Design and Evaluation of Crash Tolerant Protocols for Mobile Ad-hoc Networks

Thesis by

Einar Wiik Vollset

In Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

UNIVERSITY OF
NEWCASTLE UPON TYNE

(Defended September 30, 2005)

BLANK PAGE
IN
ORIGINAL

To Laura.

BLANK PAGE
IN
ORIGINAL

Acknowledgements

I'd like to thank my supervisor, Paul Ezhilchelvan for his support and guidance and for taking me on-board the EPSRC PACE project, Simon "the cripple" Woodman for being a genius (BubbleSearch!), a great friend and also various pleasurable zymurgy related incidents, Isi Mitrani for numerous maths lessons, Santosh Shrivastava for generous financial support, Francois Bonnet for proof reading (literally), John Lloyd for being foolish enough to fund my trip to Hawaii, Dave Cooper for sharing the pain that is GloMoSim hacking, whichever poor misguided soul decided that I was good enough for an ORS award, Rimon Barr for Java simulator related joy, and Dave Ingham for that chat which helped me through one of my darkest moments.

I'd also like to thank my parents. Ever since I was a kid my parents have always told me to pursue what makes me happy, and have supported me in doing so. For that I'm eternally grateful.

BLANK PAGE
IN
ORIGINAL

Abstract

Mobile ad-hoc networks are wireless networks operating without any form of supporting infrastructure such as base-stations, and thus require the participating nodes to co-operate by forwarding each other's messages. Ad-hoc networks can be deployed when installing network infrastructure is considered too expensive, too cumbersome or simply too slow, for example in domains such as battlefields, search-and-rescue or space exploration.

Tolerating node crashes and transient network partitions is likely to be important in such domains. However, developing applications which do so is a difficult task, a task which can be made easier by the availability of fault-tolerant protocols and middleware.

This dissertation studies two core fault-tolerant primitives, reliable dissemination and consensus, and presents two families of protocols which implement these primitives in a wide range of mobile ad-hoc networks. The performance of the protocols is studied through simulation indicating that they are able to provide their guarantees in a bandwidth efficient manner. This is achieved by taking advantage of the broadcast nature and variable message delivery latencies inherent in ad-hoc networks.

To illustrate the usefulness of these two primitives, a design for a distributed, fault-tolerant tuple space suitable to implement on mobile ad-hoc networks is presented. This design, if implemented, would provide a simple, yet powerful abstraction to the developer of fault-tolerant applications in mobile ad-hoc networks.

BLANK PAGE
IN
ORIGINAL

Contents

BLANK PAGE
IN
ORIGINAL

Chapter 1

Introduction

This dissertation is concerned with fault-tolerant reliable group communication in mobile ad-hoc networks. It studies two fault-tolerant group communication problems, reliable dissemination and consensus, in potentially highly mobile, transiently disconnected and crash-prone ad-hoc networks.

Two families of protocols, one solving the reliable dissemination problem and one solving the consensus problem are introduced. Extensive simulations covering a wide range of mobile ad-hoc network scenarios demonstrate that, contrary to what is commonly assumed, these do not incur too high overheads to be feasible.

To illustrate the usefulness of these two primitives, a design for a distributed, fault-tolerant tuple space suitable to implement on mobile ad-hoc networks is presented.

1.1 Background and motivation

A *mobile ad-hoc network* is a wireless network operating without any form of supporting infrastructure such as base-stations. The absences of such supporting infrastructure requires that the devices which constitute the ad-hoc network, called *nodes*, co-operate by forwarding each other's messages. This is made difficult by the fact that nodes typically have limited bandwidth, storage and energy resources available and, as nodes are mobile, network topology is highly dynamic.

A further complication arises if we assume that nodes can suffer *crash-failures* at any time. A node which crashes stops functioning and sends no further messages. A node in an ad-hoc network can crash by for example being dropped, kicked, drowned, set on fire or simply switched off. A protocol which can tolerate a specified number of such failures while still providing its service is called crash- or *fault-tolerant.*

A substantial amount of recent research in the area of mobile ad-hoc networks has focused on implementing unicast routing protocols (e.g. [JMB01], [Per97]), which allow two nodes in an ad-hoc network to communicate in a best-effort fashion. Another research topic which has attracted a lot of interest is how to enable users of an ad-hoc network to connect to the Internet through some

gateway node (see for example [RK03][SRB01]). The nodes in such ad-hoc networks are only loosely associated, and typically fault-tolerance is not critical. It may for example be simple to detect and remedy a node crash, or the network as a whole may not necessarily fail in accomplishing its task simply because of a few node crashes.

However, if we start to consider scenarios where the nodes which make up the ad-hoc network form a cohesive group performing some important task, the situation changes dramatically. Consider for example the case of a team of mobile robots collaborating towards a common goal, perhaps excavating a landing site on Mars or putting out a fire in a nuclear reactor.

In such scenarios the robots must be able to co-ordinate their actions despite the possibility of some robots crashing. For example, the robots must (i) be able to decide on an agreed course of action despite possibly differing initial action plans, and (ii) ensure that any important information or requests for support is received by a sufficient number of robots.

In fault-tolerant distributed computing, the first of these issues is known as the *consensus* problem, while the second is the *reliable dissemination* or broadcast problem. Reliable dissemination and consensus are two fundamental primitives in *fault-tolerant group communication*, and protocols which implement these have been shown to be vital building blocks in building sophisticated fault-tolerant applications in wired networks[GHM00].

However, it is commonly assumed that protocols which aim to provide such primitives in mobile ad-hoc networks must incur very high transmission overheads or at least require the network topology to be essentially static. The issue most frequently cited in support of this assumption is the problem of *ack-implosion*. Ack-implosion arises when a very large number of acknowledgements is generated in response to a single message (see e.g. [DDC97]).

The perception is that the limited bandwidth available in ad-hoc networks will exacerbate the ack-implosion problem causing severe congestion. In addition, the (correct) observation is made that sending even a single acknowledgement from a receiver back the originator may incur much higher transmission overheads in ad-hoc than in wired networks, and further that, unless the network is essentially static, amalgamation of acknowledgements is unlikely to work.

As mobile ad-hoc network by definition have highly dynamic network topologies and limited bandwidth, such protocols have been thought to be unfeasible (see for example [LW04][LEH03b]). This is one of the reasons why weaker, less powerful primitives have,been proposed as alternatives.

Examples of such weaker primitives include protocols which only provide probabilistic guarantees (e.g. [CRB01][LEH03a]), and protocols which assume no crash failures will occur (e.g. [VKT04][WWV01]). Protocols that only provide probabilistic guarantees, where there is a non-zero probability that a protocol terminates without doing whatever it is supposed to do, puts the onus on the application developer to check that the behaviour is as expected. This can be a significant burden, and in some critical applications may not be possible. Protocols which assume that

no crash failures will occur are useless in any scenario where crashes do occur.

It thus appears that if sophisticated, fault-tolerant applications running over mobile ad-hoc networks are to be built, solutions to the reliable dissemination and consensus problems must be sought, even if they are currently thought impractical. The alternative is for each application developer to have to implement her own application specific consensus and dissemination solutions whenever these are required. This is a complex and thus error prone process.

1.2 Contributions

This dissertation aims to demonstrate that reliable, fault-tolerant group communication protocols can be implemented in a wide range of mobile ad-hoc networks, from fully connected, relatively static networks through to highly mobile, frequently partitioned networks. In addition, it aims to demonstrate that the overheads associated with such protocols are not too high to be feasible. In doing so it makes a number of contributions.

The first contribution lies in precisely defining a system model for mobile ad-hoc networks and deriving three foundational results which arise from this model. The system model includes requirements about the connectivity of the network, and the assumption that there is a finite bound on the resources which can be used by any protocol. The requirements about network connectivity aim to be minimal so as to encompass as wide a range of network conditions as possible. A finite bound on resources is necessary because of the resource constrained nature of mobile ad-hoc networks.

The three foundational results show how no reliable dissemination protocol can guarantee that all nodes receive a message, and dictate behaviour which must be included in any reliable dissemination protocol. These results influence all the work in this dissertation, and also have wider implications for reliable dissemination protocols designed for the same environment.

The second contribution is the development of a family of crash-tolerant reliable dissemination protocols. The protocols allow the end user to specify the minimum number of nodes which should be guaranteed to receive a message. They avoid attempting to construct and maintain routing structures such as trees or meshes, and use a novel distributed method to detect that a sufficient number of nodes have received a message.

The protocols are shown to be able to guarantee delivery in even frequently partitioned and highly mobile network conditions. One of the protocols is optimised to take advantage of the peculiarities of mobile ad-hoc networks, such as the omnidirectional nature of wireless transmissions, as well as the fact that network conditions are not always extreme. Extensive simulations suggest that this protocol is able to provide its guarantees with overheads on par with an unreliable flooding protocol in relatively normal network conditions.

The third contribution is a solution to the consensus problem in mobile ad-hoc networks. The

solution is based on an existing wired network protocol which works in asynchronous, consecutive rounds. The existing protocol is shown to require a very high number of rounds to reach a decision and also involves a high transmission overhead per round. This makes the protocol impractical in ad-hoc networks. The protocol also requires a reliable dissemination protocol that can guarantee that all nodes receive a message, which is impossible when resources are finite. Finally a straight forward implementation of the existing protocol on a mobile ad-hoc network will stop working if more than a quarter of the nodes fail; the wired network implementation only stops working when a majority of the nodes fail.

A family of consensus protocols is derived by optimising the protocol in three important ways. The first optimisation removes the need for a reliable dissemination to all nodes. The second reduces the number of rounds required to reach a decision by several orders of magnitude by taking advantage of the highly variable network latencies found in ad-hoc networks. The third reduces the per round transmission overhead by 1-2 orders of magnitude, changing the required number of invocations of the reliable dissemination primitive per round from O(n) to O(1). This is done by means of a cross-layer optimisation. The protocol which combines all these optimisations is shown to reach a decision on average in 2-4 rounds over a very wide range of simulation parameters, and only stops working when a majority of nodes fail. The combined protocol is also proven correct.

The final contribution is the design of a fault-tolerant *tuple space* suitable for mobile ad-hoc networks. A tuple space is an implementation of the shared associative memory paradigm for parallel/distributed computing. Existing tuple space solutions for mobile ad-hoc networks have had to either provide no fault-tolerance or weaken the semantics compared to what one would expect from a wired network implementation. This is argued to be because of the lack of suitable fault-tolerant, reliable group communication protocols for ad-hoc networks. The design presented demonstrates how the reliable dissemination and consensus protocols can be used to develop a fault-tolerant tuple space which does not weaken the semantics of the tuple space primitives. Such a system, if implemented, is likely to make developing sophisticated fault-tolerant applications for mobile ad-hoc networks easier.

1.3 Dissertation Structure

This dissertation is structured bottom up; we first explicitly define the assumptions we make about the underlying ad-hoc network in chapter 2. The system model is the foundation upon which we base the protocols presented in this dissertation. Three theoretical results which arise from the system model is also presented and proven in this chapter.

Chapter 3 then studies the reliable dissemination problem, deriving a family of dissemination protocols which are extensively studied. The design of these is guided by the theoretical results

in chapter 2 and the correctness of the protocols is based on the assumptions made in the system model.

Chapter 4 deals with the consensus problem in mobile ad-hoc networks. A thorough review of the theory is given, and two existing wired network protocols presented. The wired network protocol depends critically on a reliable dissemination protocol of the type presented in chapter 3. The new protocols are derived by optimising the existing protocol in 3 important ways.

Chapter 5 presents a design for fault-tolerant tuple space. It shows how the two primitives presented in chapters 3 and 4 can be used to implement the first tuple space suitable for mobile ad-hoc networks which both provides equivalent semantics to what one would expect on a wired network system and also is fault-tolerant.

Finally chapter 6 concludes with future work.

Chapter 2

System model and foundational results

2.1 Introduction

When designing reliable, fault tolerant protocols for mobile ad-hoc networks, two core issues must be addressed. First the protocol must be shown to guarantee to provide the functionality it claims to provide despite a specified number of failures. This is true for any reliable, fault-tolerant protocol as without such guarantees an end user will not be able to rely on the protocol to perform correctly. Second, as ad-hoc networks are typically resource constrained, the protocol needs to be shown to provide its guaranteed functionality with reasonable overheads.

To make claims about what functionality a protocol is guaranteed to provide, either an informal correctness sketch or a more formal proof can be provided. In this dissertation, if the argument required to show correctness is relatively straight forward, a correctness sketch is used, while a more formal proof is used if the argument is more complex.

The basis of such correctness arguments is the *system model*. The system model is the assumptions made about the environment the protocol operates in, and provides the basis upon which correctness arguments are made; proofs only hold if the assumptions outlined in the system model hold.

A system model should be a realistic approximation of the physical system in which the protocols operate. This implies that the assumptions made in the model should not be too restrictive.

For example, consider a system model which makes the assumption that the maximum communication delay between any two nodes is 3 seconds, and that the proofs of some protocol relies critically on this assumption. If this protocol is deployed in a scenario where this assumption does not hold, say one where the communication delay is 4 seconds, then the proofs no longer hold and the end user cannot rely on the protocol to perform its task correctly. However, determining that such an assumption has been broken, or ensuring that it is met, may not be trivial.

For this reason, a system model should make as few and weak assumptions as possible; the fewer and weaker the assumptions are, the more operational environments the protocol can be deployed in and the easier it is for the end user to ensure that assumptions are met. In a sense, a weaker system model requires a more powerful protocol, and a more powerful protocol is easier for the end user to use, at least if this more powerful protocol does not add complexity.

The system model used throughout this dissertation is presented in the next section. The system model as described gives rise to three foundational results, described in section 2.3. These results (i) limit the number of nodes which can be guaranteed to receive a message, showing how if failures are to be tolerated, not all nodes can be guaranteed to receive a message, and (ii) prove that when the network acts as an adversary, *all* nodes which have received a message may have to actively participate in the dissemination of that message to guarantee that a sufficient number of nodes receive it.

