0958
80	3.7623	1.9931	3.9132	2.3432	3.6548	2.3565
120	3.8928	2.0532	4.3912	2.6936	4.2379	2.8991
160	4.0105	2.1587	4.7509	2.9266	4.7905	3.2524
200	4.3395	2.3505	5.1023	3.1313	5.2086	3.7953

Table 5.6: Average number of hops and standard deviation of accepted update messages using the CSMA/CA MAC protocol.

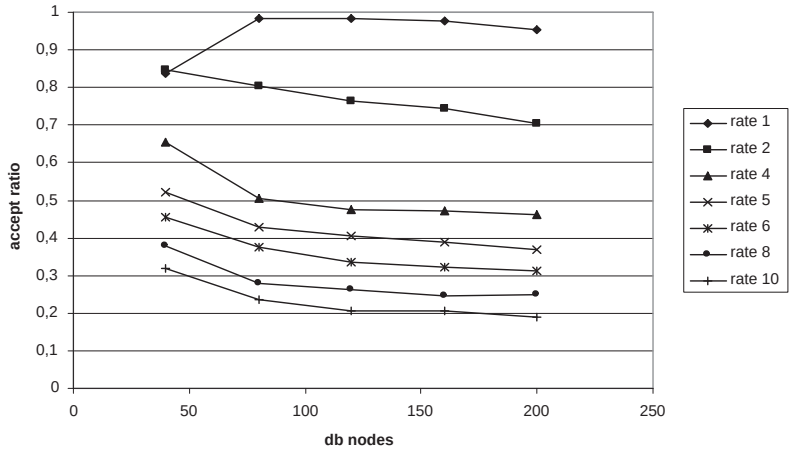


Figure 5.7: Average update success ratio over number of DB nodes using the CSMA/CA MAC protocol (higher is better).

messages traverse in the network depicted in Table 5.6. As the update rate increases for a given number of DB nodes, the average number of hops changes only slightly. However, the standard deviation increases significantly with the update rate, indicating that more and more updates exist which traverse paths other than the shortest path between the observer and the DB node.

Update Success Ratio

The update success ratio using the CSMA/CA MAC protocol is depicted in Figure 5.7. Compared with the update success ratio in the experiments using the perfect MAC protocol (see Table 5.3), the experiments with the CSMA/CA MAC protocol show a strong decrease of the success ratio as the update rate increases. With an update rate of 1 per second, the success ratio is close to the optimal values. As the update rate increases, the success ratio decreases significantly. The cause for this behavior, however, is the increasing network load which leads to frequent collisions of messages. The examination of the total average ratio of update messages that are received on DB nodes (not necessarily accepted) supports this assumption. Exemplary

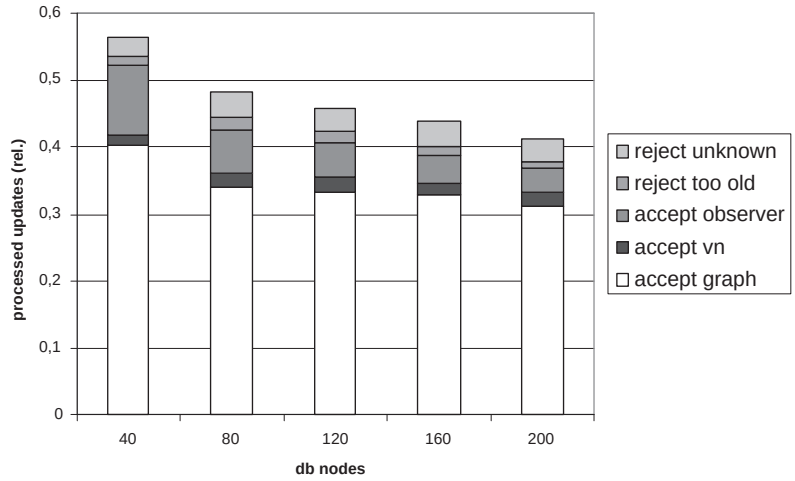


Figure 5.8: The cause of accepting and rejecting updates with an update rate of 5 update per second using the CSMA/CA MAC protocol.

for an update rate of 5 updates per second, Figure 5.8 shows the ratio of update messages that are accepted using the ordering graph, by comparison of version numbers, and by receiving updates directly from an observer (from bottom to top). Additionally, it shows the fraction of updates that were explicitly rejected by the algorithm because they were knowingly too old or could not be ordered. The remainder of the update messages, for example, approximately 45 % of all updates in the 40 DB node scenario, were not received by the nodes on average. Therefore, it can be concluded that the decrease in the update success ratio is mainly caused by the communication mechanism used for distributing update messages.

Update Recency

The next step in evaluating the full replication algorithm is to examine the recency of each update wich is accepted by the algorithm. Figure 5.9 shows the average update recency over the number of DB nodes in the scenario. Each curve shows the results for a different update rate between one update

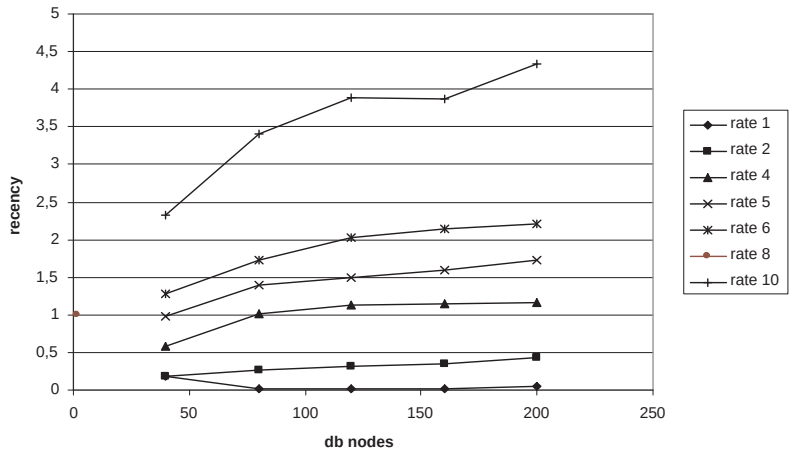


Figure 5.9: Average result recency over number of DB nodes using the CSMA/CA MAC protocol (lower is better).

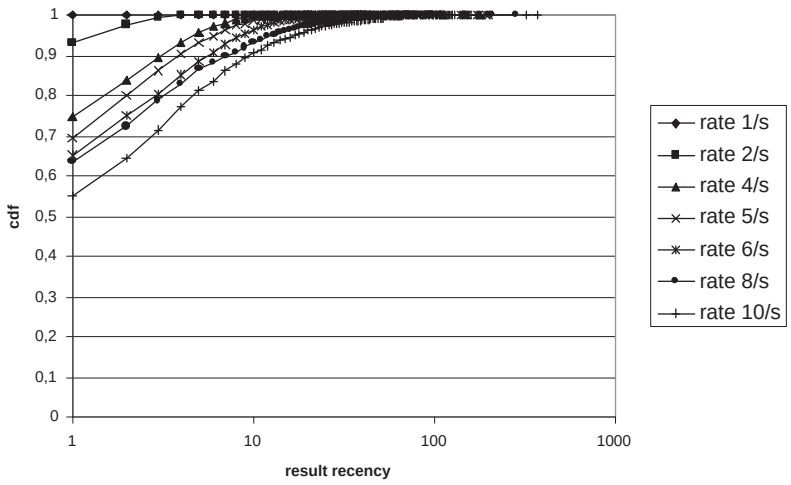


Figure 5.10: Cumulative distribution of the gaps for 120 DB nodes using the CSMA/CA MAC protocol (x-axis is log-scale).

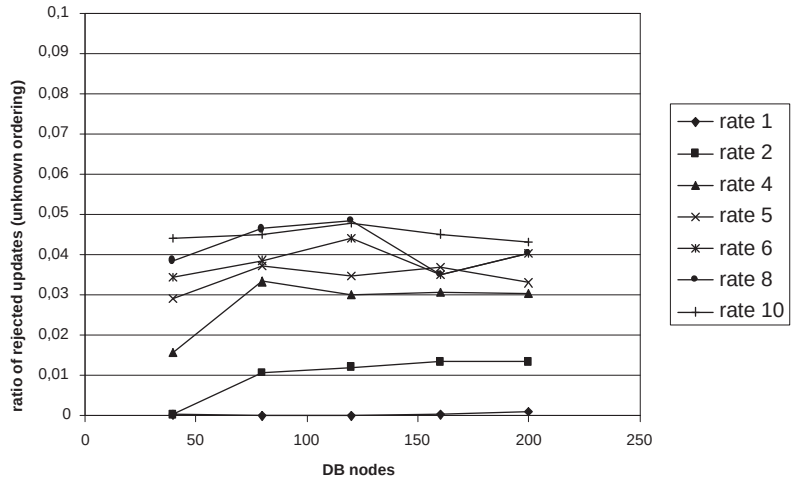


Figure 5.11: Fraction of rejected updates per DB node caused by insufficient ordering information using the CSMA/CA MAC protocol.

per second (bottom curve) and 10 updates per second (top curve). The set of experiments with one update per second comes close to the optimal recency of 0, i.e., almost every update is received and accepted by the algorithm. As the update rate increases, the recency of the information copies stored on DB nodes decreases, i.e., the number of updates messages which have not been accepted between two accepted updates increases. This corresponds to the results which have been presented for the update success ratio in the previous section. However, the cumulative distribution of the recency shown in Figure 5.10 shows, exemplary for the scenarios with 120 DB nodes, that most gaps detected between two accepted updates are small. With an update rate of 10 per second (bottom curve), for example, shows that approximately 90 % of all gaps are less or equal to 10.