These results guide the design of both the reliable dissemination and the consensus protocols presented in chapters 3 and 4, and influence the design of the tuple space presented in chapter 5.

In mobile ad-hoc networks, simply guaranteeing some functionality is not enough. An ad-hoc network is inherently resource constrained and thus protocols should demonstrate that they do not incur too high overheads to be practical. What is required is a way to measure whether the overheads associated with the protocol are reasonable. A simulation environment provides such a means. The environment should be as realistic as possible without being too scenario specific. The simulation environment used throughout this dissertation is presented in section 2.4.

2.2 System model

Throughout this dissertation we will consider the system as a group, $\mathcal{G}$, of nodes collaborating towards some common goal. An example of such a system could be a group of mobile robots collaborating in putting out a fire at a nuclear power station.

The nodes are assumed to be able to communicate using only the omnidirectional wireless transmission functionality of a CSMA/CA-like MAC layer protocol (e.g. IEEE 802.11b). Further, the terrain in which the nodes operate is assumed to have no fixed infrastructure for supporting communication between nodes (that is, there are no fixed base-stations to rely on), and information exchange is thus strictly limited to ad-hoc networking.

Assumptions about group membership

The membership of $\mathcal{G}$ is dynamic; a new node can join $\mathcal{G}$ and a collaborating node can leave $\mathcal{G}$ at any time. Upon joining $\mathcal{G}$, each node gets assigned a unique node identifier, *nid*. However, joining or leaving the group can only happen after the nodes of $\mathcal{G}$ have approved the join/leave requests.

For this reason, the number, n, of nodes in the group $\mathcal{G}$ at any given time can vary, but is known in the group.

The details of how nodes can join and leave is covered in detail in chapter 5. For ease of exposition, and without loss of generality, chapters 3 and 4 assumes there are no requests to join or leave and that each node has a *nid* already assigned and is aware of the number of nodes in the group.

Assumptions about node failures

This dissertation is concerned with tolerating a finite number of *crash-failures.* A node which suffers a crash-failure (or more colloquially, *crashes*) stops functioning and sends or receives no further messages. We assume that a node does not recover from a crash, though if the node is reset, it can join the group as a new node if this is desirable. We further assume that node crashes are undetectable, that is, there is no way for other nodes to detect that a node has crashed. This assumption is based on two observations: (i) it is very difficult for a node to determine if a node has crashed or is simply in another network partition, and (ii) attempting to detect crashes inevitably incurs some transmission overhead, which is undesirable in ad-hoc networks.

A node is either *correct* or *faulty.* A faulty node can crash at any moment, while a correct node is one that does not crash until the collaboration ends or the node leaves the group. We also assume that the number of faulty nodes in the system is bounded to within some known value $0 < f < n$. Thus $\mathcal{G}$ contains at least $n - f$ correct nodes at any time.

Assumptions about network connectivity

Consider two correct nodes that are in wireless range of each other. We will say that a *congestion- and collision-resilient (CCR)* channel exist between them, if at least one of a few consecutive attempts made by each node to send a packet to the other, is successful. (These attempts are typically made at the MAC layer.)

Let δ be the maximum delay which a packet can experience to be received over such a CCR channel. Two correct nodes are said to be *directly connected* at any given moment, if a CCR channel exists between them for B or more seconds starting from that moment, where $B \gg \delta$ is an application specified parameter (note that in this dissertation the "applications" are the fault tolerant protocols). The intuition is that unless two nodes directly connect when they are in range of each other, the nodes might as well have been out of range in terms of their ability to communicate.

Having defined direct connectivity, we can now define a network *liveness property* as follows: Let $\mathcal{O}$ be the set of all nodes of $\mathcal{G}$ that are correct. Let $\mathcal{P}$ be a non-empty and proper subset of $\mathcal{O}$, and $\overline{\mathcal{P}}$ be its complementary set in $\mathcal{O}$; that is, $\overline{\mathcal{P}}$ contains those nodes that are correct, but not in $\mathcal{P}$.

If no node in $\mathcal{P}$ ever has direct connectivity with any node in $\overline{\mathcal{P}}$, then $\mathcal{P}$ and $\overline{\mathcal{P}}$ are said to be *permanently partitioned* from the perspective of the application that specified B. The liveness

property disallows this by requiring at least one node in $\mathcal{P}$ to directly connect with some node(s) in $\overline{\mathcal{P}}$ at least once during $[t, t+I]$, where $I \geq B$ is finite but unknown.

Assuming the collaboration is initiated at t_0, and N_i and N_j denotes any two correct nodes in the collaborating group $\mathcal{G}$, the *network liveness requirement, NLR*, is defined as follows:

Definition. *A network satisfies the* ***network liveness requirement, NLR,*** *if no permanent partitioning of* any *set of correct nodes $\mathcal{P}$ from the collaborating group, $\mathcal{G}$, occurs at* any *instance, $t > t_0$, during the collaboration of $\mathcal{G}$. (Intermittent or transient partitions are of course allowed.) The NLR can be stated formally as:*

$\forall \mathcal{P}, \forall t \geq t_0, \exists I, B \leq I \neq \infty$: $\exists N_i \in \mathcal{P}, N_j \in \overline{\mathcal{P}}$: *$N_i$ directly connects with N_j.*

To get a feel for the what the NLR means, consider figure 2.1. In this figure there are three

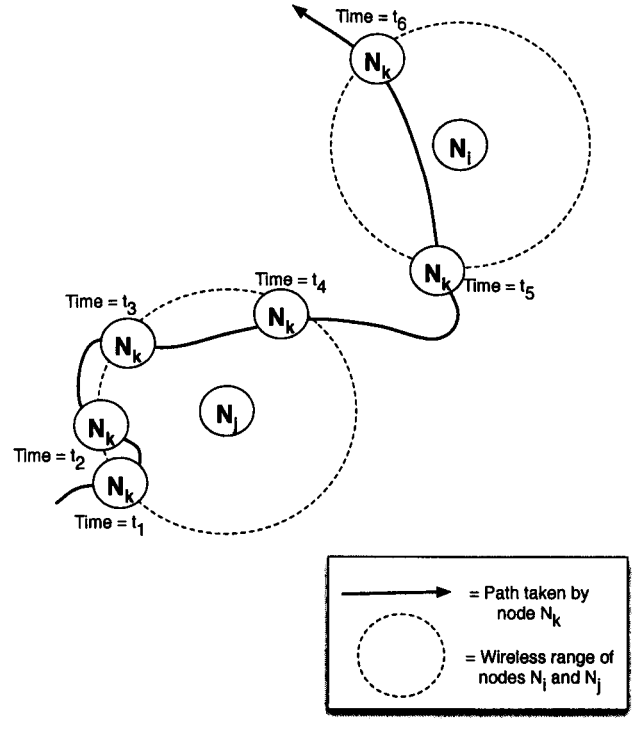

Figure 2.1: An example of when the NLR holds despite the network not being in one partition and no multi-hop paths being formed.

nodes, N_i, N_j and N_k making up the ad-hoc network. Assume that nodes N_i and N_j never move. From the figure, we can deduce that nodes N_i and N_j are unable to communicate directly; if there are no other nodes in the network, nodes N_i and N_j are permanently partitioned. Such a scenario (consisting only of nodes N_i and N_j) is disallowed by the NLR

However, if we introduce the movements and connectivity of node N_k, we get a different result. Node N_k moves along the path shown; six snapshots of the positions of node N_k from time t_1 to time t_6 is indicated. If the time between t_1 and t_2 is less than B, then node N_j and node N_k do not directly connect, as they are not in each others wireless range for long enough. However, assume that the time between t_3 and t_4 and also the time between t_5 and t_6 is greater than B. In this case, node N_k directly connects to node N_j, and node N_k directly connects to node N_i. This means the NLR is satisfied because node N_i and node N_j can communicate via node N_k. That is, they are not permanently partitioned, even though they are in separate network partitions[1].

As can be seen from figure 2.1, the NLR is a very weak assumption, as it does not specify which nodes should directly connect or say anything about the ability to establish a contemporaneous multi-hop communication path between a pair of nodes. For example, in figure 2.1 there are no multi-hop paths and the group is never in one partition, but the NLR is still satisfied. As argued above, the weaker the assumptions, the more powerful the protocol needs to be, as the protocol designer can assume less of a "helping hand" from the environment.

Assumptions about finite node resources

Mobile ad-hoc networks are typically relatively resource constrained, with for example limited available bandwidth or storage at the each node. For this reason a more powerful protocol should not come at a cost of it requiring potentially unbounded resources at each node, and it is important that protocols designed for ad-hoc networks allow nodes to free most of the resources used by the protocol within some finite time.

Specifically, for reliable dissemination protocols, it is important that a protocol is designed such that it does not require nodes to retain a copy of each message, or transmit control messages relating to the message, indefinitely. We thus define the *subsidence properties* for reliable dissemination protocols as follows:

Definition. *Let the message m be originated at time t_0. A reliable dissemination protocol satisfies the **subsidence properties** if it ensures that a correct node with m discards m (Storage Subsidence Property, SSP) and stops transmitting any packet relating to m (Bandwidth Subsidence Property, BSP) at some finite time, $t_e > t_0$.*

Note that the SSP does not require *all* information about a message to be deleted, only the message itself. Requiring nodes to delete all information about a given message, m, is near impossible;

[1]This is perhaps a bit confusing: a connected component of a network is called a "partition", and a network can be "partitioned" into several "partitions". Two nodes are said to "be partitioned" from each other if each node is in a different partition. However, they need not be "permanently partitioned", unless there is absolutely no way to communicate between the two nodes at any time, even if intermediary, moving nodes (such as N_k in figure 2.1) are used to support this communication.

consider for example how to ensure at-most once delivery of m if all information about m (including that it had already been received) had to be deleted.

2.3 Foundational results

Three general results arise from the system model as defined in the previous section. These arise based on the assumptions that (i) node crashes are considered undetectable and (ii) only permanent partitions are disallowed.

The results apply to *any* reliable dissemination protocol where these two assumptions are made, and thus has obvious implications for the reliable dissemination protocols presented in chapter 3. It also has important ramifications for the consensus solution in chapter 4 and the way the fault-tolerant tuple space is designed in chapter 5, as both these depend critically on a reliable dissemination primitive. For this reason these three results are presented separately from the chapter describing the reliable dissemination protocol.

2.3.1 Implications of the liveness property

The liveness property was designed to be as weak as possible, while still permitting a reliable protocol to provide guarantees about message delivery. The following theorem shows that in order for any reliable dissemination protocol to guarantee delivery to a specified number of nodes (called achieving a specified *coverage*), it may potentially require *all* nodes which have received a message to keep a copy of, and be ready to transmit, the message.

Definition. *A node is* ***responsible*** *for a message, m, if the node keeps a copy of m ready to transmit if instructed by the protocol.*

Note that a node can become responsible for a message whenever it receives a copy of it, not just the first time it receives the message, or when it originates it. Likewise a node can become responsible for a message, then stop being responsible for it after a while (and thus by definition not keep a copy of m or be able to transmit it), and then decide to become responsible for it the next time it receives it.

Theorem 2.1. *Suppose that nodes never crash (every node is correct). A reliable dissemination protocol which satisfies the subsidence properties cannot guarantee a coverage, $c > 1$, unless there is a time after which every node becomes responsible for m upon receiving it, until c is achieved.*

Proof. Let us hypothesize that there is a reliable dissemination protocol, $\mathcal{R}$, which in every execution satisfies the subsidence properties, guarantees a coverage of c', $1 < c' \leq n$ and also allows some nodes which have received m to not be responsible for it.

Let us consider an execution in which a node, say N_o, initiates a dissemination of a message, m, at time t_0. We define RECEIVED and NOT_RECEIVED as the sets containing the nodes which have received m and not received m respectively. Further, we define RESPONSIBLE as the subset of nodes in RECEIVED which are responsible for m, and NOT_RESPONSIBLE as the complimentary subset of the nodes in RECEIVED which are not responsible for m (and thus are unable to transmit it).

At time t_0 RECEIVED is assured to contain only the initiator of m, and NOT_RECEIVED the rest of the nodes. Consider an instance t_1 when $1 \leq |\text{RECEIVED}| < c'$, that is NOT_RECEIVED is non-empty. Since $\mathcal{R}$ is supposed to ensure a coverage of c', its execution must continue after t_1.

Consider an execution in which the ad-hoc network controls its own topology in the following adversarial manner after t_1: The ad-hoc networks forbids the direct connectivity between nodes in RESPONSIBLE and nodes in NOT_RECEIVED. Further, the direct connectivity between RESPONSIBLE and NOT_RESPONSIBLE, and NOT_RESPONSIBLE and NOT_RECEIVED occurs at different timing instants (i.e. there is no contemporaneous path between any node in $RESPONSIBLE$ and any node in $NOT\_RECEIVED$).

Observe that the ad-hoc network meets the network liveness requirement, NLR, as there is no permanent partitioning between nodes in NOT_RECEIVED and nodes in RESPONSIBLE. However, they are only able to communicate through the nodes which by definition do not keep a copy of m (the nodes which are not responsible), and thus coverage is not increased (the number of nodes in NOT_RECEIVED does not decrease).

This means that the coverage will never reach c' unless $\mathcal{R}$ makes every node in RECEIVED responsible for m.

To put it differently, in order to ensure that coverage c' is achieved in a bounded amount of time, and the subsidence properties to be satisfied, any protocol must have a timing instance after which all nodes in RECEIVED should be responsible for m (i.e. keep NOT_RESPONSIBLE $= \emptyset$) until c' is achieved.

□

Another result arises if we assume that the reliable dissemination protocol has no access to *neighbourhood knowledge*. Neighbourhood knowledge is the information provided to a node about which nodes that are in its wireless range. The following theorem dictates how a reliable dissemination protocol without any neighbourhood knowledge should be designed. Specifically, it shows that all such protocols need to include the possibility of executions where all nodes which have received a message periodically transmit it.

Theorem 2.2. *Suppose that nodes never crash (every node is correct) and have no knowledge about neighbourhood. A reliable dissemination protocol which satisfies the subsidence properties cannot*

guarantee a coverage, $c > 1$, unless there is a time after which every node with m transmits m at least once every τ time $\tau < (B - \delta)$, until m is received by all c nodes.

Proof. Let us hypothesize that there is a reliable dissemination protocol which preserves the subsidence properties and also ensures coverage of c', where $1 < c' < n$, in every execution.

Let us consider an execution in which m is originated at time t_0. Let us also define RECEIVED and NOT_RECEIVED as the sets containing nodes that have and have not received m at any given moment respectively. Note that at time t_0, RECEIVED contains only the initiator and NOT_RECEIVED all other nodes.

A node that has received m is said to be *active* (on m) during $[t, t+D]$, for some finite $D > B$, if it transmits m at least once every τ time, $0 < \tau < (B - \delta)$, during $[t, t+D]$. It is said to be *inactive* on m during $[t, t+D]$ if there exists one or more occasions during $[t, t+D]$ in which the node does not transmit m for a duration of at least $(B - \delta)$ time.

Consider an instance t_1, $t_1 \geq t_0$ when $1 \leq |\text{RECEIVED}| < c'$. Since the protocol ensures a coverage of c', its execution must continue after t_1. Suppose that the ad-hoc network controls its topology in the following adversarial manner after t_1: it delays the development of CCR-channels between nodes of RECEIVED and those of NOT_RECEIVED until the protocol chooses to keep some nodes of RECEIVED inactive. When some nodes of RECEIVED are made inactive, the ad-hoc networks allows CCR-channels to be formed only between such inactive nodes and some nodes of NOT_RECEIVED for exactly B time and after δ time has elapsed since the nodes of RECEIVED started becoming inactive.

Since nodes have no neighbourhood knowledge, they cannot sense the emergence of CCR-channels connecting them with other nodes . Therefore the protocol cannot make use of the CCR-channels between inactive nodes of RECEIVED and nodes of NOT_RECEIVED for disseminating m any further. While coverage c' does not increase, CCR-channel formation helps the ad-hoc network to meet the NLR. That is, even though the liveness requirement is met, coverage c' does not increase because the adversarial ad-hoc network exploits the protocol feature of keeping some nodes of RECEIVED inactive occasionally.

From the above discussion it is obvious that the required coverage, c' can be guaranteed to be achieved, only if the protocol, at or some time after t_1, forbids the nodes of RECEIVED from becoming inactive until m is received by all c' nodes. Forbidding node inactivity leads to increase in coverage when m is disseminated to nodes in NOT_RECEIVED over the CCR-channels which the ad-hoc network is obliged to bring about within every finite I. □

2.3.2 Implication of undetectable node crashes

The following theorem shows how, given that node crashes are undetectable, it is impossible for any reliable dissemination protocol to both satisfy the subsidence properties and also guarantee that more than $n - f$ nodes receive a given message, even if no nodes actually crash during the execution of the protocol.

Recall that the subsidence properties requires there is some finite time, t_e, after which nodes that have received a message, m, are not required to retain it, and where no further packets relating to m are to be transmitted.

Theorem 2.3. *It is impossible for any crash-tolerant reliable dissemination protocol that satisfies the subsidence properties to guarantee that more than $n - f$ nodes (including the originator) receive a message m originated by a correct node even if less than f nodes crash before t_e.*

Proof. By contradiction. Assume there is such a protocol, $\mathcal{R}$, which satisfies the termination property and which can guarantee that more than $n - f$ nodes receive a message m if less than f nodes crash before t_e. Let $\mathcal{F}$ be any set of f nodes which does not include the originator of m, and $\overline{\mathcal{F}}$ be its complementary subset. Further, let t_0 be the time at which the originator of a m starts disseminating it.

Consider two executions of $\mathcal{R}$:

Execution 1: All nodes in $\mathcal{F}$ have crashed before t_0. Since there are only $n - f$ correct nodes in the system, $\mathcal{R}$ cannot guarantee that more than $n - f$ nodes receive m.

Execution 2: No nodes have crashed before t_0, and no nodes crash until after t_e. However, the network (acting as an adversary) keeps all nodes of $\mathcal{F}$ outside the wireless range of every node in $\overline{\mathcal{F}}$ until after t_e. Nodes of $\mathcal{F}$ thus never receive m.

Execution 2 is possible as $\mathcal{R}$ has no way of distinguishing between the case where all nodes in $\mathcal{F}$ have crashed, and the case where all nodes in $\mathcal{F}$ have not crashed, as: (i) nodes of $\mathcal{F}$ do not execute the protocol and (ii) there is no way for any node in $\overline{\mathcal{F}}$ to detect if a node in $\mathcal{F}$ has crashed.

Therefore to satisfy the SSP, $\mathcal{R}$ must chose some finite t_e after which the nodes in $\overline{\mathcal{F}}$ can stop being responsible for, and thus delete m. The network acting as an adversary can then directly connect $\mathcal{F}$ and $\overline{\mathcal{F}}$ immediately after the chosen t_e and thus satisfy the NLR. This is true whatever (finite) t_e is chosen by $\mathcal{R}$.

Therefore, only $n - f$ nodes receive m in both executions. This contradicts the hypothesis that $\mathcal{R}$ can guarantee that more than $n - f$ nodes receive a message m, if less than f nodes crash before t_e since no nodes crashed before t_e in execution 2. Hence the theorem.

□

2.4 A note on the simulation environment

In an ideal world, all proposed protocols for ad-hoc networks would be tested on a real ad-hoc network. Unfortunately this is near impossible as at time of writing so few non-trivial ad-hoc networks are in use, and even fewer of these are readily available to researchers. For this reason most protocols are only tested in simulators.

Ad-hoc network simulators are widely used, as a brief survey of the proceedings of conferences such as ACM MobiCom, MobiHoc and IEEE InfoCom will indicate. However, simulators have some issues to be aware of.

Cavin et al. showed that even relatively simple protocols could get *qualitatively* different results if implemented on different simulators[CSS02], and they argue that protocols should ideally be implemented on more than one simulator to obtain more reliable results. This has actually been done for some of the work in this dissertation, though more by accident than design. The initial performance study of the reliable dissemination was done using GloMoSim[ZBG98], a widely used simulator developed in a parallel version of the C programming language. However, for reasons of programmer efficiency, later implementations (and all work on the consensus protocols) was done in the JiST/SWANS simulator, a Java based discrete event simulator[BHR05]. The results for the GloMoSim and JiST/SWANS simulator were for all intents and purposes identical, thus alleviating this concern.

A further difficulty arises in getting realistic mobility models; most performance studies assume the nodes move randomly in a confined, 2-dimensional space with no obstacles. Clearly, most people (if we assume the nodes are PDAs) or even mobile robots do not move randomly. However, as so few ad-hoc networks exist, it is difficult to know exactly what should replace the random movement models. This dissertation therefore uses the most widely used random mobility model, the Random Waypoint model throughout. This has the advantage that the experiments should be relatively easy to replicate.

Finally, probably the most critical issue is that protocols designed for mobile ad-hoc networks must avoid relying on properties which may only exist in simulation. For example, Kotz et al. observe[KNE03] that a number of assumptions, such as assuming that "hello packet" propagation is symmetric, or that signal strength is a simple function of distance, are often made by protocol designers, but that these typically do not hold in a "real" wireless network.

Having an explicit system model as outlined in the previous section alleviates this problem somewhat; the model presented in section 2.2 explicitly states the assumptions we make, and none of these rely on anything that is likely to exist only in simulation. Further, we do not rely solely on simulation to validate our protocols; proofs or correctness sketches also accompany all protocols.

The default simulation parameters used throughout this dissertation, unless otherwise specified,

Table 2.1:

Default simulation parameters	
Simulator	SWANS v1.0.1 [BHR05]
Simulation time	3000s
Node placement	Random
Number of nodes	50
Choice of initiators	Random
Percentage of actual crashes	10% ($f = 5$)
Choices of crashing nodes	Random
Wireless range	250m
Area size	1000m x 1000m
Mobility model	Random Waypoint
Node speed [min, max]	[1m/s, 10m/s]
Pause time	0s
Nodes' buffer size	50 packets
Fading model	Rayleigh
Pathloss model	Two-Ray

are shown in table 2.1. Note that each simulation was run at least 10 times and that the first 1000s of simulation time was discarded to remove initial bias.

Throughout this dissertation we will be varying the following 3 parameters:

Wireless range: Changing the wireless range of the nodes in the simulation changes the *density* of the network and has a strong impact on the likelihood of the network being transiently partitioned. In general, a low wireless range (e.g. 150m) results in the network being partitioned quite frequently (often into several partitions), while a high wireless range (e.g. 350m) is likely to cause the network to be connected.

Node speed: Varying node speed has a direct impact on the rate of change of topology; high maximum node speeds cause the network graph to be very dynamic, where as low speeds cause the network to remain fairly static.

Number of nodes: Changing the number of nodes in the system shows if the protocol is scalable. Note that when changing the number of nodes in the system we also modify the size of the area in which the nodes operate. This is so that the density of the network remains constant and we are able to isolate the effects of varying the number of nodes.

An indication of the network conditions experienced at low densities is given in figure 2.2 which shows how the number of network partitions varied throughout a typical simulation run when the wireless range was 150m. Note that 1 partition implies the network is not partitioned (is in one connected component), which does not happen in this case. This means that establishing contemporaneous multi-hop paths between all nodes would have been impossible in this simulation run. However, this does *not* necessarily mean the network is permanently partitioned; for that to occur

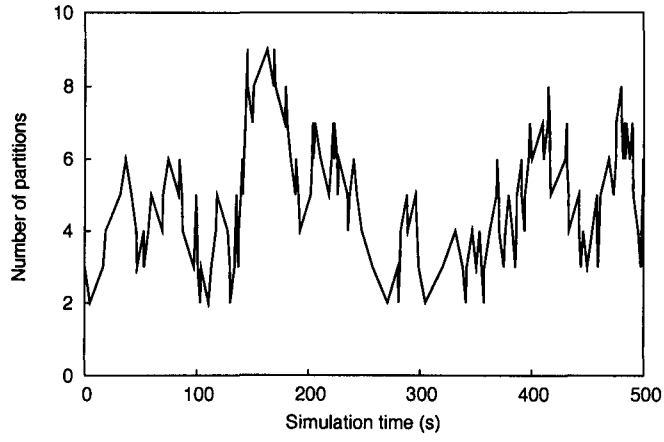

Figure 2.2: Number of partitions in a typical simulation run with wireless range = 150m and average speed = 5 m/s

there would have to be a complete segregation of nodes for the duration of the simulation run. In the simulation run depicted in figure 2.2 nodes moved frequently between partitions and thus avoided this.

Finally, a protocol designed for ad-hoc networks should incur a limited amount of *transmission overhead*, so as to reduce interference, increase throughput and potentially save energy. In addition, it should have low *latency*, so as to provide its service quickly. Thus these are the two main measures of interest to us in this dissertation. In the few instances were we are interested in other measures, these will be explained in detail.

2.5 Conclusion and summary

This chapter has defined the assumptions made about the environment which the protocols presented in the rest of this dissertation operates in. In particular, in the rest of this dissertation we will assume that (i) node crashes are undetectable , and (ii) that the mobile ad-hoc network is not permanently partitioned.

These assumptions give rise the following foundational results:

1. Any reliable dissemination protocol must include the potential for executions where all nodes retain a message, ready to transmit if instructed by the protocol (theorem 2.1).

2. Any reliable dissemination protocol which has no neighbourhood knowledge must include the potential for executions where all nodes periodically transmit a message (theorem 2.2).

3. No reliable dissemination protocol can guarantee that more than $n - f$ nodes receive any given message, where n is the number of nodes in the network and f is the number of tolerated failures, even though less than f nodes actually crash in any given execution (theorem 2.3).

These results influence the design of reliable dissemination protocol presented in the next chapter, but also has important implications for the consensus solution in chapter 4 (which makes use of the reliable dissemination primitive) and the design of the tuple space in chapter 5 (which makes use of both the consensus and the reliable dissemination primitives).

Finally, this chapter has described the simulation environment used to evaluate the performance of the reliable dissemination and the consensus protocols, including discussing some important caveats to be aware of when simulating mobile ad-hoc networks.

Chapter 3

Reliable dissemination in ad-hoc networks with crash failures

3.1 Introduction

In this chapter we will study the problem of how to reliably disseminate messages to a group of nodes in mobile ad-hoc networks where transient network partitions and undetectable node crashes can occur. The reliable dissemination protocols presented here allows the user to specify the minimum number of nodes a message should be delivered to, called the desired *coverage*. The protocols guarantee that the specified minimum coverage is achieved despite node failures and network partitions.

Such protocols are useful when developing fault-tolerant applications. Without the strong guarantees provided by these protocols, the burden of ensuring message delivery gets pushed from the middleware to the application level. Doing so necessarily makes application code more complex, and therefore more error prone. In addition, as highlighted by Friedman[Fri03], it is much harder to prove the correctness of applications which rely on protocols that provide weaker guarantees. Finally, a reliable dissemination protocol is a crucial building block when attempting to solve harder distributed problems such as agreement.

Previously proposed reliable dissemination protocols for ad-hoc networks typically create and maintain some form of routing structure over which the reliable dissemination is performed. The overheads associated with creating and maintaining such structures can become high given the volatile nature of ad-hoc networks, particularly as these structures are used both to disseminate a message, and to send acknowledgements back to the node which originated the message. Sending acknowledgements back to the originator (or some other "core" node) is necessary as typically only the originator has the ability to determine when sufficient coverage has been achieved.