Message Overhead

Using plain flooding as an update distribution algorithm implies that each DB node sends one message per update that has either been accepted or

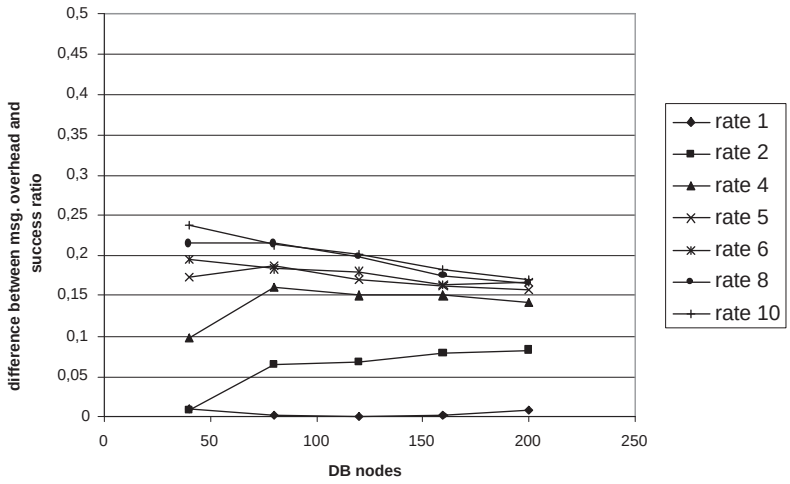


Figure 5.12: Difference between message overhead (per update and DB node) and update success ratio over the number of DB nodes using the CSMA/CA MAC protocol.

whose ordering is unknown. Additionally, a message is forwarded, if an update is not accepted, but the ordering graph on a DB node changes.

Therefore, Figure 5.11 shows the average percentage of rejected update messages caused by unknown ordering information. In comparison, Figure 5.12 shows the difference between the average message overhead per node and per update and the average update success ratio as a measure for the additional message overhead caused by messages sent because of unknown ordering information. These two sets of results show a similar behavior. However, these rejected update messages cause an message overhead which is, on average, 5 to 10 times higher, i.e., each update which is rejected because of unknown ordering information causes multiple messages. This is due to situations where more than one DB node rejects the same update message, leading to multiple messages for synchronizing ordering information per rejected update.

The maximum length of a message is determined by the number of ob-

DB node	**success ratio**		**update recency**		**update latency**	
speed	average	std. dev.	average	std. dev.	average	std. dev.
0.0	0.978	0.074	0.021	0.942	0.168	0.181
5.0	0.978	0.014	0.021	0.288	0.159	0.175
10.0	0.976	0.010	0.024	0.228	0.160	0.179
15.0	0.976	0.011	0.023	0.226	0.164	0.182
20.0	0.978	0.011	0.021	0.203	0.158	0.176

Table 5.7: Average and standard deviation for the update success ratio (higher is better), the update recency (lower is better), and the update latency using the CSMA/CA MAC protocol with 80 DB nodes at variable node speed.

servers which have reported updates for a particular object, if the *lossy-1-reduce* function is used (defined in Section 3.4.2, p. 53). In the simulation experiments, the ordering graph is represented as an adjacency matrix and a list of vertices. Let n_{obs} be the number of observers, s_{state} the size of an object's state in bytes, and s_{vertex} the size of vertex in bytes (which includes the observer *id* and the version number). Therefore the maximum message size $s_{\max}$ can be calculated by

$$s_{\max} = \lfloor (n_{\text{obs}}^2 + 8)/8 \rfloor + (n_{\text{obs}} \times s_{\text{vertex}}) + s_{\text{state}} \tag{5.2}$$

where the first term is the size of the adjacency matrix (given that each edge is represented by one bit), the second term is the size of the vertex list, and the third term is the size of the object's state. In the experiments we chose $n_{\text{obs}} = 100$, $s_{\text{state}} = 10$, and $s_{\text{vertex}} = 6$. The maximum payload which has to be expected for these parameters is therefore $s_{\max} = 1862$ bytes. The maximum messages size measured in the experiments was 1841 bytes, because not all observers made an observation in the experiments. The considerations for the message sizes do not contain any optimizations, for example, data compression.

5.3.3 The Impact of Mobility

In this section we investigate the impact of mobility on the performance of the full replication algorithm. To achieve this, the mobility of DB nodes was varied between 0 and 20 $\frac{m}{s}$ in a scenario with 80 DB nodes and an update rate of one update per second. The rest of the parameters was chosen according to Table 5.2. The previous experiments have shown that choosing 80 DB nodes and an update rate of one update per second have a high enough node density with only few network partitions and provide an environment which is not impacted by broadcast storms. For this reason this setting was chosen to evaluate the impact of varying speeds of DB nodes.

Table 5.7 shows the respective results for the update success ratio, the update recency, and the update latency (average and standard deviation). All three quantities show an almost constant behavior across varying node speeds. The only irregularity that can be seen lies in the standard deviations of the success ratio and the recency, which is higher for the static networks than for the mobile scenarios. In these cases the network contained some small partitions of DB nodes which did not change over time. In the mobile scenarios, these partitions changed over time such that individual nodes were isolated only for shorter periods of time. As a consequence, this led to fewer DB nodes which missed many updates in a row and therefore a smaller standard deviation in the mobile networks.

5.3.4 The Impact of Transmission Failures

Finally, we have investigated the effect of transmission failures on the performance of the full replication algorithm. For the same reason as in the previous section, the scenario with 80 DB nodes and one update per second was chosen as a basis for these experiments. In addition, the packet loss ratio was varied between 0 and 50 % at the receiver using a uniform distribution.

The results in Table 5.8 show that the update success ratio decreases as the packet error increases. However, due to the robustness of the underlying flooding-based broadcast protocol, the decrease is slow. Even with a packet error of 50 %, the success ratio decreases to only 0.89 (from 0.98 with

packet	**success ratio**		**update recency**		**update latency**	
error	average	std. dev.	average	std. dev.	average	std. dev.
0.00	0.982	0.024	0.017	0.372	0.154	0.174
0.10	0.964	0.028	0.037	0.428	0.165	0.185
0.20	0.961	0.028	0.040	0.432	0.178	0.196
0.30	0.955	0.033	0.046	0.457	0.174	0.184
0.40	0.922	0.040	0.083	0.522	0.183	0.195
0.50	0.896	0.050	0.115	0.612	0.189	0.200

Table 5.8: Average and standard deviation for the update success ratio (higher is better), the update recency (lower is better), and the update latency using the CSMA/CA MAC protocol with 80 DB nodes at various packet error probabilities.

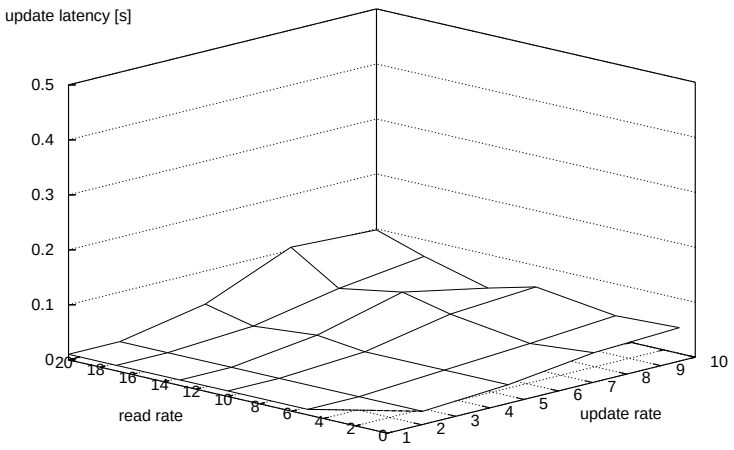
no transmission failures). As a result, the update latency increases, since redundant update messages arrive later at the DB nodes.

Parameter	**Value**
Number of observer nodes	100
Simulation area	500 x 500 m^2
Transmission range	100 m
Mobility model	random-waypoint (steady state)
Movement speed (db nodes)	$[1.0, 2.0]$ m/s
Maximum data rate	1 Mbit/s
Multicast routing	MAODV
Unicast routing	AODV
Max. number of read retries	1
Read timeout	3 s
Observer announce period	10 s

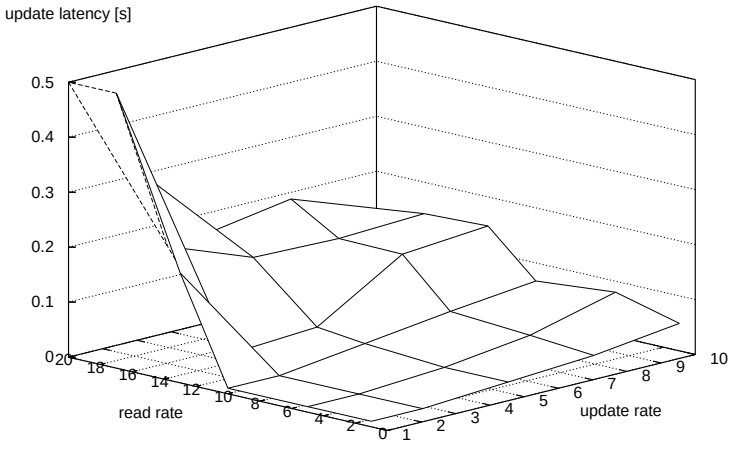
Table 5.9: Parameters used for all experiment with the partial replication algorithm.

5.4 Evaluating the Partial Replication Algorithm

This section serves to present the results of a set of simulation experiments using the partial replication algorithm. The multicast algorithm used in these experiments is MAODV [RP99]. The particular implementation for the simulator used was provided by [ZK04]. For the movement of nodes, the same mobility traces as in the evaluation of the full replication algorithm are used. Each of these experiments is started with one DB node. All other DB nodes are selected on demand by the replica allocation protocol. Each read operation is repeated once, if a read timeout of 3 s occurs. Every observer announces itself every 10 s. The basic set of parameters used for the experiments is depicted in Table 5.9. The presentation of the results for the full and the partial replication algorithms are different. The results of the partial replication algorithm depend on the read and update rate, which is reflected in the following figures.

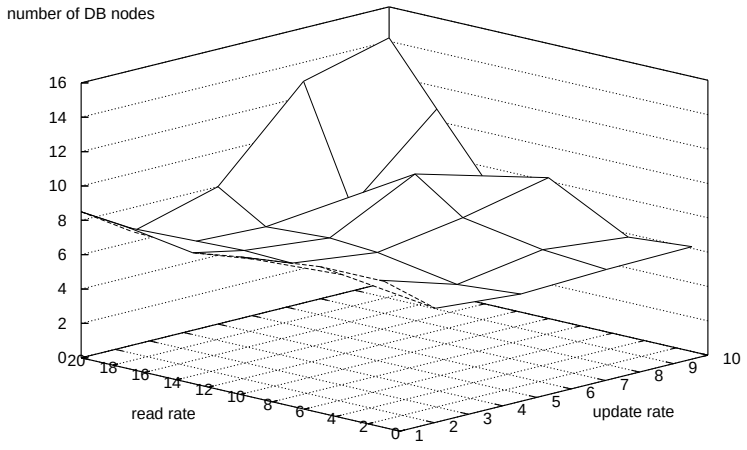


(a) 80 DB nodes and clients

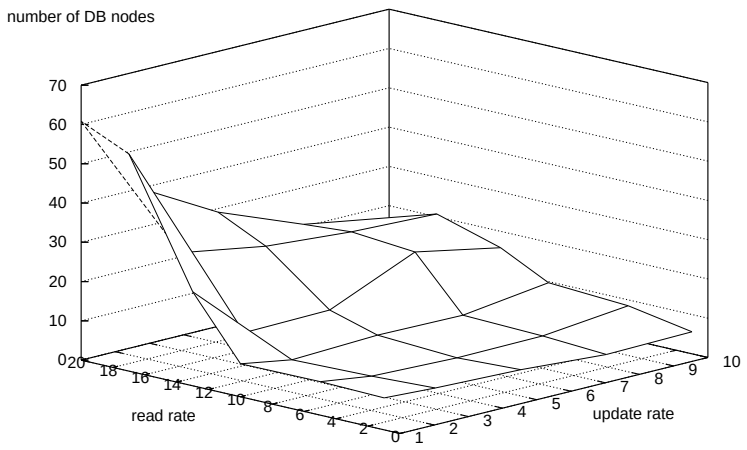


(b) 120 DB nodes and clients

Figure 5.13: Average update latency in dependence of read and update rate.

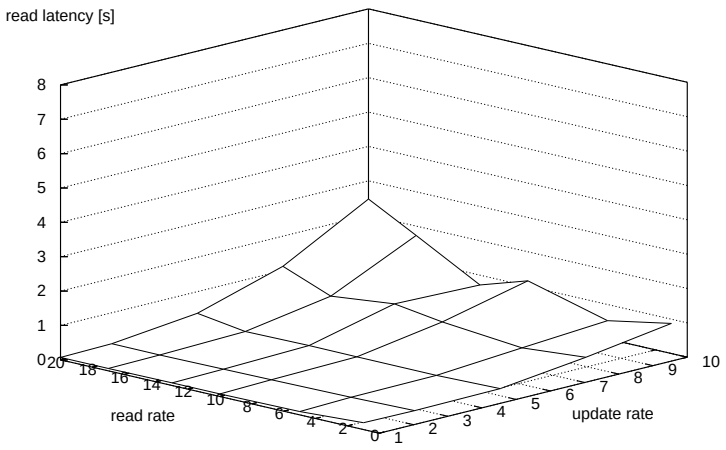


(a) 80 DB nodes and clients

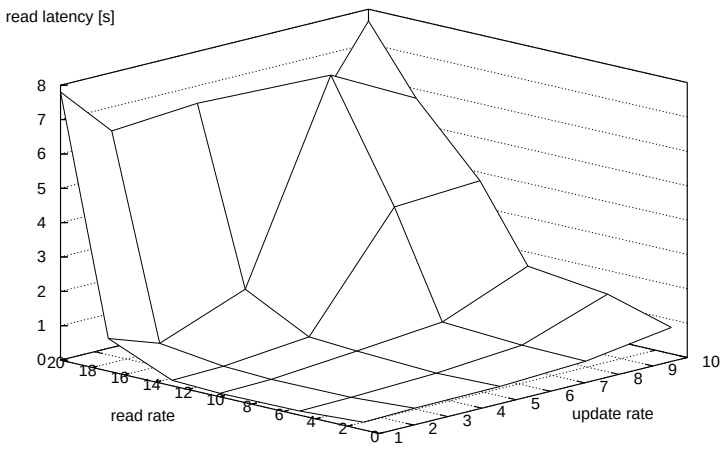


(b) 120 DB nodes and clients

Figure 5.14: Mean number of DB nodes in dependence of read and update rate.

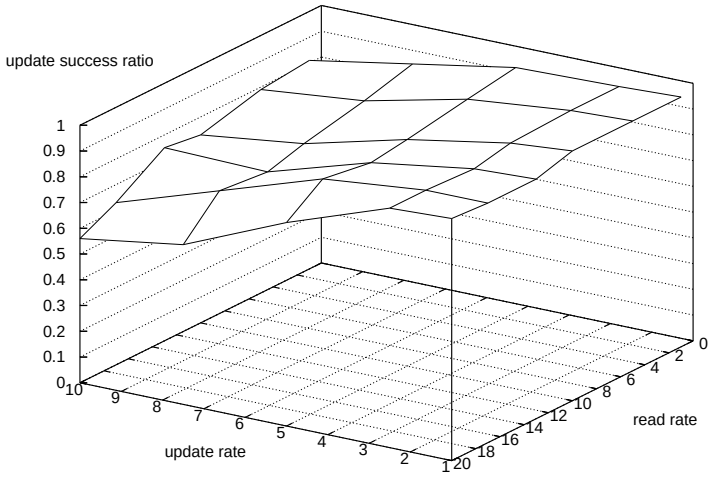


(a) 80 DB nodes and clients

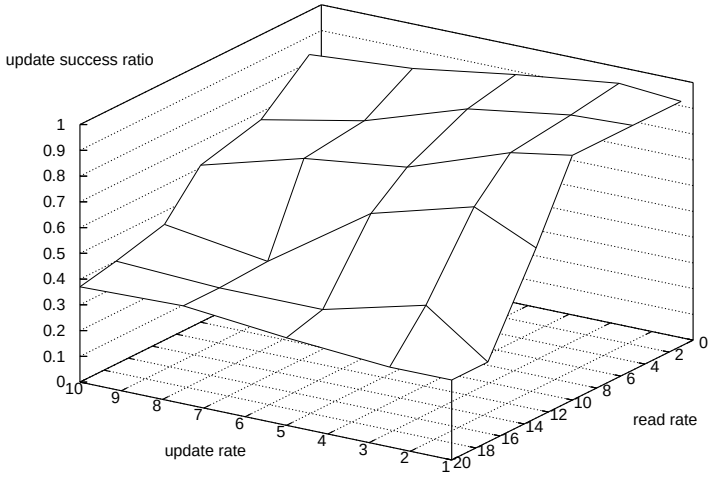


(b) 120 DB nodes and clients

Figure 5.15: Mean read latency in dependence of read and update rate.

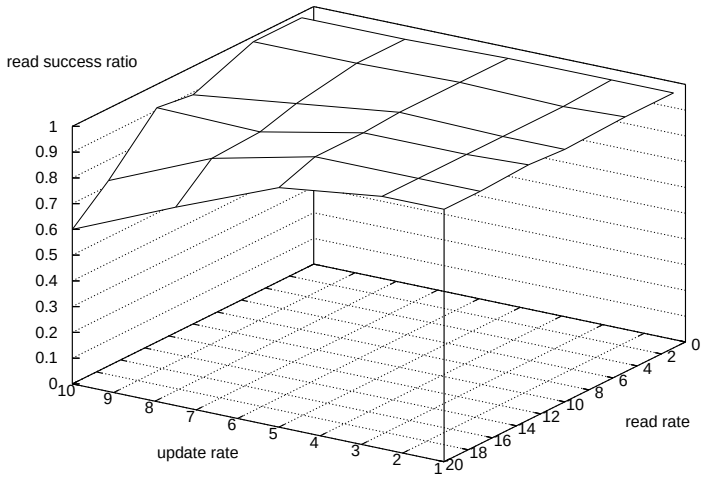


(a) 80 DB nodes and clients

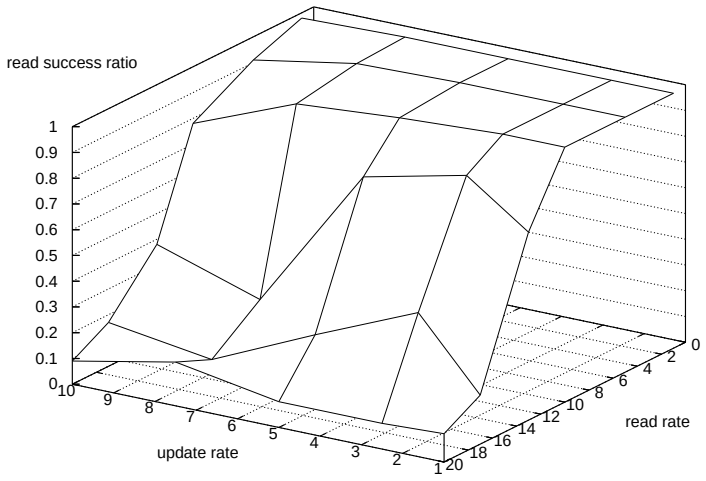


(b) 120 DB nodes and clients

Figure 5.16: Mean update success ratio in dependence of read and update rate (higher is better).