However, as shown in theorem 2.1, creating multi-hop routes between two nodes, much less between the originator and every node in the network, is by no means guaranteed to be successful in the types of ad-hoc networks considered in this dissertation. What is required instead is (i) a method

by which the ability to detect when sufficient coverage has been achieved can be distributed among the nodes in the network, and (ii) a dissemination strategy which does not rely on the existence of routes, but still guarantees sufficient coverage.

This chapter presents a distributed method to detect when the protocol can terminate. This method addresses (i) above by requiring each node to add a unique "signature" to every message, thus allowing any node to determine if sufficient coverage has been achieved. This method removes the reliance on a single node to perform this task found in other protocols (e.g. [TOL02][PR97][GS99]).

We present two core dissemination strategies, one *reactive* and one *proactive*, which are able to guarantee that sufficient coverage is achieved (and thus address (ii) above). When combined with the distributed termination method, these two strategies render two reliable dissemination protocols for ad-hoc networks. The performance of these two protocols is studied, providing some insights into the network conditions under which they perform well.

The lessons learnt from the two basic dissemination strategies are used in developing a third, optimised protocol, combining the benefits of the proactive and reactive approach. The extensive simulations performed indicate that the optimised protocol meets our objective of developing a fault-tolerant reliable protocol which has overheads on par with a simple, unreliable, flooding protocol. This suggests that the overheads associated with reliable dissemination are not inherently too high to be feasible in ad-hoc networks.

3.2 Reliable dissemination in ad-hoc networks without routes

This section presents and studies the problem of how to develop crash-tolerant reliable dissemination protocols when routing structures are not guaranteed to work. After a formal definition of the problem to be solved, the distributed method to detect that sufficient coverage has been achieved is presented. This method underpins the remainder of this chapter.

We then combine this method with two core dissemination strategies to derive two quite simple reliable protocols. These protocols satisfy theorem 2.1 by making all nodes responsible from the outset. The performance of the resulting protocols is studied and some general conclusions drawn.

3.2.1 Problem definition

Recall that a faulty node can crash at any moment, while a correct node is one that does not crash. The number of faulty nodes is assumed to not exceed some known bound, f (see section 2.2).

For the dissemination of a message, m, to a specified number of nodes, $1 < k \leq (n - f)$, *a crash-tolerant reliable dissemination protocol* offers the following guarantees despite at most f, $0 < f < n$, faulty nodes:

Delivery : if the initiator is correct (i.e. does not crash), or at least one correct node receives m, then at least k nodes (including the initiator) receive m within some bounded time.

Termination : if a node that receives m is correct, it discards m and stops transmitting any packet relating to the dissemination of m within some finite time.

There is no prior knowledge of which nodes crash and when; all or some of the f crashes may have occurred before the dissemination was initiated or may occur during the protocol execution.

Therefore it is possible that the initiator crashes before completing the protocol and the few nodes, if any, that receive m also crash. In that case, no delivery guarantees can be given; however, if a correct node receives m, then at least k nodes are guaranteed to receive m.

The termination property is essentially the same as the subsidence properties and is crucial in mobile ad-hoc networks where bandwidth, energy and node capabilities are typically limited. In wired networks, reliable protocols with similar properties, such as eventually not having to transmit m, are called *quiescent* [ACT00].

3.2.2 Distributing the ability to detect termination

The reliable dissemination of a message, m, can terminate once it is guaranteed that enough nodes have received m.

To distribute the ability to detect termination, all responsible nodes maintain a set of "signatures" of nodes it knows to have received m. A signature is a unique identifier associated with a node. This set of signatures is known as the "knowledge of m", and is denoted as $K(m)$. A node's $K(m)$ always contains its own signature. Whenever a node transmits a copy of m, it sets a header field, $m.K$ to its current $K(m)$. Whenever a node receives a copy of m, it merges the contents of the received $m.K$ with its $K(m)$.

The originator of m, specifies the minimum number of nodes, k, which m should be delivered to by setting a header field, $m.k$. This means any node can compare the number of signatures in $K(m)$ and know when enough nodes have received it. The ability to determine that the dissemination can terminate thus gets distributed throughout the network instead of residing at any single node. Detecting that enough nodes have received m is called the *realisation of m.*

If all nodes in the network are assigned unique *node identifiers*, *nids*, which are monotonically increasing integers from 0 to n, $K(m)$ can be represented as a bit vector of size n; a node's signature is represented as a 1 in the bit vector in the position indicated by its *nid.* Assigning node ids to satisfy this constraint in a fault-tolerant manner can be achieved by means of a consensus protocol (see next chapter) and is explained in chapter 5. Throughout this chapter and the next we will assume all nodes have been assigned such *nids*.

The above method distributes the ability to detect if and when sufficient coverage has been achieved. However, it does nothing to ensure that sufficient coverage is actually guaranteed to be achieved.

3.2.3 Strategies for ensuring sufficient coverage

The alternative to creating routing structures is to make optimal use of the direct connectivity experienced by each node. There are two basic approaches: (i) the *proactive* approach; a responsible node periodically transmits the whole message regardless of which nodes are in its neighbourhood, and (ii) the *reactive* approach; a responsible node only transmits the minimum necessary based on which nodes are in its neighbourhood and what it knows about these.

The following two sections describe two protocols which use these strategies. Note that in both the descriptions we assume that all nodes know the values of n and f. This assumption is also based on the assumed use of the consensus protocol to assign *nids*.

Proactive dissemination

The proactive dissemination protocol, PDP, is the simplest of the two. It only requires that nodes transmit m periodically once every β seconds. β is a configurable parameter such that $B \geq 2(\beta+\delta)$. Recall that B is the minimum time a "direct" connection lasts, and δ is the maximum transmission delay of a message (see section 2.2). Requiring $B \geq 2(\beta + \delta)$ thus allows a node to both receive a message, and respond to that message during the direct connectivity.

By periodically transmitting m every β seconds, PDP ensures that direct connectivity is taken advantage of (by theorem 2.2, such periodic transmissions may be required in any protocol which has no neighbourhood knowledge).

The protocol steps are as follows:

1. **Starting the dissemination:** The initiator initialises the set of signatures $K(m)$ to be all 0's, and sets its own bit to 1. It generates a unique id, $m.id$, for the message and sets the $m.k$ to the minimum number of nodes it wants to guarantee receives m (note that $k \leq (n - f)$). It then becomes *responsible* for m.

2. **Being responsible:** Every message a node is responsible for is retained, and the message is transmitted periodically every β seconds. Immediately prior to transmission, $m.K$ is set to $K(m)$ containing the latest knowledge a node has about which nodes have received m.

3. **Receiving a message:** Upon receiving a message for the first time, a node initialises $K(m) = m.K$ and sets its own bit to 1. It then becomes responsible for m. Every time it receives a message it is responsible for, it merges the received $m.K$ with $K(m)$.

4. **Realising a message:** every time a node updates $K(m)$, either because it adds its own signature to it, or because it merges $K(m)$ with an incoming $m.K$, it checks if $|K(m)| \geq m.k$. If it is, the node realises m. Realising m involves canceling the periodic transmission of m and deleting m from stable storage. Every time a node receives a packet relating to a message which has been realised, it transmits a small realisation packet, *R_Pkt*, containing the message's unique id. A node which receives an $R\_Pkt(m.id)$ takes the same action as if it had detected $|K(m)| \geq m.k$ itself.

Correctness arguments for PDP (sketch)

Consider an execution where a message, m, is initiated at time t_0, and where either the initiator is correct, or a correct node receives m. Let $t \geq t_0$ be a timing instance during this execution.

Delivery: Assume that no correct node that has m has realised m at t. Choose $\mathcal{P}$ to be any non-empty subset of all the correct nodes with identical $K(m)$ at t. (By the nature of the execution considered, there is at least one such singleton $\mathcal{P}$.) Let $\overline{\mathcal{P}}$ denote the set of all those correct nodes not in $\mathcal{P}$ at t. Given this, $\overline{\mathcal{P}}$ cannot be empty as otherwise $\mathcal{P}$ contains all correct nodes (of which there are at least $n - f$) and all of them have identical $K(m)$s; since a node has 1 for its own bit in its $K(m)$, all nodes in $\mathcal{P}$ must have realised m which is not the case at t.

When node $N_i \in \mathcal{P}$ and node $N_j \in \overline{\mathcal{P}}$ directly connect, either N_j receives m for the first time or the nodes exchange their different $K(m)$s. Thus, as the execution progresses with no node realising m, each occurrence of direct connectivity increases by at least 1 the number of 1-bits in the $K(m)$ of some correct node. Since $(n - f)$ is finite, some correct node(s) must realise m within some finite duration.

Termination: Assume not all correct nodes have realised m at t. Let $\mathcal{P}$ be the set of those correct nodes that have received, but not realised m at t. Let $\overline{\mathcal{P}}$ denote the set of all those correct nodes not in $\mathcal{P}$ at t. When node $N_i \in \mathcal{P}$ and node $N_j \in \overline{\mathcal{P}}$ directly connect, either: (i) N_i will realise m if N_j has realised m, or (ii) N_j receives m for the first time.

Since the number of correct nodes is finite, case (ii) will cease and all correct nodes that have m will realise m in finite time. After this, no more realisation packets will be transmitted.

Reactive dissemination

The reactive dissemination protocol, RDP, assumes that information about which nodes are in a node's neighbourhood is provided without any additional transmission overhead (e.g. by MAC-layer signaling). This information is represented as a set of *nids*. Let $Neigh$ denote this set. The basic strategy is to transmit m only when a node observes someone in its neighbourhood who it does not know to have received m (e.g. when $Neigh - K(m) \neq \{\}$). This is evaluated once every β seconds, where $B \geq 2(\beta + \delta)$ as above.

Although sufficient to guarantee that enough nodes receive m, the basic strategy is insufficient to guarantee that a node realises m. Consider for example an ad-hoc network of 3 nodes ($n = 3$) arranged in a straight line, with each node having only its immediate neighbour(s) in its $Neigh$. Assume the middle node originates m. When it transmits m, each of its two neighbours (the end nodes) receives m and forms $K(m)$ with two 1 bits (as in PDP above). Since each end node has only the transmitting node in its $Neigh$, it will find $Neigh - K(m) = \{\}$ and choose not to transmit m. If $f = 0$, the originator, which cannot know (for sure) whether its neighbours have received m, cannot realise m even though all have received it.

It thus follows that additional data structures and control packets are needed.

Nodes maintain two more bit vectors: knowledge on the propagation knowledge of m ($KK(m)$) and knowledge on the realisation of m ($KR(m)$). If node N_i knows that a node N_j has the same $K(m)$ as itself, then its $KK_i(m)[j]$ is set to 1; otherwise, its $KK_i(m)[j]$ will retain the initialised value of 0. Similarly, if node N_i knows that node N_j knows of the realisation of m, then its $KR_i(m)[j]$ is set to 1; otherwise, $KR_i(m)[j]$ will retain the initialised value of 0.

Note that, unlike in $K(m)$ and $KR(m)$, the number of 1s in $KK(m)$ can decrease, because $KK(m)$ will be reset every time $K(m)$ changes. Similarly, while $KK_i(m)[j] = 1$, node N_j may have added more 1s to its $K_j(m)$ without node N_i being aware of this addition. Therefore, the only certainty that N_i can derive from $KK_i(m)[j] = 1$ is that N_j had the same $K(m)$ as itself at some point in the past.

Packets of the following types may be transmitted by a node: *K_pkt(m.id)* containing the *m.id* and the transmitting node's knowledge $K(m)$ and $KK(m)$; *R_Pkt(m.id)* containing the *m.id* and the transmitting node's $KR(m)$; and *R_Ack(m.id)* which acts as an acknowledgement to receiving an *R_Pkt(m.id)*.

The protocol steps are as follows:

1. **Starting the dissemination:** The initiator initialises $K(m)$, $KK(m)$ and $KR(m)$ to be all 0's, and sets its own bit to 1 in the two former. It generates a unique id, *m.id* and sets the $m.k$ as above. It then becomes responsible for m.

2. **Being responsible:** Every message a node is responsible for is retained, and the following tests are evaluated every β seconds:

   if($Neigh$ contains nodes not in $K(m)$): The full message, m, with $m.K = K(m)$ is transmitted.

   else if ($Neigh$ contains nodes not in $KK(m)$): Only K_Pkt(m.id), containing $K(m)$ and $KK(m)$ is transmitted.

   else: Nothing is transmitted.

3. **Receiving a message:** Upon receiving a message for the first time, a node initialises $K(m) = m.K$ and sets its own bit to 1. It also initialises $KK(m)$ and $KR(m)$ to be all 0's, and sets its own bit to 1 in the former. It then becomes responsible for m. Every time it receives a message it is responsible for, it merges the received $m.K$ with $K(m)$, and if this changes its $K(m)$ it resets $KK(m)$ to be all 0's and sets its own bit to 1.

4. **Realising a message** : every time a node updates $K(m)$, either because it adds its own signature to it, or because it merges the $K(m)$ with the $m.K$ of an incoming message, it checks if $|K(m)| \geq m.k$. If it is, the node *realises* m. Realising m involves deleting m from stable storage, and setting its own bit in $KR(m)$ to 1. A node can also realise m upon receiving an $R\_Pkt(m.id)$ or an $R\_Ack(m.id)$

5. **Having realised a message** : If a message has been realised, a node evaluates the following test every β seconds until $|KR(m)| = n$:

   if($Neigh$ **contains nodes not in** $KR(m)$)**:** An $R\_Pkt(m.id)$, containing $KR(m)$, is transmitted

   A node which has realised m transmits an $R\_Pkt(m.id)$ whenever it receives m of $K\_Pkt(m.id)$, and transmits an $R\_Ack(m.id)$ whenever it receives an $R\_Pkt(m.id)$

RDP is not as simple as PDP, and in addition to requiring neighbourhood knowledge, also suffers from one other drawback; because a node must retain $KR(m)$ until it knows that all nodes have realised m, it is possible for all nodes to have to maintain this data structure, and perform the evaluation described in step 5, for ever. The reason is that if even just one node has crashed before realising m, the number of signatures in $KR(m)$ will never reach n.