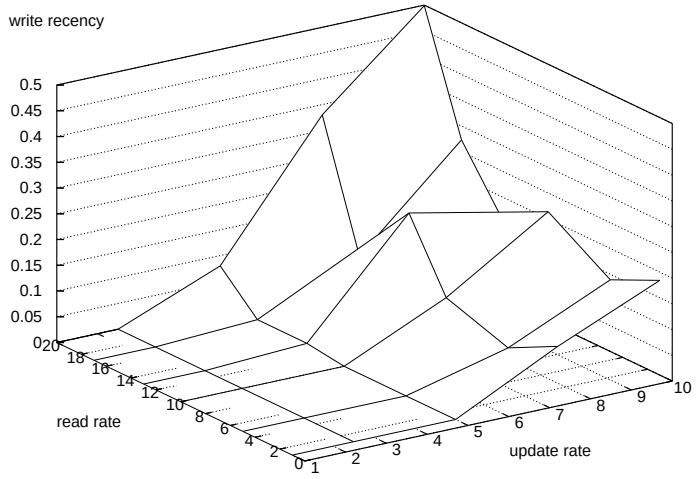


(a) 80 DB nodes and clients

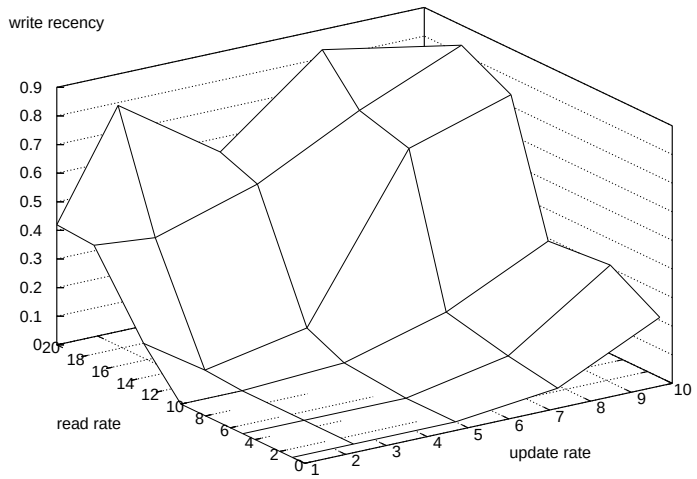


(b) 120 DB nodes and clients

Figure 5.17: Mean read success in dependence of read and update rate (higher is better).

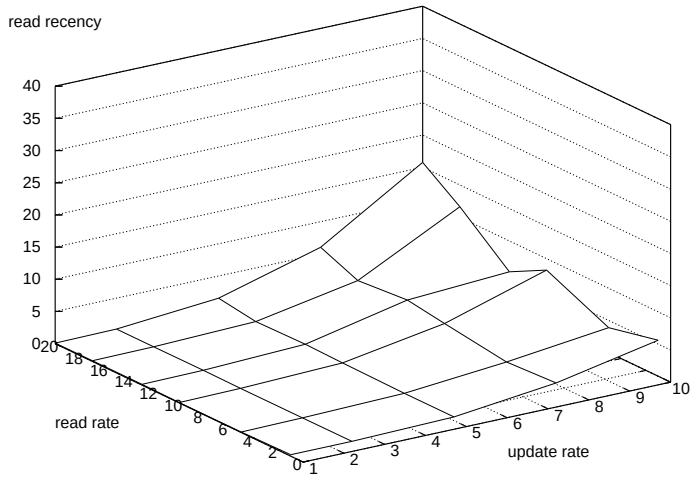


(a) 80 DB nodes and clients

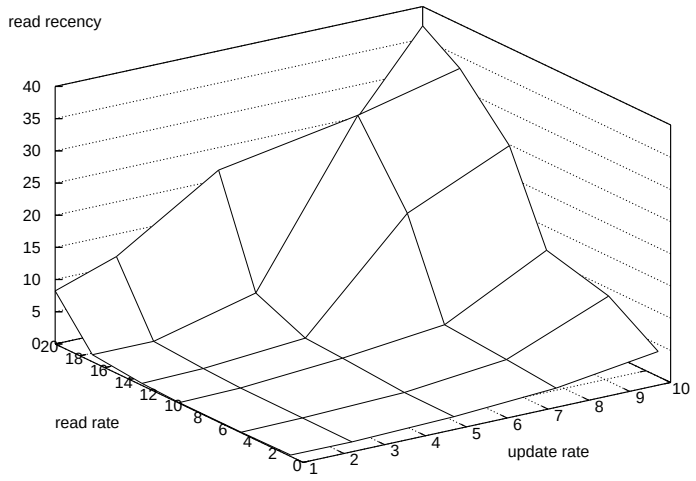


(b) 120 DB nodes and clients

Figure 5.18: Mean update recency in dependence of read and update rate (lower is better).



(a) 80 DB nodes and clients



(b) 120 DB nodes and clients

Figure 5.19: Mean read recency in dependence of read and update rate (lower is better).

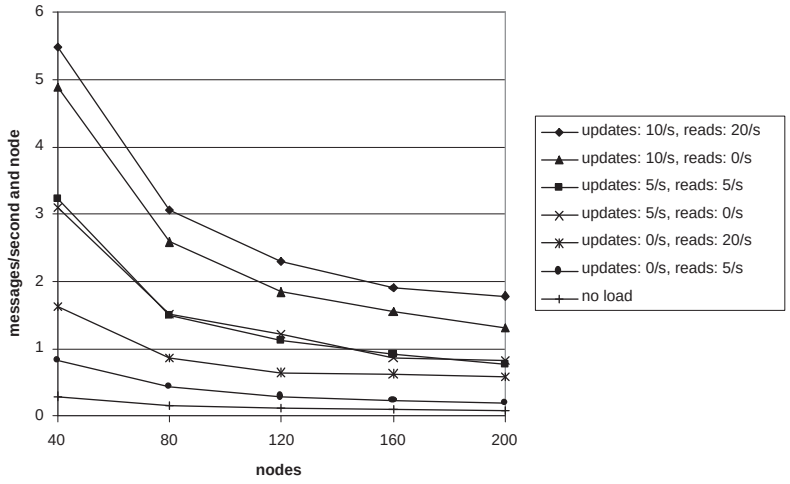


Figure 5.20: Messages sent per second and node

5.4.1 Operation Latency

Figure 5.13 depicts the average update latency for networks with 80 and 120 DB nodes and clients. The update latencies in both sets are similar for all update rates and for read rates between 1 and 10 read operations per second. As the read rate increases beyond 10 operations per second in the 120 node example, an increase of latencies can be observed. This is particularly evident for scenarios with high read rates and low update rates where the number of DB nodes increases as shown in Figure 5.14. Each of these update operation causes a multicast message which is sent to a higher number of nodes and therefore has a higher latency. The comparatively high number of DB nodes in these scenarios in turn is caused by an increasing number of handover operations as a result of read failures (see Figure 5.17 for the results of the read success).

The read latencies for the same set of experiments is depicted in Figure 5.15. Overall, it can be observed that the read latencies are in the range of up to approximately 1.5 s for scenarios with a read rate of up to 10 operations per second and update rates between 1 and 10 updates per second. As

the read rate increases beyond 10 operations per second, the read latency in the 120 nodes example increases significantly and reaches several seconds.

These results lead to the conclusion that the impact of the read rate on the overall performance is higher than the impact of the update rate. Especially in densely populated networks, for example with 120 nodes, this effect is evident form the results shown. This degradation in performance is caused by the AODV routing protocol which uses message flooding for the discovery of routes [PR99]. This overhead created by the routing protocol adds to the messages sent by the application.

5.4.2 Success Ratio and Recency

The results for the update and read success ratios of the 80 and 120 node networks are depicted in Figures 5.16 and 5.17[5]. In the 80 nodes example it can be observed that the update success ratio decreases slightly from 95 % down to approximately 80 % when the update rate is increased and the read rate is low. For high read and update rates the decrease is more distinct leading to update success ratios as low as 55 %. The read success ratio for the 80 nodes example shows results of 90 % and above except when both — the read and update rates — are high. Here, the lowest read success ratio is approximately 60 %. In the 120 nodes example, the read success ratios for higher update and read rates are strongly influenced by network contention. This leads to a sharp decrease in read and update success, especially for read rates greater than 10 operations per second. The decrease in update and read success corresponds to the increase of latencies discussed in the previous section and is caused by increasing network contention.

The results for the update and read recency in the 80 and 120 node examples are shown in Figures 5.18 and 5.19. The average update recency measured in all experiments is below 1, i.e., on average less than one update was missed at each DB node between two accepted updates. In the scenarios with low network load the standard deviation of the update recency is well below 2 indicating that only few DB nodes experience large gaps between two

[5]Please note that the viewing angle of the figures is adapted depending on the results.

accepted updates. For higher network load, e.g., high update and read rates in the 120 node example, the standard deviation of the update recency grows up to 8. This is caused by a small number of DB nodes that miss a large number of updates. However, even in the scenario with the highest network load, i.e., 20 read operations and 10 update operations per second in the 120 node network, it has been measured that 98 % of all update operations have a recency of less than 5. This indicates that the average is influenced by some large values for the recency, while most gaps are small.

In comparison to the update recency, the results for the read recency (Figure 5.19) expose significantly higher values for many scenarios. Especially in the experiments with update rates of 7.5 and more operations per second. In these experiments, the time between two update operations is at most 0.15 s. The average read latency (between sending the read request and receiving the reply) in these experiments exceeds 0.25 s in most cases. This means that in many cases updates will be received by DB nodes after a read reply has been sent and before it has been received by the corresponding client. This way the result contained in the reply will be less recent than the globally most recent value. In scenarios with 5 or less updates and 10 or less read operations per second the read recency remains below 1.

5.4.3 Message Overhead

Figure 5.20 shows the messages overhead measured in the number of messages per second and per node (client and DB node) in dependence of the number of nodes. These results include all periodic overhead caused by the routing protocols and the replication algorithm.

The lowest curve in the figure shows the message overhead in a scenario without any read and update operations. The remaining curves show the message overhead for the scenarios with 10 update and 20 read operations per second ($(10, 20)$-scenario) as well as the scenario with 5 update and 5 read operations per second ($(5, 5)$-scenario). Additionally, the results for 5 and 10 update operations without read operations ($(5, 0)$) and $(10, 0)$ scenarios) and 5 and 20 read operations without update operations ($(0, 5)$ and $(0, 20)$ sce-

narios) are shown. As shown before, the $(5, 5)$ scenario represents a setting where the replication algorithm delivers a good performance. Here, the overhead is approximately the same as for the $(5, 0)$ scenario. This means that the read operations introduce almost no additionally overhead on average, since AODV and MAODV a tightly integrated. As the load is increased in the $(10, 20)$ scenario the read operations cause a significant increase of overhead in comparison to the $(10, 0)$ scenario, where read operations fail frequently causing a higher number of handovers from clients to new DB nodes.

Messages that contain update requests have essentially the same length independent from the replication algorithm used (see Section 5.3.2, p. 123). Using the partial replication algorithm they are increased by a constant amount of data encoding the header of the routing protocol. Read request messages contain the routing header of AODV and the *id* of the object that shall be read and thus have a constant size. The size of read reply messages depends on the size of the state information for the object read. Apart from this, the size is constant (header). The periodic messages sent by observers within the replica allocation protocol have constant size, just as the corresponding reply messages sent by clients and DB nodes. The size of the messages that are part of the server selection algorithm sent by clients depends linearly on the number of distinct objects the client has read until it sends the message. The reply messages that are sent by DB nodes have constant size.

5.4.4 The Impact of Mobility

In the next set of experiment we investigate the impact of mobility on the performance of the replication algorithm. As a basis, a scenario with 100 observers and 80 nodes is chosen, where 5 update and 5 read operations are executed per second. This scenario delivered a good performance in the previous experiments. In this setting the mobility of nodes was varied between 0 and 20 m/s. In the scenario denoted as 5 m/s, each node (DB nodes and clients) moved at a constant speed of 5 m/s following the random way-point mobility pattern without pause time, etc. Table 5.10 shows the results for the update success ratio, the read success ratio, and the read recency. As the mobility of nodes is increased, the accuracy of the routing information

DB node	**upd. success ratio**		**read success ratio**		**read recency**	
speed ($\frac{m}{s}$)	average	std. dev.	average	std. dev.	average	std. dev.
0.0	0.968	0.015	1.000	0.000	0.302	0.567
5.0	0.862	0.052	0.966	0.099	1.485	3.376
10.0	0.826	0.073	0.939	0.128	2.317	5.642
15.0	0.813	0.054	0.905	0.166	2.283	5.525
20.0	0.784	0.097	0.833	0.223	3.328	8.833

Table 5.10: Average and standard deviation for the update success ratio (higher is better), the read success ratio (higher is better), and the read recency (lower is better) using the partial replication algorithm with 80 DB nodes at variable node speed (5 update and 5 read operations per second).

maintained by the AODV and MAODV algorithms decreases, because topology changes in the network become more frequent. Therefore, the update and read success ratio show a slow decrease as the mobility of the nodes is increased. In the static network, all read operations and 96.8 % of the update operations are executed successfully. All update operations that are not successfully accepted in this scenario are not delivered by the multicast algorithm. None are rejected by the replication algorithm. As the node speed increases, the average amount of update operations rejected by the replication algorithm increases to 1.3 % (min. 0 %, max. 4.5 %) for $v = 20\frac{m}{s}$. In conclusion, the amount of updates that are rejected by the algorithms in this case is significantly less than the overall reduction in the update success ratio caused by the increased mobility (which is 18.4 %). In summary, it can be concluded that the decrease in the performance of the replication algorithm is mainly due to communication failures which arise more frequently, when the speed of the nodes is increased.

5.4.5 The Impact of Transmission Failures

Finally, we have investigated the impact of transmission errors on the performance of the replication algorithm. The scenario with 100 observers, 80

packet	**upd. success ratio**		**read success ratio**		**read recency**	
error	average	std. dev.	average	std. dev.	average	std. dev.
0.0	0.928	0.029	0.992	0.041	0.614	1.154
0.1	0.922	0.031	0.986	0.058	0.657	1.149
0.2	0.906	0.031	0.984	0.066	0.772	1.511
0.3	0.742	0.169	0.795	0.319	2.774	7.317
0.4	0.480	0.082	0.134	0.224	14.191	23.974
0.5	0.298	0.090	0.013	0.077	27.652	34.196

Table 5.11: Average and standard deviation for the update success ratio (higher is better), the read success ratio (higher is better), and the read recency (lower is better) using the partial replication algorithm with 80 DB nodes at variable packet error ratio (5 update and 5 read operations per second).

nodes, and 5 update and 5 read operations per second was chosen as a basis for these experiments. The reason for this choice is the same as described in the previous section. The node mobility used is the same as given in Table 5.9. In different experiments the packet loss ratio was varied between 0 and 50 % at the receiver with uniform distribution. The results for these experiments are listed in Table 5.11. The packet loss shows only little impact on the performance of the replication algorithm for loss ratios of up to 20 %. When the packet loss is higher than 20 %, the performance of the replication algorithm decreases drastically. At a packet error ratio of 50 %, for example, the read success ratio is very close to 0. Similar to the performance shown in the previous section, the number of update requests that are rejected by the replication algorithm is low in comparison to the overall decrease in performance. In the scenarios without packet error, no updates are rejected. In the scenario with 50 % packet error, the average percentage of rejected updates is approximately 7 % (min. 0 %, max. 18.4 %). At the same time, the overall difference of the update success ratio in both scenarios is 63 %. This indicates that most of the updates, which are not reflected on the DB nodes are lost due to communication failures.

5.5 Comparison and Conclusion

In this chapter we have presented an extensive performance analysis of both, the full and partial replication algorithm, which are described in Chapter 4. For the experiments, we have varied the load model (update and read rate), the node density and mobility, and the quality of the transmission channel. In this section, we present a comparison of both algorithms and discuss the question of when to chose which of the two replication algorithms. This discussion is subdivided into the following questions:

- What is the expected read and update rate of applications ?
- Are the nodes going to be mobile? What is their speed of movement going to be?
- What is the excepted quality of the underlying transmission channel?

By comparing the results of the experiments, it can be concluded that the rate at which data items are read by application software plays a significant role. Read operations in the full replication algorithm do not create any communication in the network, since all clients are co-located on a DB node. In the partial replication algorithm, read operations are based on the communication between clients and DB nodes which may not be located on the same physical node.

The results obtained for the partial replication algorithm show that the overall performance is heavily impacted by the rate at which read operations are performed. This is particularly evident in Figure 5.17 (p. 133) where the read success decays sharply in the 120 node scenario as the read rate is higher than 10 operations per second. Looking at low read rates reveals that the read success remains almost constant for all update rates between 1 and 10 operations per second. At the same time, the update success ratio shows only a slight decay of approximately 10 % as the update rate is increased from 1 to 10 operations per second (Figure 5.16, p. 132).

The results for the full replication algorithm show that the update rate strongly impacts the update success (Figure 5.7, p. 120). Taking the results

for the 80 node network reveals that the update success ratio decreases from approximately 98 % to 25 % as the update rate is increased from 1 to 10 operations per second. At the same time the update success ratio for the partial replication algorithm varies between approx. 96 % and 80 % — however only for low read rates (Figure 5.16, p. 132).

The full replication algorithm has proven to be very robust with respect to mobile nodes. The variable node speed did not show any impact on the performance (Table 5.7, p. 125). In a similar scenario, the partial replication algorithm shows a significant decrease in performance as the speed of nodes is increased. The update success ratio, for example, remains at over 97 % when the full replication algorithm is used, while it decreases from 96.8 % down to 78.4 % for the partial replication algorithm (Table 5.10, p. 140).

Similarly, the full replication algorithm is very robust with respect to high packet error ratios. In the experiments the packet error ratio varied between 0 an 50 % with uniform distribution. The update success ratio of the full replication algorithm in the given scenario varied between 98.2 % and 89.6 % when the packet error was increased from 0 to 50 % (Table 5.8, p. 127). In a similar scenario the update success ratio dropped from 92.8 % down to 29.8 % when the partial replication algorithm was used (Table 5.11, p. 141).

Looking at the message overhead, it is important to consider that the full replication algorithm only creates messages, if an update request is created by an observer, i.e. it does not have any time dependent overhead. This is very attractive for systems with low update rates. In addition to the time-dependent overhead of the routing protocol that has been used, the partial replication algorithm has a time dependent overhead caused by the dynamic replica allocation protocol (Figure 5.20, p. 136).

In summary, it can be concluded that using the full replication algorithm is attractive, if the read rate is high or the update rate is low. Additionally, in systems that are exposed to high node mobility or frequent communication failures, it showed to be very robust. The partial replication algorithm shows good performance when the read rate is low. Its message overhead becomes attractive for higher update rates, justifying its periodic message overhead.

Chapter 6

Related Work

The models and algorithms which are presented in this dissertation are in the intersection of three areas of research: the chronological ordering of events in distributed systems, data replication, and data consistency. This chapter gives a discussion of related approaches from the field of consistency models, time synchronization, and data replication algorithms.

6.1 Consistency Models

6.1.1 Consistency Models in Database Systems

Strong consistency based on the concept of serializability has been addressed in the domain of distributed databases extensively [EGLT76, HR83, BHG87]. Since consistency is considered to be a trade-off to availability [DGMS85], strong consistency most likely results in poor availability in the presence of frequent network partitioning. Therefore, these approaches cannot be applied in arbitrary mobile ad hoc networks without significantly reducing the availability of data in many settings.