This is impractical, but does not violate the termination property from section 3.2.1; if some nodes have crashed the correct nodes will never find the (crashed) nodes in its neighbourhood, and therefore never actually transmit anything after all correct nodes have realised m (and thus bandwidth subsidence is maintained). Storage subsidence is also maintained, as the actual message, m, can be discarded, it is only the $KR(m)$ which cannot be.

Correctness arguments for RDP (sketch)

Claim: Consider node N_i and node N_j with $K_i(m) \neq K_j(m)$ at some time t during an execution. It is not possible for both $KK_i(m)[j] = 1$ and $KK_j(m)[i] = 1$ at t.

With no loss of generality, suppose that $KK_j(m)[i] = 1$ at t. This is possible only if N_j has known in the past that $K_i(m) = K_j(m)$. But at t, $K_i(m) \neq K_j(m)$. That is, N_i has increased its $K_i(m)$ which must have caused its $KK_i(m)$ to be reset. Further, N_i could not have learnt that N_j also increased its $K_j(m)$ in the same way. Therefore, $KK_i(m)[j]$ cannot be 1, and can only be 0.

The claim suggests that when unrealised nodes N_i and N_j with different $K(m)$ experience direct connectivity, at least one of them will transmit at step 3. Further, if $KK_i(m)[j] = 0$ and $KK_j(m)[i] = 1$, then N_j gains more 1-bits. The rest of the arguments can be constructed by choosing appropriate $\mathcal{P}$ as was done for the proactive protocol above, and is omitted.

3.2.4 Comparing basic dissemination strategies

Studying the performance of PDP and RDP provides an insight into the advantages and disadvantages of the proactive, and the reactive dissemination strategies. In this section we will study their relative performance using one simple optimisation; instead of static periods of β seconds between actions (whether transmitting m, as in PDP, or evaluating whether to transmit, as in RDP), each node will pick a random interval between 0 and β and then perform the action after this interval. After the action has been performed, a new random interval is chosen if required.

Recall that we are interested in the transmission overheads and the latency of the protocols. In general, for both the protocols, increasing the value of β decreases transmission overheads, while increasing the latencies. A value of $\beta = 5$ seconds provides a good balance between these two, and is used throughout the dissertation.

The measure of overheads is based on how many bytes get put on the network by all nodes during the execution of the protocol. The total number of bytes is then divided by k as guaranteeing delivery to 50 nodes generates more overheads than guaranteeing delivery to 2, independently of which protocol is being studied. Similarly, guaranteeing delivery of a message with a payload of 1024 bytes causes more bytes to be put on the network than guaranteeing delivery of one with a payload of 64 bytes. For this reason the total number of bytes is also divided by the payload size, yielding the formula for overhead shown below:

$$Overhead = \frac{\text{total number of bytes transmitted by all nodes}}{\text{k * size of payload}}$$

A good way to get an intuition for how this measure works is to consider how an "idealised" flooding protocol would perform. An "idealised" flooding protocol in this case is a protocol where every node transmits each message once. Such a protocol would have an overhead of approximately 1 assuming that $k = n$[1]. Wherever appropriate, a line indicating how such an idealised protocol would perform for $k = n$ has been added to the graphs for reference. It is important to note however, that a simple flooding protocol is not reliable and that this line is only intended as a reference point in terms of transmission overheads.

Latency for PDP and RDP is defined as how long it takes for the first node to realise that k nodes have received a given message. This measure was chosen as it encompasses both how long it

[1]Actually slightly above 1, as even a simple flooding protocol needs packet headers which would add to the overheads

takes for at least k nodes to receive m, and how long it takes the first node to determine that this has happened. Latency is measured in milliseconds.

Figures 3.1 and 3.2 shows the impact on overhead and latency of varying the wireless range of the nodes while keeping all other variables constant (recall that the default variables used throughout this dissertation is listed in table 2.1 in section 2.4).

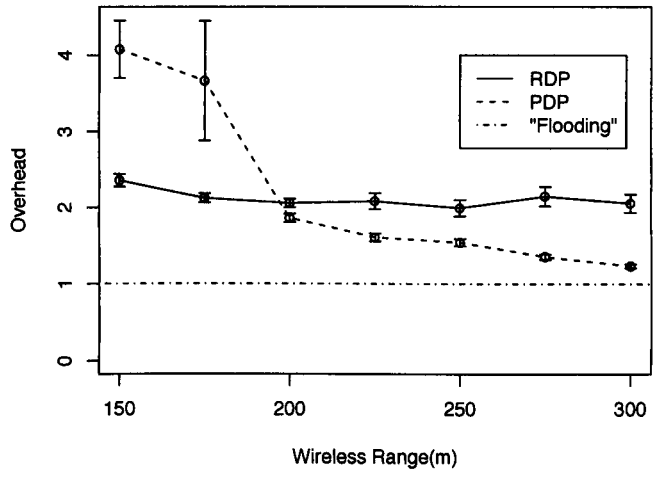

Figure 3.1: Overhead vs. Wireless range for PDP and RDP. Max speed = 5m/s, $k = 45$.

There is a clear trend in these figures; RDP outperforms PDP at low densities, while PDP outperforms RDP at higher densities.

At low densities PDP performs poorly as the simple strategy of transmitting the whole message periodically results in a number of transmissions where there are no nodes in the neighbourhood to receive the message. RDP performs better as it is able to make use of its neighbourhood knowledge to avoid transmissions when there are no nodes in a node's neighbourhood. It is thus evident that at low densities, adopting a more reactive strategy is beneficial both in terms of latency and overhead.

At high densities PDP performs better, as its strategy of proactively pushing the message (including $K(m)$) leads to rapid realisation of m, at costs not much greater than idealised flooding. RDP's use of neighbourhood knowledge leads to quite a few redundant transmissions, including redundant transmissions of m. What appears to happen is that responsible nodes determine that nodes in their neighbourhood are not known to have received m, and thus transmit it. However, very often these nodes have received m from elsewhere and so the transmission is redundant. In addition, a node in RDP strives to ensure that all nodes in its neighbourhood has the same $K(m)$ as it self; at high densities this is nearly never achieved prior to realisation, though in the attempt a lot of redundant control information gets transmitted. It thus appears that the proactive approach

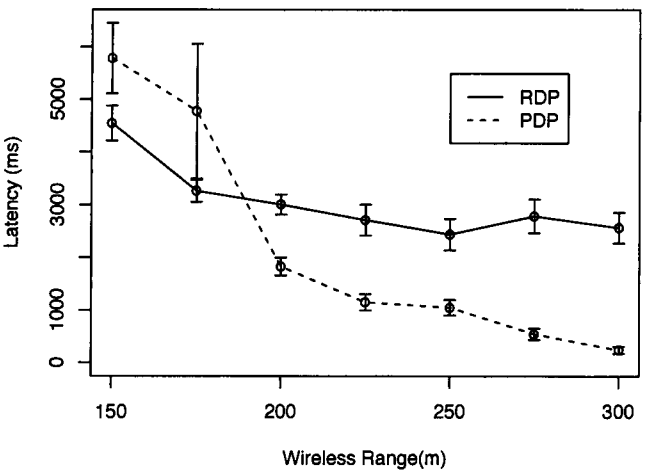

Figure 3.2: Latency vs. Wireless range for PDP and RDP. Max speed = 5m/s, $k = 45$..

has the edge at higher densities.

Figure 3.3 shows the impact of node speeds on overheads when the wireless range is 250m. The figure suggests that nodes speeds have a negligible impact on the overhead of both protocols. This was also found to be true at both higher and lower densities than shown in this figure. Similar results were observed with regards to the impact of node speeds on latency. The reason there is so little impact is that none of the protocols make use of any routing structure which may break and then needs repairs in order to work; the protocols are "opportunistic" in making use of what connectivity there is. In general, this a very beneficial feature for protocols designed for ad-hoc networks.

Figures 3.4 and 3.5 indicate how the protocols scale. The overhead associated with PDP increases slightly as the number of nodes increase, while the overhead is reduced for RDP.

The reason the performance of PDP suffers is that increasing the number of nodes while keeping density constant (by varying the size of the simulation area) means a smaller percentage of the nodes in the network hear each transmission of m (and thus $K(m)$). As fewer nodes hear each transmission, more transmissions must occur before realisation happens. When more transmissions are required both overhead and latency increases.

For RDP, keeping the density constant means the same number of nodes are in each node's neighbourhood, and the number of transmissions of m therefore remains relatively unaffected by the change in scale. As in PDP, the number of transmissions overall increase, though typically increasing the number of nodes in the network while retaining the same density means these extra transmissions are K_Pkts. This increased number of transmissions is more than compensated for by the fact that the measure of overheads divides by (the increasing) value of k.

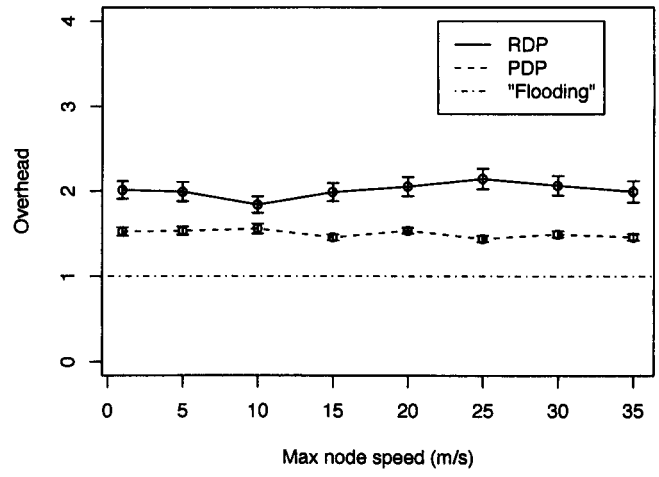

Figure 3.3: Overhead vs. Maximum node speed for PDP and RDP. Wireless range = 250m, $k = 45$.

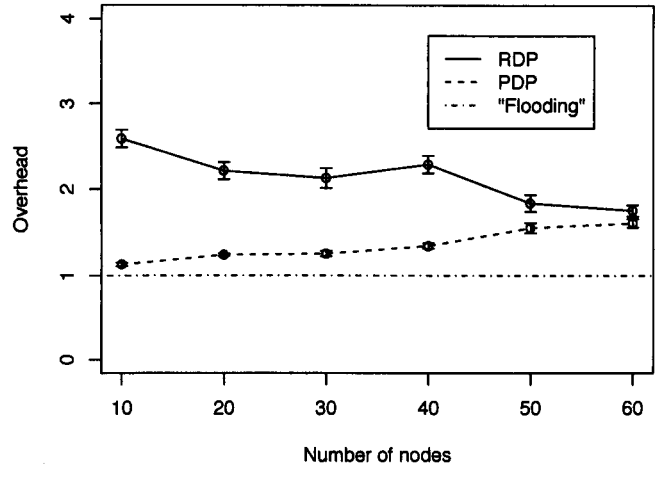

Figure 3.4: Overhead vs. Number of nodes for PDP and RDP. Wireless range = 250m, $k = (n - 5)$.

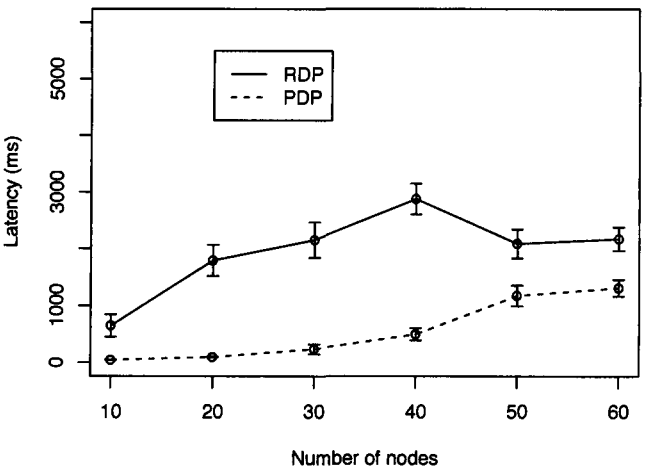

Figure 3.5: Latency vs. Number of nodes for PDP and RDP. Wireless range = 250m, $k = (n - 5)$

Summary

The findings in this subsection can be summarised as follows:

- Both RDP and PDP are able to provide their guarantees in a very wide range of network scenarios.
- The proactive dissemination strategy is better at high densities, the reactive at low.
- Node speeds are largely immaterial to both protocols. This is a good result in ad-hoc networks.
- Increasing the number of nodes slightly benefits the reactive approach and slightly disadvantages the proactive.

In the next section we will use some of these findings to develop an optimised reliable dissemination protocol.

3.3 An optimised reliable dissemination protocol

The two protocols presented in the previous section, PDP and RDP, were designed to show how purely proactive and purely reactive dissemination strategies performed. To this end, only a very simple optimisation, namely choosing a random time between 0 and β to perform the evaluation of neighbourhood/transmission of m, was added.

This section presents an optimised reliable dissemination protocol which takes into account the lessons learned from PDP and RDP, and attempts to optimise for the case when the network does not behave in an adversarial manner.

3.3.1 Pulling large payloads

As we have seen in the previous section, a proactive approach is better at high densities, while a reactive approach has the upper hand at low densities. An ideal solution is a protocol which combines the best of these two approaches while retaining the simplicity of PDP, including avoiding the need for neighbourhood knowledge. A *Push-Pull, PP,* optimisation of PDP goes some way towards providing this.