Several consistency models have been proposed to suit mobile environments. In the Bayou project [TDP+94] a set of consistency models have been proposed to provide so-called *client-centric* guarantees to applications. In particular the authors proposed the "monotonic reads", "monotonic writes",

"read your writes", and "write follows read" models. With monotonic reads, a process that has read a value of some data item x, will read the same or a more recent value of x for all successive reads. However, this guarantee for read operations does not enforce any ordering for write operations. Monotonic writes guarantee that a write operation of the same process is completed before a successive write operation occurs. In our model, however we assume that the state of physical objects is written (updated) by multiple independent processes. The "read your write"-model is used to express the guarantee that the result of a write operation will always be seen by successive read operations of the same process. The "write follows read"-model requires that a write operation which follows a read operation in the same process is applied to a value which is the same or more recent than the value which was previously read. The latter two models consider the ordering of read and write operations within a single process. In our model, we assume that the state information of objects is updated by one class of processes (observers) and read by another class of processes (clients), since write operations (update requests) are concerned with physical state, which cannot be altered by write operations of clients.

A number of consistency models have been presented in the context of database replication systems based on epidemic algorithms. The work described in [HSAA03] describes two algorithms. The first algorithm is a implements a pessimistic approach that guarantees serializability. The second algorithm is an optimistic algorithm which delegates the resolution of conflicts completely to the application. The work in [DGH+87] proposes epidemic algorithms to update copies in fixed networks. Their concept of consistency ensures that all copies converge to some common state. This concept does not enforce state transitions to be consistent with the order in which state changes have been observed in the physical world. However, consistency with respect to the order of the corresponding events in reality is most important for many monitoring and tracking applications.

The system presented in [BK85,SBK85] has been designed for partitioned networks and allows to execute transactions also while network partitions prevail. The consistency model used in this system is based on ordering transac-

tions by means of an abstract algorithm which preserves the local timestamps used on each node. Additionally, the algorithm serves as a system-wide tie-breaking algorithm. The task of the algorithm is to bring transactions, which have been executed in distinct network partitions, into the same order on all nodes without explicit negotiation. The authors do not present particular algorithms for that purpose. In this dissertation we provide an approach for maintaining the chronological ordering between operations which are created by independent nodes without using synchronized clocks.

6.1.2 Consistency in Distributed File Systems

In classical distributed file systems the semantics of file-sharing can be divided into UNIX-semantics, session semantics, immutable shared files, and transaction-like semantics [LS90]. The classical UNIX-semantics requires that all read operations to a file see the result of all previous write operations. The concrete order of read and write operations is typically beyond the scope of file systems using this semantics. Session semantics defines that the result of write operations are directly visible to the writing process whereas other processes have access to the changes only after the file has been closed and is therefore similar to the "read your writes" discussed previously. The definition of immutable shared files is based on the idea that a file — once created — can only be replaced by a new file but cannot be changed. The transaction-like semantics is similar to the definitions used in transactional databases, where opening and closing a file corresponds to beginning and committing a transaction. These approaches are not applicable in the scope of this work, because the chronological ordering of update operations executed by independent processes, which may observe the same object, must be maintained.

The Coda system [SKK+90] provides a file system which was especially designed to support disconnected operations. In disconnected operation [KS92] a client is allowed to operate on a cached copy of a file. This means that operations performed while being disconnected may conflict with the operations of other clients. Once a client is reconnected, the Coda system detects write/write conflicts based on so-called *storeid*s and Coda version vec-

tors (CVV). If a conflict arises, it is reported to the application for conflict resolution. When real-world phenomena are observed by sensors of independent processes, it is in general not possible to resolve conflicting operations on application layer. Given, for example, two temperature updates for an object. Without any further knowledge an application cannot decide which of the reported temperature readings is more recent than the other.

6.1.3 Memory Consistency Models

According to [Mos93] memory access can be categorized along the following dimensions: direction of access (read or write), causality of access, and category of access. The category of access distinguishes between competing and non-competing accesses, synchronizing and non-synchronizing accesses, and exclusive and non-exclusive accesses.

The sequential consistency model defined by Lamport [Lam79] defines a system that is executed on a multi-processor system to be correct, if the interleaving of the execution of each processor is equivalent to some serial execution. This model requires that all pairs of conflicting operations (read-/write and write/write) in distinct processes are executed in the same order. The cache consistency model [Goo89] is a location relative weakening of sequential consistency, i.e. it requires sequential consistency for all operations on a particular memory location. In these models the chronological ordering of operations is not conserved, because an operation in some process A may have to be delayed until another operation of some process B has been executed. Based on sequential consistency, a number of so-called *hybrid* consistency models have been proposed that divide memory locations into two classes: synchronization variables and data variables. Weak consistency [DSB88], for example, requires that the access to synchronization variables is sequentially consistent.

Causal consistency [HA90] represents, compared to sequential consistency, a weaker model. Here, operations that are in a potential causal relationship must be seen in the same order by all processes. Using the definition of causality based on message exchange [Lam78], a write operation

may causally depend on a preceding read operation (given that both operations access the same variable). Similarly, the result of a read operation may causally depend on the effect of a previous write operation. Just as with Lamport's definition the transitivity of causality also holds. In this work we maintain the chronological ordering between physical-world events which may be causally unrelated from the system's point of view, since they occur outside the system.

The pipelined RAM model (PRAM) [LS88] is based on the assumption that in a multi-processor system each processor has its own local memory. Write operations are written to the local memory of the writer and then sent to all other processors. In terms of ordering constraints this means that each processor reads the result of its own write operations straight away. Different processors may see write operations of other (distinct) processors in arbitrary order. However, they agree on the order of write operations of the same processor. In comparison to the PRAM model, the *slow memory* model defines a weaker level of consistency based on a location-relative weakening. Thus, processors only have to agree on the order of write operations on a single object. Both models do not consider chronological ordering between write operations that originate from different processors.

The concept of physical time has been considered in a number of consistency models such as linearizability [HW90], timed consistency [TRAR99], and delta consistency [SRH97]. For linearizability, each operation on an object is explicitly modeled as a pair of so-called invocation and response. Both, the invocation and the response of the operation are associated with one point in time at which they occur. At some point in time between invocation and response, the operation takes effect on the state of the object. For timed consistency and delta consistency it is required that any write operation completed at some time t is available for reading at *all* copies no later than $t + \delta$. For arbitrary MANETs and WSNs, it cannot be guaranteed that an operation takes effect within some specified time on all copies of an object. This may, for example, be unfeasible due to network partitioning caused by node mobility. In our work, we allow a write operation to be applied to copies at different points in time. However, our concept of consistency re-

quires that the order in which multiple write operations are applied to one copy is consistent with the points in time at which these operations were invoked. The consistency model for monotonic random registers [LW01] defines that a read operation reads from a write operation, if the write operation begins before the read operations ends. It is required that the read operation returns the result of the write operation, and that the write operation is the latest write operation. Furthermore, the model defines that every read must return the result of some previous write operation. However, the definition does not enforce any particular order on write operations and the ordering can only be applied to events that occur within a given system (similar to Lamport [Lam78]).

6.1.4 Consistency in the Internet

The Domain Name System (DNS) [MD88] comprises a hierarchical organization of names in the Internet. Names that are contained in a single zone are updated by a single DNS server. All other (slave) servers may maintain copies of these entries for reading. Clients poll the master server periodically to detect changes. In the timespan between an update and the next poll time a slave server maintains a stale copy of the entry which may be served to clients. In the scope of this work we allow the state of objects to be updated by multiple independent processes.

The network news transfer protocol (NNTP) [KL86] is used to distribute so-called news articles to a set of servers in the Internet. Clients may submit new articles to any news server. Each article is associated with a time stamp, a message *id*, and belongs to a certain category (newsgroup). Articles are replicated periodically among servers in so-called news-feeds. In a news-feed a server connects to other known servers and exchanges articles based on their *id*. This means that all servers will eventually have replicas of the same articles for the newsgroups they host. NNTP does not support update operations on articles. Update operations, however, are necessary for the work presented here in order to reflect changes in the physical world within the system.

The consistency of web caches — also called cache coherence — has been studied extensively. According to the work presented in [Wan99] cache coherence mechanisms can be divided into two classes: strong and weak cache consistency. The mechanisms providing strong cache consistency can be further subdivided into: client validation and server invalidation. In the former approaches the client validates the freshness of a cache entry on each access. In the latter approaches the server sends invalidation messages to clients, if an object is updated. However, both classes of coherence mechanisms require reliable network connections at all times. The class of weak cache consistency can be subdivided into: cache invalidation based on a (adaptive) time-to-live (TTL) of each entry and on piggyback invalidation. All these approaches have in common that objects are only updated by a single process, e.g., the web server.

6.2 Time Synchronization

During the past decades many approaches have been proposed to synchronize the clocks of a set of hosts in a computer network. In Christian's algorithm [Chr89] there is one dedicated time server in the network which has the correct time by assumption. Each client that wishes to synchronize its clock sends a periodic request message to the time server, which in turn replies with the current time at the server. In order to estimate the time at which the server has sent the reply, the one-trip time (OTT) is approximated by half of the round-trip time (RTT) of request/response messages. In the Berkeley algorithm for clock synchronization [GZ89] the authors assume that one dedicated host, the time server, is responsible for initiating the synchronization. The time server periodically polls every machine in the network and computes the average time. Next, the server sends to each client the difference between the clock of the client and the average time in the system in order to adjust the clocks. In this approach the RTT is also measured to approximate the OTT. Both approaches are highly centralized and rely on the assumption that the OTT is approximately half of the RTT. In MANETs, however, the OTT of messages may change rapidly as the connectivity and the load of

the network changes. Therefore, this fluctuation of the OTT impacts the accuracy of the synchronization significantly. The network time protocol (NTP) [Mil94] is one of the most commonly used protocols for time synchronization in wired networks. The time synchronization service is provided by a tree hierarchy of servers which are organized in so-called *strata*. The root of this tree belongs to stratum 1, the stratum of other servers is set according to their distance to the stratum 1 server. Each stratum-$(n+1)$ server synchronizes with a server on stratum-n, enabling a distributed synchronization. However, this approach introduces higher synchronization errors on higher strata. Synchronization with NTP also relies on the OTT estimation based on the measurement of RTTs.