The PP optimisation requires responsible nodes to periodically transmit (push) $K(m)$, at least once every β seconds where $B \geq 3(\beta + \delta)$, while nodes which have not received m must request (pull) m from responsible nodes upon receiving a $K(m)$ for a message they have not yet received. The protocol steps are as follows (note that for clarity, new functionality not found in PDP *has been italicised*):

1. **Starting the dissemination:** The initiator initialises $K(m)$ to be all 0's, and sets its own bit to 1. It generates a unique id, $m.id$, for the message and sets the $m.k$ to the minimum number of nodes it wants to guarantee receives m (note that $k \leq (n - f)$). It then becomes *responsible* for m.

2. **Being responsible:** *Every message a node is responsible for is retained and a K_Pkt, containing m.id and $K(m)$, is transmitted periodically with intervals randomly chosen between 0 and β seconds. After each transmission of a K_Pkt, a new random interval is chosen.*

3. **Receiving an m:** Upon receiving a message for the first time, a node initialises $K(m) = m.K$ and sets its own bit to 1. It then becomes responsible for m. Every time it receives a message it is responsible for, it merges the received $m.K$ with $K(m)$.

4. ***Receiving a K_Pkt:*** *If a node receives a K_Pkt(m.id) for a message it has not received, it transmits a request packet, Req_Pkt, containing m.id.*

5. ***Receiving a Req_Pkt:*** *A responsible node receiving a Req_Pkt(m.id) transmits m. A node which has not received m ignores the Req_Pkt.*

6. **Realising m:** Every time a node updates $K(m)$, either because it adds its own signature to it, or because it merges $K(m)$ with an incoming $m.K$, it checks if $|K(m)| \geq m.k$. If it is, the node realises m. Realising m involves canceling the periodic transmission of *K_Pkt* and deleting m from stable storage. Every time a node receives a packet relating to a message which has been

realised, it transmits a small realisation packet, *R_Pkt*, containing the message's unique id. A node which receives an *R_Pkt(m.id)* takes the same action as if it had detected $|K(m)| \geq m.k$ itself. *A node which has not received m and which receives a R_Pkt(m.id) transmits a Req_Pkt(m.id), and only realises m if and when it receives it.*

The PP-optimised protocol's correctness follows from that of PDP, with the observation that given $3(\beta + \delta) \leq B$, the PP-optimised protocol uses the direct connectivity between an unrealised, but responsible node N_i and some other node N_j effectively; either (i) N_j receives m, if it has not already received it, or (ii) or N_i and N_j will exchange their $K(m)$s if both are unrealised.

If N_j has not already received m (case (i) above), then N_j is guaranteed to receive a *K_Pkt(m.id)* during the first β seconds, as N_i is executing step 2. N_j then and has the time to send (step 4), and N_i will receive, its *Req_Pkt* during the middle β seconds. Finally N_i will respond by transmitting m (step 5), which N_j will receive m during the final β seconds.

If both N_i and N_j are responsible, but have not realised m (case (ii) above), then both will exchange their *K_Pkts* during the first β seconds.

Figure 3.6 shows how the PP-optimised protocol performs in terms of overhead compared to PDP and RDP as density is varied (this is the same setting as in figure 3.1).

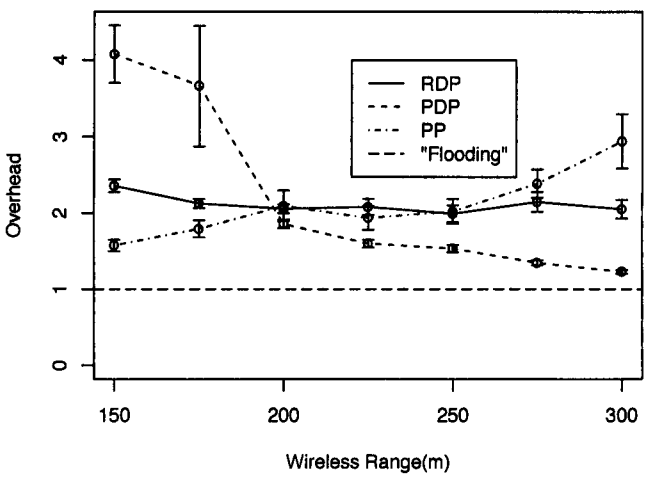

Figure 3.6: Overhead vs Wireless range for PDP and RDP and the PP-optimised protocol. Max speed = 5m/s, $k = 45$.

At low densities, PP performs better than both PDP and RDP.

PP outperforms PDP at low densities because it puts less bytes on the network than the simple PDP approach; the use of *Req_Pkts* is similar in spirit to utilising neighbourhood knowledge, and therefore PP is able to be more selective about when it transmits m. This is beneficial at low

densities as we saw in section 3.2.4.

PP is also able to outperform RDP at low densities as it causes fewer copies of m be transmitted than RDP. The reason is that RDP forces a responsible node to transmit m to any node in its neighbourhood which it does not know to have received m. This is because neighbourhood knowledge, as it is assumed to incur no extra overhead and thus cannot be protocol specific, does not include information about what messages any given neighbour has already received. This quite frequently results in RDP transmitting copies of m to nodes which have already received it (as discussed in section 3.2.4). With PP, m is only transmitted if a node actually requests it, and is thus less likely to be redundant.

However, as nodes' wireless ranges exceeds 200m, the PP-optimised protocol performs worse than PDP, and at above 250m is worse than both. The reason is that at higher densities, the additional overhead and delay associated with having to request m is detrimental to performance; it is better simply to push m (as PDP does) at such densities.

A further optimisation

Based on this observation we present a further optimisation, called *Push-Pull with initial push of data payloads*, or *PP++*. PP++ is the same as PP, but in addition each node is required to retransmit m once the first time they receive it. That is, step 3 becomes:

3. **Receiving a message:** Upon receiving a message for the first time, a node initialises $K(m) = m.K$ and sets its own bit to 1. *It then immediately retransmits* m, and becomes responsible for m. Every time it receives a message it is responsible for, it merges the received $m.K$ with $K(m)$.

The rationale is to attempt to get m to as many nodes as possible, as quickly as possible without having to send *Req_Pkts*. This captures some of the beneficial characteristics PDP has at both high and low densities. At high densities we have seen that quickly pushing m is the better approach (see for example figure 3.6), but even at low densities making nodes which happen to be directly connected to the initiators responsible, is beneficial.

Figure 3.7 shows how the PP++ protocol performs in terms of overhead compared to PDP and RDP as density is varied. The PP++ protocol is now quite efficient. It is able to provide its guaranteed delivery with transmission overheads of only about 1.5 times that associated with an unreliable, idealised flooding protocol. Further, it consistently outperforms RDP and is only beaten by PDP as wireless ranges reach 300m. The reason it is outperformed at this density is that any thing other than a single transmission from any one node is redundant when the density is that high.

Figure 3.8 indicates that both the PP and PP++ optimisations have not increased latency substantially. In fact it suggests (at high densities, at low densities there is no identifiable trend)

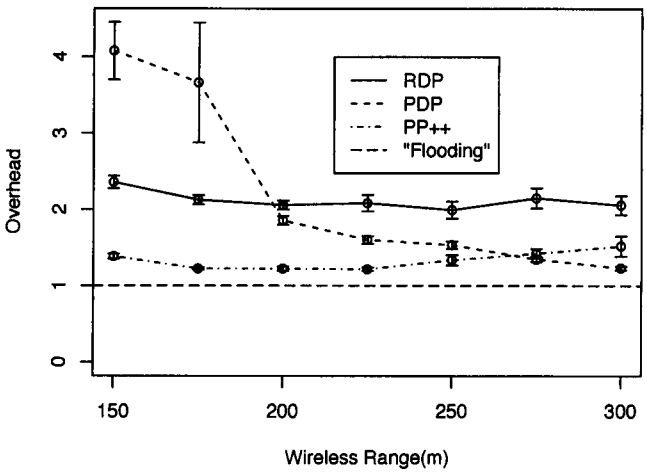

Figure 3.7: Overhead vs Wireless range for PDP and RDP and Push/Pull with initial push of data payloads. Max speed = 5m/s, $k = 45$.

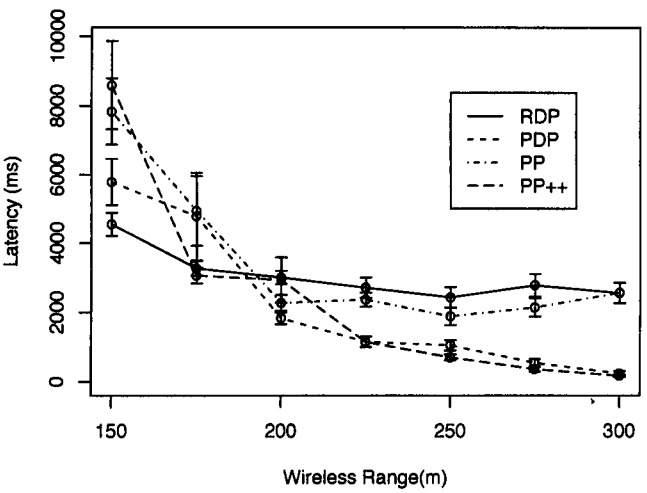

Figure 3.8: Latency vs Wireless range for PDP, RDP, PP and PP++. Max speed = 5m/s, $k = 45$.

that PP has similar latency to RDP and PP++ has a similar latency to PDP. This is another benefit of the initial push of data by PP++; latency is lower.

Finally we note that the push-pull optimisations' (PP/PP++) efficiency is dependent upon the size of the payload in each message, particularly at lower densities. As a rule, there was no, or a deteriorating benefit to using the push-pull optimisations if the payload was less than about 100 bytes. This is natural as when payloads get smaller, the difference in size between the full m and only the K_Pkt is reduced.

3.3.2 Suppressing equivalent transmissions

Mobile ad-hoc networks are broadcast by nature, as most communication happens by means of omnidirectional antennae. The implication of this is that a node will receive messages transmitted in its neighbourhood, even if it is not the destination node. This feature can be taken advantage of to reduce the overhead of our protocol.

Ni et al.[NTC99] were the first to use this feature to reduce the overhead of unreliable flooding. Specifically, they proposed a simple counter-based scheme where a node, upon receiving a message for the first time, picks a short random interval called *random assessment delay, RAD.* During the RAD (typically between 0-100ms), a node counts the number of transmissions of the same message it receives, and if this count is greater than some threshold, the message is not retransmitted by the node and simply deleted.

The rationale behind this approach is that the additional physical area a node's transmission of m can expect to cover drops dramatically as the number of redundant transmissions of m received increases. This is shown graphically in figure 3.9.

The figure shows, for example, that if a node has received 3 transmissions of the same message, it can expect that 91% of the area covered by its wireless range has already been covered by another transmission. In such cases it is probably not prudent for the node to retransmit.

These results are based on a number of simplifying assumptions; specifically that a node's neighbourhood is perfectly spherical and that nodes do not move. None the less, this simple counter-based scheme performs well when compared against other, more complex optimisations. Examples of the more complex schemes include a distance-based scheme, where the physical distance to the transmitting node is approximated and a node only retransmits if it is more than a threshold distance from the transmitter, and a cluster-based scheme, where nodes are grouped into clusters and only the cluster-head retransmits [NTC99].

However, the approach taken by Ni et al. must be adapted to work with our protocol, as Ni et al. only considers unreliable, best-effort broadcast. The differences arise because in Ni et al.'s approach, any two messages with the same message identifier ($m.id$) are considered equivalent and because each node is only required to make a decision once about whether to retransmit any given

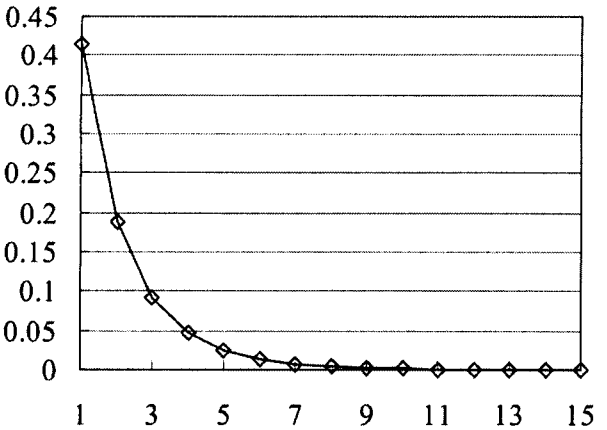

Figure 3.9: "Expected additional coverage" vs. number of transmissions received during the random assessment delay. From [NTC99]. Note that 1.0 = 100% of the area covered by a node's wireless range.

message. This is not the case for our protocol.

The reason two messages with the same $m.id$ may not be equivalent is because there are two things that are important in our protocol; the actual data payload, and the knowledge about which nodes have received the message, $K(m)$, contained in the message header. The data payload is the same in any transmission of m. However, this is not the case for $K(m)$s, as $K(m)$s with different signatures are not equivalent. For example, it would be undesirable for a message with only 1 signature to suppress the transmission of a message with lots of signatures.

In addition, in the best effort case there is no need to consider how long ago a transmission happened. This is because RADs are short, and a node will never transmit a message more than once; all transmission received during the RAD is counted, and if some threshold is reached the message is simply discarded. In a reliable protocol this is not possible as a node may have to transmit the message more than once. The question then becomes; how recent should a transmission be for it to be counted?

The first of these issues has been addressed by observing that in general, it is undesirable for a message to suppress the transmission of another message if the former does not contain all the signatures in the latter, as this can hinder the protocol's ability to successfully terminate. We therefore define an "equivalent" $K(m)$ to be one where the received $m.K$ contains all signatures in the local $K(m)$; e.g. $K(m) - m.K = \emptyset$. We now keep two counts; one for equivalent $K(m)$s received, *eqvKmCount*, and one for ms received, *eqvDataCount*. The *suppression threshold*, the number of equivalent transmissions after which a transmission is suppressed, is denoted by α.