In the area of MANETs and wireless sensor networks time synchronization received a lot of attention in the past years. The idea of time stamp synchronization in MANETs was first introduced in [Röm01][1]. In this approach each time stamp is represented as a time interval $[t-\delta, t+\delta]$ where t represents a point in time and δ represents the error. The real point in time can be anywhere within the interval. Therefore, two time stamps can be correctly ordered only if their intervals do not overlap. Synchronization is performed pairwise between nodes whenever time stamps are exchanged. Since the synchronization is performed in an on-demand manner, δ is influenced not only by the inaccuracy introduced by measuring the RTT, but also on the age of a time stamp. Additionally, δ is increased every time a time stamp is exchanged between a pair of nodes. In general, the accuracy of this approach decays with the network diameter and the age of time stamps. The author of [Röm01] has conducted measurements that show a linear increase of the inaccuracy with respect to the number of hops a time stamp traverses and the age of a time stamp. For example, inaccuracies of approximately 2.5 ms for an age of 600 s and 0.9 ms for traversing 5 hops have been measured. The accuracy of chronological ordering in this dissertation is based on the assumption that the maximum jitter of the single-hop communication delay is bounded and reasonably small. In systems where this assumption holds, the algorithms presented in this dissertation provide

[1] A similar approach called *post-facto* synchronization has been proposed in [EE01]

ordering with constant accuracy. In particular, the accuracy does not depend on the network diameter or the age of information.

6.3 Data Replication in MANETs and Sensor Networks

Data replication has been addressed in the context of mobile ad-hoc and sensor networks by many authors. The adaptive broadcast replication protocol (ABR) [XWC00] was proposed to disseminate sensor data through a wireless sensor network. ABR ensures weak consistency assuming that there exists a single update source per object, i.e. each update created by a node can be totally ordered by means of version numbers. On each local update, the updating node calculates a distance to the previous value in order to estimate the benefit of sending the update to other nodes in the system. In our work we allow multiple update sources for each object.

Deno [KC00] presents an epidemic replication algorithm based on weighted-voting for weakly connected mobile ad hoc networks. On object creation a certain amount of so-called currency is associated with each object. When an additional copy of a particular object is created, the amount of currency is split between copies. An update on an object can only be committed, if a sufficiently large subset of all copies agrees to commit. The subset is sufficiently large, if the sum of currency for the object is greater than half of the total currency for that object. This algorithm ensures that each copy commits updates in the same order. However, there are no chronological constraints concerning this order.

The authors of [LHE03] present a collection of protocols for probabilistic quorum-based data storage in MANETs. Read operations will return the result of the latest update operation that has been written to a quorum. No assumptions are made in which order update operations, especially those from different update sources, are applied to a quorum. Similarly, in [KMP99] a quorum-based system is used to provide access to the most current state information of objects. Each update is time-stamped under the assumption

that the clocks in the system are synchronized. Queries are sent to an arbitrary quorum by clients. After collecting all responses, the client selects the datum with the youngest time stamp as a query result. Depending on the connectivity of the network, clients may not reach arbitrary nodes of a given quorum, which in turn may lead to reading stale information. In contrast to these approaches, our consistency model explicitly enforces the chronological order of update operations from multiple update sources.

The Passive Distribution Indexing scheme (PDI) [LW03] was proposed to store $(key, value)$ pairs on mobile devices in a mobile ad-hoc network. Within the algorithm, a combination of query and result dissemination is used. Mobile hosts that receive results for queries may cache these for future use. For cache invalidation, the authors propose a hybrid-strategy of explicit invalidation and timeout values for cache entries. Data items may only be modified by the origin server (single update source).

The work presented in [Har01] provides a set of algorithms for replica placement and update dissemination in mobile ad-hoc networks. However, the authors take the assumption that data items — once created — cannot be changed. This means that the ordering of update operations does not have to be considered in this system. In [HHN05] the authors assume that data items are only updated by a single node in the system.

6.4 Discussion

Many of the consistency models which have been discussed previously are used to provide strict ordering guarantees such as sequential or causal ordering, e.g. [BHG87, HSAA03, HA90, Wan99]. Therefore, these models are not suitable for applications in MANETs envisioned in this dissertation for two major reasons: they do not maintain the chronological ordering between operations and result in poor availability in dynamic networks, for example because they use locking of information objects.

Weaker existing models, e.g. [TDP+94, SKK+90, MD88, DGH+87, SBK85] that have been designed for dynamic environments or disconnected operations do not provide chronological ordering guarantees. Here, the focus is,

for example, on the convergence to a common state [DGH+87, SBK85] or the ordering of operations from the perspective of a single client [TDP+94]. Other examples [SKK+90] focus on the detection of conflicts which have to be resolved by the application software or assume that updates can only be applied to a single dedicated copy [MD88]. These models do not consider the ordering of operations – executed by multiple independent processes – with respect to physical time.

Those models which explicitly consider physical time, for example [HW90, TRAR99, SRH97, LW01] are not suitable for dynamic environments such as MANETs. These models define time-bounds in which operations must be executed [HW90, TRAR99, SRH97] or require no particular ordering on write operations [LW01].

The data replication algorithms that have previously been proposed can be used in combination with existing consistency models, but do not propose a particular model [LHE03], assume that data objects are only updated by a single dedicated process [XWC00, Har01], or propose that updates are executed in the same yet any order at all copies [KC00].

The consistency model presented in this work defines a weak consistency model that takes the chronological ordering of read and write operations into account. It requires that write operations of independent processes are ordered according to the time at which their execution *starts* and that processes read the results of those write operations in increasing (chronological) order. The model is suitable for networks with dynamic topology changes and can be guaranteed even in networks that are temporarily partitioned. The replication algorithms presented in this work are based on the read-one and write-all approach. On the basis of our consistency model no locking of data objects is required and thus a high availability in MANETs is achieved, i.e., both read and write operations can be executed at any time.

Chapter 7

Summary and Conclusions

The work in this dissertation addresses applications that require the state of their physical surroundings in order to fulfill their overall goals while operating in an environment where no networking infrastructure is available. An example for such applications is the coordination of rescue workers which is based on the situation of all workers (their state) and the state of physical objects in their operational environment. Since the chronological order in which changes to this state information occur is important for making decisions, we provide a *novel consistency model* called update-linearizability that provides chronological ordering guarantees for update and read operations on data objects. The key idea of the model is that update operations are only applied to the data model held in the system in the order in which they occurred in the physical world. If, for example, some object increases its temperature over time, successive changes have to be reflected in the system according to the physical *time at which they are observed* by sensors in the system. This guarantee must be independent from other system properties, such as the overall communication latency in the network.

We have provided two replication algorithms that are suitable for a wide range of systems. The system model which is defined for the algorithms only requires that communication between nodes is possible on best-effort basis. The communication latency between nodes can be arbitrary, except for the communication on the first hop between a node that senses a state change and the neighboring node that receives the state change for further processing.

Here, we require that the maximum jitter of the communication latency is bounded. This is required in order to guarantee that the chronological order of update operations can be captured correctly in the system and has been shown to be realistic in real systems such as nodes used in wireless sensor networks.

The replication algorithms presented in this dissertation have two very important attributes: they provably guarantee update-linearizability and they do not require any pair of physical clocks in the system to be synchronized. This means that the chronological ordering defined by update-linearizability is guaranteed without using traditional clock synchronization schemes. This is a preferable property, since the accuracy of clock synchronization schemes typically depends on the network diameter (in hops) and the age of information. The former dependency is either introduced by a hop-by-hop synchronization where each hop introduces an uncertainty or a high multi-hop communication jitter in the system. The latter dependency is introduced by the time that passes between two synchronization events. The accuracy with which the chronological ordering is reflected in systems using the algorithms proposed in this work solely depends on the maximum jitter of the single-hop broadcast communication delay in the network. This is achieved by using a special data structure, the so-called ordering graph. The chronological ordering between state changes is derived from the communication between an observer which detects a state change and the first node that receives the information over a single hop in the network. Once this ordering information is stored in the graph, it does not change over time and can be passed over multiple hops without becoming inaccurate. Additionally, arbitrary ordering graphs can be merged in order to consistently combine ordering information that has been collected by independent nodes. In summary, the ordering graph is used to derive the ordering of information which has been sensed by independent observers in the system.

The two algorithms presented in this work are used to replicate the most recent state of each observable object in the system. The first algorithm is used to fully replicate the most recent state of each observable object in the system on all nodes. The second algorithm replicates this information on

a subset of all nodes. In this case, other nodes may read state information remotely. The observers used to sense state changes are only required to hold a small state that describes the physical objects that are currently under their observation. These nodes do not need to maintain any history of their observations. When the full replication algorithm is used, these nodes only have to send messages, i.e., they do not require any means for receiving messages on application level. When the partial replication algorithm is used, they need to be capable of sending single-hop broadcast messages. In both cases they do not require any routing protocol implementation which reduces the complexity of the software.

Within this dissertation we have presented experimental results, which show that the achieved performance of the proposed algorithms is high for a wide range of system parameters. In summary, the recency of the state information available to applications is high as long as the network is not congested. In these cases, the recency in comparison to the globally most current state information mainly depends on the multi-hop communication latency in the network.

Using the algorithms presented in this dissertation, it is possible to develop systems that

- capture and store the state of physical world objects,
- capture and store the chronological ordering between these state changes,
- do not require a networking infrastructure,
- do not require synchronized clocks, and
- use these state changes to make decisions in the applications.

In the context of this dissertation, future work can be divided into three categories: communication mechanisms, extension of the available state information, and systems design.

The communication mechanisms used in conjunction with the replication algorithms are standard protocols. On the one hand this is an advantage, because of their multi-purpose character. On the other hand it may be possible

to develop communication mechanisms that are optimized for distributing physical world events (update requests after all) in the system.

The state information which is available to applications may be extended by the provision and management of state histories. Currently the aim is to maintain the most recent state information for each object. If an application requires a history of state information, it has to maintain this information by itself. Extensions may be made to manage state histories within the system software in a generic way. A question that arises when state histories are managed by the system is: how to treat updates that are older than the currently most recent information available at the node when they arrive?