The suppression optimisation can be added to the $PP++$ optimisation described in section 3.3.1

by performing the following additional steps at the events described:

Receiving an m : The $eqvDataCount(m.id)$ is incremented by 1.

Receiving a $m.K$ (Note that this can be both in a m and a K_pkt): The received $m.K$ and local $K(m)$ is merged. If the number of signatures in $K(m)$ is increased as a result of this, $eqvKmCount(m.id)$ is reset to 0. If $K(m) - m.K = \emptyset$, $eqvKmCount(m.id)$ is incremented by one.

Transmitting m : If $eqvDataCount(m.id) > \alpha$, the transmission of m is suppressed. Then $eqvDataCount(m.id)$ is reset to 0.

Transmitting K_Pkt : If $eqvKmCount(m.id) > \alpha$, the transmission of K_Pkt is suppressed. Then $eqvKmCount(m.id)$ is reset to 0.

Figures 3.10 and 3.11 shows how varying the value of α impacts overhead and latency.

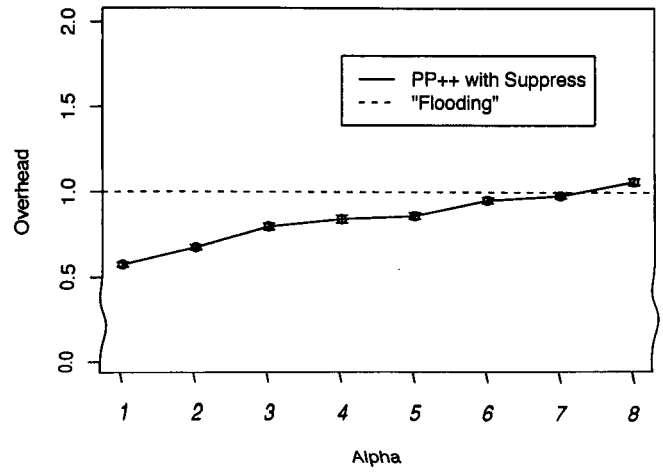

Figure 3.10: Overhead vs various values of α with Push/Pull with initial push of data payloads *(PP++). Wireless range = 250m and max speed = 5m/s*

The figures show a clear trade-off between latency and overhead, with overhead increasing and latency decreasing as α is increased. For example, the overhead with $\alpha = 1$, is only half that when $\alpha = 8$, but the latency is increased by around 1500ms in doing so. As it is difficult to predict what the end user will find more important, latency or overheads, *it is probably a good idea to allow the* application developer to tune the values of α. It is therefore left as a configurable parameter.

Throughout the rest of this dissertation we are *mostly concerned with reducing overhead, and* thus use the dissemination protocol with $\alpha = 1$; in this case the overhead associated with the reliable

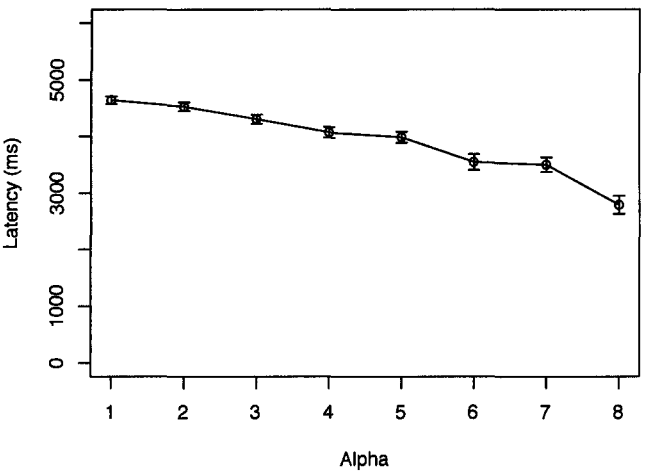

Figure 3.11: Latency vs various values of α with Push/Pull with initial push of data payloads (PP++). Wireless range = 250m and max speed = 5m/s

dissemination is less than an idealised flooding protocol, as the suppress optimisation means many nodes never need to transmit m. Such low overheads make reliable dissemination a practical option in ad-hoc networks.

3.3.3 Performance study

This section briefly studies the performance of the "complete" protocol; that is, the reliable dissemination protocol which includes the push-pull with initial push of data (PP++)(section 3.3.1) and the suppress on equivalent (section 3.3.2) optimisations.

Figures 3.12 and 3.13 shows how the protocol performs as density is varied. The overhead associated with the reliable dissemination protocol is now less than an unreliable, idealised flooding protocol for all wireless ranges greater than 100m. This improvement is achieved while still maintaining quite a reasonable latency, as seen in figure 3.13

Figure 3.14 shows that the complete protocol maintains one of the nice properties of PDP and RDP, namely that node speed does not impact overheads. This is not surprising as none of the optimisations involve routing structures. We observe that, as with PDP and RDP, this also holds at higher and lower densities than shown and also applies to latency.

Finally, the results in figures 3.15 suggests that the optimised protocol has the same scaling properties as RDP. This is because the optimised protocol makes use of the "virtualised" neighbourhood knowledge afforded by the push-pull optimisation to avoid making redundant transmissions. However, this comes at a cost of slightly increased latency, as can be seen in 3.16.

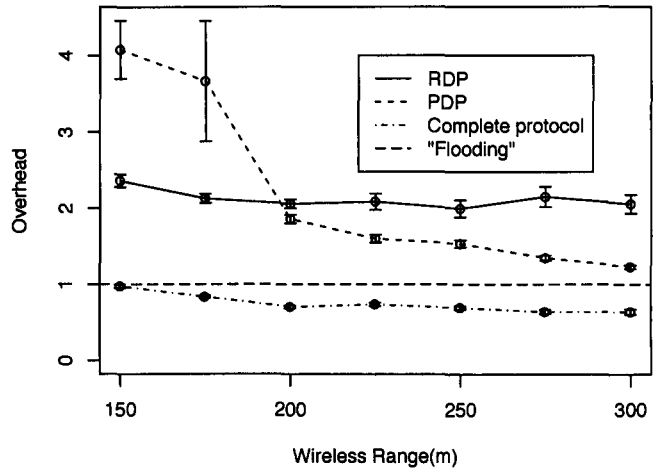

Figure 3.12: Overhead vs node density. Max speed = 5m/s. Number of nodes = 50, k = 45.

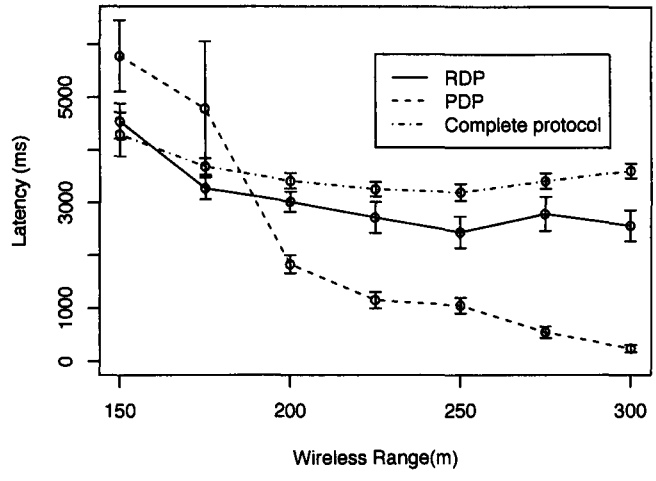

Figure 3.13: Latency vs Density. Max speed = 5m/s. Number of nodes = 50, k = 45.

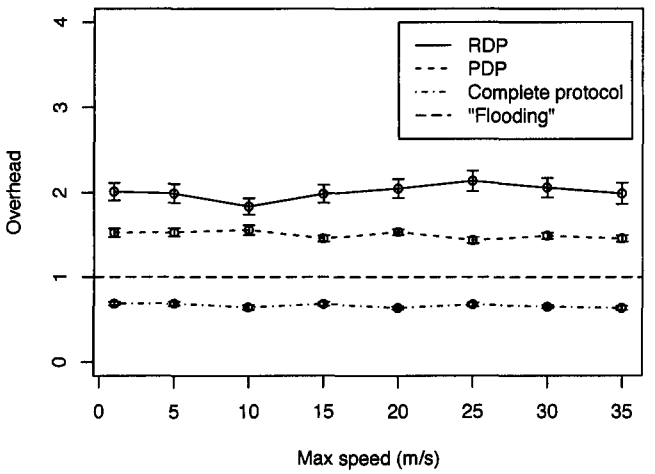

Figure 3.14: Overhead vs Node speed. Wireless range = 250m. Number of nodes = 50, k = 45.

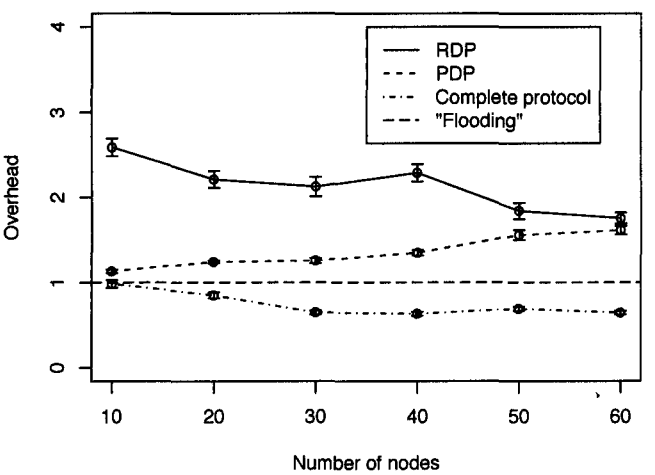

Figure 3.15: Overhead vs Number of nodes. Wireless range = 250m. Max speed = 5m/s.

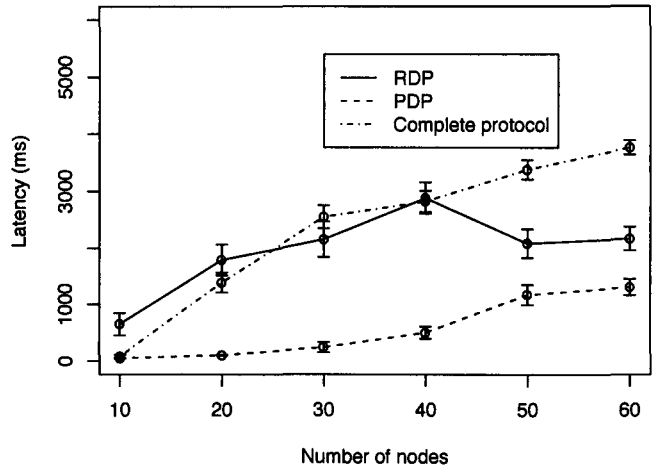

Figure 3.16: Latency vs Number of nodes. Wireless range = 250m. Max speed = 5m/s.

3.4 Related work

Few, if any, *fault-tolerant* reliable dissemination protocols (be it broadcast or multicast) have been proposed specifically for mobile ad-hoc networks. However, a number of protocols which are not fault-tolerant, but none the less reliable, have been proposed. Representative examples include Reliable Broadcast (RB)[PR97], Adaptive Reliable Broadcast (ARB)[GS99] and Reliable Adaptive Lightweight Multicast(RALM)[TOL02]. This section briefly describes these three and discusses the differences and similarities between them and the protocols presented in this chapter.

3.4.1 Reliable Broadcast (RB) protocol

The RB[PR97] protocol was the first reliable broadcast protocol designed explicitly for mobile ad-hoc networks. The authors argue that for low mobility ad-hoc networks, well known spanning tree algorithms could be used, as the network is essentially static. At the other extreme, where the mobility of the nodes is very high, the authors claim that: "there is no alternative to flooding". The RB protocol is designed for when the mobility is in between these two extremes, and also provides the ability to switch to flooding once the rate of topology change in the network is deemed too high.

RB assumes the existence of a clustering algorithm (e.g. [LG97]), and works by having each node wishing to reliably broadcast a message do a blocking send to the cluster head of the cluster it is currently in. The cluster head then sends the message to all the nodes in its cluster, and waits for acknowledgments from each of the cluster members. Any nodes acting as gateway will then forward the message onto the gateway or cluster head of the cluster to which it is the gateway, and delay the

acknowledgment of the message received from the original cluster-head until the message has been successfully diffused in the nearby cluster. This in turn could involve a recursive wait, as the this cluster might be connected to another cluster and so on.

The protocol essentially constructs a routing tree, using the underlying clustering algorithm, over which messages and acknowledgments are sent. In cases when the underlying clustering algorithm is unable to cope with increased node mobility, the protocol will switch to flooding acknowledgments back to the cluster head of the cluster containing the originating node. This is required as acknowledgments from the destination nodes' cluster heads is essential for the protocol to ensure that the message has been received by all nodes. This is an example of the type of centralised mechanism which is necessary to detect that a sufficient number of nodes have received a message when a distributed mechanism (such as that introduced in section 3.2.2) is not used.

A further observation is that even when RB reverts to flooding acknowledgements and messages, the protocol is not guaranteed to succeed if only the NLR is guaranteed to hold; the network acting in an adversarial manner can easily disallow any contemporaneous paths (which a flooding protocol relies on) between the originating node's cluster head and other nodes in the network. What is required (as dictated by theorem 2.1) is that all nodes must potentially *retain* each message. Flooding does not require nodes to do this.

3.4.2 Adaptive Reliable Broadcast (ARB) protocol

The ARB protocol [GS99] is a reliable multicast protocol that adapts to the rate of topology change in the network. The protocol works by constructing a core based shared multicast tree. Whenever a sender wants to multicast a message to members of the group, it sends a multicast message to the core node of the given group, which initiates the dissemination of the message down the multicast tree. The acknowledgments from individual nodes travel in the opposite direction up the tree to the core node. In order to reduce bandwidth consumption, the protocol uses acknowledgment aggregation. A message is said to be "stabilized" when the core node has received acknowledgments from all the group nodes. This is another good example of the type of "centralised realisation mechanism" which is required when there is no distributed mechanism available.