In the area of systems design it will be important to gather more knowledge, both theoretically and practically, on how to design and evaluate systems with the concepts and methods that have been proposed in this work.

Bibliography

[AGU72] Alfred V. Aho, Michael A. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[BBH02] Christian Becker, Martin Bauer, and Jörg Hähner. Usenet-on-the-fly - supporting locality of information in spontaneous networking environments. In *Workshop on Ad hoc Communications and Collaboration in Ubiquitous Computing Environments at ACM CSCW 2002*, 2002.

[BG84] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, 1984.

[BHG87] Phil Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[BK85] Barbara T. Blaustein and Charles W. Kaufman. Updating replicated data during communications failures. In *Proceedings of 11th International Conference on Very Large Data Bases (VLDB 85)*, pages 49–58, 1985.

[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*, pages 85–97, 1998.

[Boa97] IEEE Standards Board. IEEE std 802.11-1997 Information technology- telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements-part 11: Wireless lan medium access control (MAC) and physical layer (PHY) specifications, 1997.

[BRS03] Christian Bettstetter, Giovanni Resta, and Paolo Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 2003.

[Cha89] A. Chatterjee. Futures: a mechanism for concurrency among objects. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 562–567, New York, NY, USA, 1989. ACM Press.

[Chi05] Chipcon. Cc1000 single chip very low power rf transceiver (datasheet). `http://www.chipcon.com/files/CC1000_Data_Sheet_2_2.pdf`, 2005.

[Chr89] Flaviu Christian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: A survey. *ACM Computing Surveys (CSUR)*, 17(3):341–370, 1985.

[DSB88] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, Feb 1988.

[EE01] Jeremy Elson and Deborah Estrin. Time synchronization for wireless sensor networks. In *2001 International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, 2001.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[Eic04] Felix Eickhoff. Chronologische Ordnung von Update-Operationen verschiedener logischer Objekte bei der Datenreplikation in mobilen ad hoc Netzwerken. Diplomarbeit, Universität Stuttgart, 2004.

[epc] Electronic product code (epc). `http://www.epcglobalinc.org`.

[ER03] Jeremy Elson and Kay Römer. Wireless sensor networks: a new regime for time synchronization. *SIGCOMM Computer Communications Review*, 33(1):149–154, 2003.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical report 61, SCI Committee, 1989.

[GZ89] Ricardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by TEMPO in berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, 15(7):847–853, 1989.

[HA90] Phillip W. Hutto and Mustaque Ahamand. Slow memory: Weakening consistency to enhance concurrency in distributed shared

memories. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 302–309, 1990.

[Har01] Takahiro Hara. Effective replica allocation in ad hoc networks for improving data accessibility. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1568–1576, 2001.

[HBM04] Jörg Hähner, Christian Becker, and Pedro J. Marrón. Consistent context management in mobile ad hoc networks. In *Informatik 2004 – Informatik verbindet*, volume 1, pages 308–313, 2004.

[HBMR06] Jörg Hähner, Christian Becker, Pedro J. Marrón, and Kurt Rothermel. Maintaining update-linearizability for replicated information in manets. In *Proceedings of the First IEEE Conference Communication System Software and Middleware (COMSWARE)*, pages 1–12, 2006.

[HBR03] Jörg Hähner, Christian Becker, and Kurt Rothermel. A protocol for data dissemination in frequently partitioned mobile ad hoc networks. In *Proceedings. Eighth IEEE International Symposium on Computers and Communication (ISCC 2003)*, volume 1, pages 633–640, 2003.

[HDMR04] Jörg Hähner, Dominique Dudkowski, Pedro J. Marrón, and Kurt Rothermel. A quantitative analysis of partitioning in mobile ad hoc networks (extended abstract). In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE)*, pages 400–401, 2004.

[HHN05] Hideki Hayashi, Takahiro Hara, and Shojiro Nishio. Updated data dissemination methods for updating old replicas in ad hoc networks. *Personal Ubiquitous Computing*, 9(5):273–283, 2005.

[HL01] Jörg Hähner and Max Larsson. Integration of heterogeneous services into the facility automation platform roomcomputer. Diplomarbeit, Technische Universität Darmstadt, Germany, 2001.

[Hor05] Oliver Hornung. Design of a method for measuring the maximum delay jitter of the communication between sensor nodes. Studienarbeit, Universität Stuttgart, 2005.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[HRB04] Jörg Hähner, Kurt Rothermel, and Christian Becker. Update-linearizability: A consistency concept for the chronological ordering of events in MANETs. In *Proceedings of the 1st IEEE Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2004.

[HS78] Ellis Horowitz and Sartja Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Maryland, 1978.

[HSAA03] JoAnne Holliday, Robert Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1218–1238, 2003.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[KC00] Peter J. Keleher and Ugur Cetintemel. Consistency management in deno. *Mobile Networks and Applications*, 5(4):299–309, 2000.

[KL86] Brian Kantor and Phil Lapsley. RFC 977: Network news transfer protocol. `http://www.faqs.org/rfcs/rfc977.html`, Feb 1986.

[KMP99] Goutham Karumanchi, Srinivasan Muralidharan, and Ravi Prakash. Information dissemination in partitionable mobile ad hoc networks. In *Proceedings of 18th IEEE Symposium on Reliable Distributed Systems*, pages 4–13. IEEE Computer Society, 1999.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[LHE03] Jun Luo, Jean-Pierre Hubaux, and Patrick Th. Eugster. PAN: Providing reliable storage in mobile ad hoc networks with probabilistic quorum systems. In *Proceedings of the 4th ACM International symposium on Mobile Ad Hoc Networking and Computing*, pages 1–12, 2003.

[Lom02] Michael A. Lombardi. NIST time and frequency services (NIST special publication 432). Technical report, National Institute of Standards and Technology, 2002.

[LS88] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Cr-tr-180-88, Princton University, 1988.

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Computing Surveys*, 22(4):321–374, 1990.

[LW01] Hyunyoung Lee and Jennifer L. Welch. Applications of probabilistic quorums to iterative algorithms. In *21st International Conference on Distributed Computing Systems*, pages 21–28, 2001.

[LW03] Christoph Lindemann and Oliver P. Waldhorst. Consistency mechanisms for a distributed lookup service supporting mobile applications. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 61–68, New York, NY, USA, 2003. ACM Press.

[Mat89] Friedemann Mattern. Virtual time and global states in distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226. North Holland, 1989.

[MD88] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 123–133, New York, NY, USA, 1988. ACM Press.

[MHR01] Marie-Luise Moschgath, Jörg Hähner, and Rolf Reinema. Sm@rtlibrary - an infrastructure for ubiquitous technologies and applications. In *ICDCSW '01: Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 208–213, Washington, DC, USA, 2001. IEEE Computer Society.

[Mil94] David L. Mills. *Global States and Time in Distributed Systems*, chapter Internet Time Synchronization: The Network Time Protocol. IEEE Computer Society Press, 1994.

[MLM+05] Pedro J. Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A

flexible and adaptive framework for sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[Mos93] David Mosberger. Memory consistency models. Tr 93/11, Department of Computer Science, University of Arizona, USA, 1993.

[mot] Mica motes. `http://www.xbow.com`.

[Nee93] R. M. Needham. *Distributed Systems*, chapter Names. Addison Wesley, 2nd ed. edition, 1993.

[ns2] Network simulator ns2. `http://www.isi.edu/nsnam/`.

[NTCS99] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 151–162, 1999.

[PBV05] Santashil PalChaudhuri, Jean-Yves Le Boudec, and Milan Vojnovic. Perfect simulations for random trip mobility models. In *Proceedings of the 38th Annual Simulation Symposium (ANSS)*, 2005.

[PR99] Charles E. Perkins and Elisabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications (WMCSA)*, pages 90–100, 1999.

[Röm01] Kay Römer. Time synchronization in ad hoc networks. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 173–182. ACM Press, 2001.

[RP99] Elisabeth M. Royer and Charles E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 207–218, 1999.

[Sat93] M. Satyanarayanan. *Distributed Systems*, chapter Distributed File Systems. Addison Wesley, 2nd ed. edition, 1993.

[SBK85] Sunil K. Sarin, Barbara T. Blaustein, and Charles W. Kaufman. System architecture for partition-tolerant distributed databases. *IEEE Transactions on Computers*, 34(12):1158–1163, 1985.

[SKK+90] Mahadev Satyanarayana, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SRH97] Aman Singla, Umakishore Ramachandran, and Jessica Hodgins. Temporal notions of synchronization and consistency in beehive. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 211–220, New York, NY, USA, 1997. ACM Press.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[Sto02] Ivan Stojmenović. *Handbook of Wirless Networks and Mobile Computing*. Wiley, 2002.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, 1994.

[TRAR99] Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed consistency for shared distributed objects. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 163–172, New York, NY, USA, 1999. ACM Press.

[Tur96] Volker Turau. *Algorithmische Graphentheorie*. Addison-Wesley, 1996.

[Wan99] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, 1999.

[WJH97] Andy Ward, Alan Jones, and Andy Hopper. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, 1997.

[XWC00] Bo Xu, Ouri Wolfson, and Sam Chamberlain. Spatially distributed databases on sensors. In *Proceedings of the 8th ACM International Symposium on Advances in Geographic Information Systems*, pages 153–160, 2000.

[YLN03] Jungkeun Yoon, Mingyan Liu, and Brian Noble. Random waypoint considered harmful. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2003.

[ZK91] Hui Zhang and Srinivasan Keshav. Comparison of rate-based service disciplines. In *SIGCOMM '91: Proceedings of the conference on Communications architecture & protocols*, pages 113–121. ACM Press, 1991.

[ZK04] Yufang Zhu and Thomas Kunz. MAODV implementation for ns-2.26. Technical Report SCE-04-01, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, January 2004.

Index