In case of fragmentation due to node movement, the concept of a *forwarding region* is introduced which is used to "glue together" the fragmented multicast tree. This gluing involves flooding of the forwarding region (essentially consisting of nodes which witness the topology change due to node mobility) in order to attempt to repair the routing tree. In addition, there is a notion of nodes requesting missing messages from the core node when they (re)join the multicast tree.

The (implicit) assumption made is that eventually the multicast tree will be created in such a manner as to include all destination nodes, and further that this tree remains intact (or can be repaired) until the nodes can acknowledge the reception of a message; the protocol does not

even include a "revert to flooding" strategy as in RB. From this we can conclude that the protocol would not cope with an ad-hoc network which acted in an adversarial manner, unlike the protocols presented in this chapter.

3.4.3 Reliable Adaptive Lightweight Multicast (RALM)

RALM [TOL02] is a reliable protocol which uses a TCP-like error and congestion control. The protocol assumes that the multicast receivers are known to the source, either through receiver discovery or by advance knowledge.

When a source has multicast data to send, it picks a receiver from the Receiver List, containing the receivers. It then starts to send messages to the multicast group, with the chosen receiver (called the feedback receiver) included in the packet header instructing it to unicast a reply containing an ACK or a NACK and a sequence number. All other receivers in the multicast group simply process the message without acknowledging. If the feedback receiver determines that packets are missing, it will request the missing packets one at a time from the source.

The philosophy behind transmitting each lost packet one at a time is to slow down the source when congestion is detected. Both new and retransmitted packets are multicast, which implies that most of the multicast group members should receive the data packets received by every feedback receiver. Once the feedback receiver has received all the packets, it unicasts an ACK back to the source, upon which the source picks a new receiver from the Receiver List and repeats the process until the list is empty.

The central new contribution of RALM is the introduction of a TCP-like window-based congestion control mechanism used to reduce overhead and increase efficiency. The protocol therefore does not attempt to guarantee delivery when the network acts as an adversary. The assumption is that contemporaneous paths between the source and the receivers in the Receiver List can be established. Such an assumption is not made by the protocols in this dissertation.

3.4.4 Discussion

In general, these protocols make stronger assumptions about the underlying network property than that made by the protocols introduced in this chapter. For example, RB makes the following two assumptions about network behaviour:

1. If there are pending messages for a node, p_i, then p_i receives at least one of these messages, and it succeeds in notifying the reception before the topology changes again.

2. A host remains cluster head for the time necessary to guarantee that the exchange of status information with the other cluster heads is successfully completed and, if there are partially diffused messages, that at least one of them is known amongst the cluster heads.

Clearly these assumptions are much stronger than the network liveness requirement, NLR, (see section 2.2) assumed by the protocols presented in this chapter. This explains why RB, ARB and RALM do not include the possibility that every node becomes responsible for a given message as dictated by theorem 2.1; by assuming more about the behaviour of the underlying network, the protocol has to do less work. However, this does mean that the protocols will not work in as wide a range of network scenarios as the protocol presented in this chapter. In addition, as argued previously, the more complex the assumptions made by the reliable protocol, the harder it is for the end user to ensure that these assumptions are met.

A further observation is that the protocols also make use of routing structures, whether these are clusters, multicast trees or unicast routes, and all require the initiator (or the core node in the case of ARB) to be able to detect when the dissemination has been successful. However, even in relatively benign network conditions, creating and maintaining routing structures can become expensive in terms of overhead. For example, the lower bound on message passing overhead associated with creating a minimally connected dominating set (MCDS), the most efficient form of multicast structure, is O(n log n)[WAF04]. Further, when the density is low, it may not be possible to create a route between all nodes and some form of store-and-forward approach is necessary.

3.5 Conclusion and summary

This chapter has studied the problem of providing reliable dissemination in mobile ad-hoc networks where transient partitions and undetectable node crashes can occur. The study has been guided by the foundational results presented in chapter 2 which restrict the number of nodes which can be guaranteed to receive a message, and show how protocols which cannot rely on routing structures must include the possibility of all nodes actively participating in the dissemination of a message.

A novel, distributed method for detecting that a protocol can terminate has been presented. The method removes the need for separate acknowledgements to be sent back to the originating node through the use of "node signatures" appended to message headers. This removes the reliance on routing structures and a single node to detect that a protocol can successfully terminate found in other protocols. We have also argued that these signatures can be compactly represented when nodes are assigned unique identifiers upon joining a collaborative group.

Two core dissemination strategies, a proactive and a reactive, was combined with the above method to derive two reliable dissemination protocols. The proactive protocol assumes no neighbourhood knowledge was available and thus, as dictated by theorem 2.2, periodically transmits every message. The reactive protocol attempts to make the most out of the neighbourhood knowledge, which is assumed to be available to it, by transmitting only when necessary.

These two basic protocols were studied through simulation and we concluded that both protocols

are able to provide their delivery guarantees under a very wide range of network conditions, but that the reactive strategy outperforms the proactive strategy at low densities, while at high densities the roles are reversed.

Finally, we have also presented an optimised reliable dissemination protocol which, in fairly typical network scenarios, is able to provide its strong delivery guarantees with *less* overheads associated with it than what would be expected from an *unreliable* flooding protocol. It is able to do this by means of two optimisations: "push-pull" and "suppress equivalent". The "push-pull" optimisation was arrived at by combining the best features of the proactive and reactive dissemination strategies, while the "suppress equivalent" optimisation is a modified version of the unreliable counter-based flooding protocol of Ni et al.[NTC99].

In summary, the results in this chapter suggests that fault-tolerant reliable protocols are feasible in even highly mobile, transiently disconnected and crash-prone ad-hoc networks. In the next section we will use the reliable dissemination protocols presented here as a basis for solving the more difficult consensus problem in a manner suitable for mobile ad-hoc networks.

Chapter 4

Consensus in ad-hoc networks with crash failures

4.1 Introduction

Services and applications deployed on mobile ad-hoc networks are by their very nature prone to crash failures and disconnectedness. If the service being provided is sufficiently critical, it is desirable to replicate the state of this service onto several physically distinct nodes to increase fault tolerance and improve availability. The replicated state must be kept consistent in order to be functionally equivalent to a non-replicated service.

Two main issues arise when a replicated service state must be kept consistent; (i) each replica of the service must receive the same updates and (ii) the replicas must apply the updates in the same order. The first issue can be solved by using some form of reliable dissemination primitive of the sort discussed in the previous chapter. The second issue is more difficult as it involves the replica services agreeing on a common value (the order in which to apply the updates). This problem, known as the *atomic broadcast* problem, is a typical *agreement problem* from distributed systems theory. Other agreement problems include leader election, atomic commitment, mutual exclusion, and group membership

The *consensus* problem is a general form of agreement in distributed systems[TS92]. A number of agreement problems, such as atomic broadcast, have been shown to be equivalent to consensus[CT96]. This implies that any such problem can be implemented provided a consensus solution is available, and any difficulty inherent in solving the consensus problem also inevitably applies to any of the other agreement problems. For these reasons, a rigorous solution to consensus is of critical importance if fault-tolerant and available services are to be achieved in ad-hoc networks.

However, existing wired network solutions to the consensus problem are not well suited to the types of mobile ad-hoc networks considered in this dissertation. For example, typical wired network solutions assume that best-effort unicast or multicast are readily available and low overhead

operations, and thus use them extensively. As shown by theorem 2.1, the routing structures which underpin such protocols are not guaranteed to exist in mobile ad-hoc networks, and numerous studies (e.g. [LSH00]) show that, even in relatively benign network conditions, providing low overhead best-effort unicast or multicast is by no means trivial. Further, the underlying relying dissemination protocols which some existing solutions are built on attempt to deliver messages to all nodes. As shown by theorem 2.3, this is impossible if bandwidth and storage subsidence are desirable.

In this chapter we will survey the vast amount of work done on consensus for wired networks, including a fundamental impossibility result and two main ways of circumventing it; *randomisation* and *failure-detectors*. We will also argue why a randomised consensus solution seems the most suitable for the types of mobile ad-hoc networks considered in this dissertation.

We will then derive a new randomised consensus solution based on a wired network protocol by optimizing the latter in 3 different ways. The resulting protocol is the first randomised consensus solution developed specifically for mobile ad-hoc networks. The extensive simulations performed suggests that the protocol reduces the number of execution rounds required to reach consensus by several orders of magnitude, and further reduces the transmission overhead per round by 1-2 orders of magnitude. This implies that running consensus in mobile ad-hoc networks is a practical proposition.

A further contribution is the provision of a proof of correctness. Crucially this proof removes the need to guarantee delivery to *all* correct nodes found in the original protocol. Such a dependence runs contrary to the result in theorem 2.3 which shows that this cannot be guaranteed.

4.2 The consensus problem

This section formally defines the consensus problem, followed by a fundamental impossibility result applicable in the types of mobile ad-hoc networks considered in this dissertation.

4.2.1 Problem definition

As in previous chapters, there are a set of n nodes. A node is either *correct* or *faulty*. A correct node never crashes, while a faulty node can crash at any time. A node crash is undetectable and nodes do not recover from a crash. The number of faulty nodes is bounded to within some known value f. Further detail about the system model is given in section 2.

In the consensus problem, any node N_i can propose a value v_i, and all correct nodes have to decide on some common value v which is equal to one of the initially proposed values[CT96]. Formally the consensus problem is defined in terms of two primitives: `propose` and `decide`. When a node invokes `propose`(v_i) where v_i is its initial proposal to the consensus protocol, we say that N_i "proposes" v_i. Further, when N_i invokes `decide()` and gets v as a result, we say that N_i "decides" v. The

consensus problem is defined as follows:

Validity If a node decides v then v was proposed by some node.

Agreement No two correct nodes decide differently.

Termination Every correct node eventually decides.

The termination property defines the *liveness* property associated with consensus. The agreement and validity properties are *safety* properties.

The agreement property above allows correct and faulty nodes to decide different values. This can be undesirable in certain circumstances as it can allow faulty processes to propagate inconsistent values before crashing. For this reason, in the *uniform consensus* problem, agreement is defined as:

Uniform agreement No two nodes (faulty or not) decide differently.

All the consensus solutions described in this chapter solve the uniform consensus problem.

As shown by theorem 2.1, reliably disseminating a message to *all* correct nodes is impossible when crash failures are undetectable, if the storage subsidence property (SSP) must also be met (see section 2.3.1). For this reason the termination property in the traditional consensus definition is impossible to ensure if the SSP is desirable. As ensuring storage subsidence is clearly desirable, the termination property is altered as follows:

Termination At least one correct node eventually decides.

This alteration does not weaken the safety properties of the consensus solution in any way.

A fundamental property of a distributed system is whether there are timing bounds on communication delays or processing speeds. In a *synchronous* system there is a known, finite bound on the processing speeds and communication delay between any two nodes. In an *asynchronous* system there are no such bounds.

An ad-hoc network is clearly an asynchronous system due to the possibility of transient network partitions and highly variable network latencies. Further, a solution to the consensus problem for an asynchronous system model is also applicable in the easier to solve synchronous model. The reverse is not true. For these reasons this chapter will only consider consensus in asynchronous systems.

4.2.2 A fundamental impossibility result

Developing a consensus solution on the assumption that ad-hoc networks are asynchronous is clearly preferable to making assumptions about network synchrony, as these assumptions may not hold and thus invalidate the guarantees of the consensus protocol. However, in 1985 Fisher, Lynch and Paterson showed that it is impossible to design a deterministic consensus protocol in an asynchronous

distributed system that is guaranteed to terminate in bounded time subject to the possibility of even a single crash failure [FLP85]. The proof holds even if the communication network is completely reliable. This well known result is commonly known as the *FLP impossibility result.*

The intuition behind the result is that in an asynchronous network it is impossible to distinguish a crashed node from a very slow one. As a result of this, any deterministic consensus protocol contains non-terminating executions; that is, possible executions of the consensus protocol where no decision is ever made. This violates the termination property described in the previous section.

However, as consensus is such a fundamental problem, it still needs solving. In the next section we will briefly survey the best known approaches to solving consensus in wired networks.

4.3 Known approaches to solving consensus

Two main ways of circumventing the FLP impossibility result have been proposed; (i) imposing some form of timing constraint on the network, and (ii) relaxing the deterministic termination property.

The first of these, imposing further timing constraints, includes the introduction of the *timed asynchronous* model[CF99], the *partially synchronous* model[DLS88] and the asynchronous systems with *unreliable failure detectors* model[CT96]. The general gist of these approaches is to assume that the network goes through stable periods during which progress towards consensus can be made. Of these the unreliable failure detector approach is the most general and we will therefore discuss it further in the next section.

The second approach, relaxing the deterministic termination property, circumvents the FLP impossibility result by weakening the termination property to only require termination with probability 1 (note that this does not affect the safety properties). This change means non-terminating executions of consensus still exist, but occur with probability 0. Randomisation will be further discussed in section 4.3.2, and is the approach taken in this dissertation.

4.3.1 Unreliable failure detectors

The unreliable failure detector abstraction was first introduced by Chandra and Toueg[CT96]. Failure detectors provide *approximate* views of process crashes. The ability to detect node crashes, even unreliably, allows consensus to be solved deterministically by electing a coordinator to make decisions, while at the same time not waiting forever for a crashed coordinator to respond.

Failure detectors are modules attached to each node. The failure detector module provides a list of *suspected* nodes. Naturally, as the network is still asynchronous, the failure detector cannot be mistake-free. That is, a failure detector may suspect a correct node, or fail to suspect a crashed one. However, to be useful the failure detector has to provide some correct information. The information provided is defined in terms of *completeness* and *accuracy.* Accuracy restricts the

amount of erroneous suspicions that a failure detector may make, while completeness defines to what degree crashed nodes must be suspected.

Chandra and Toueg defined a number of failure detectors based on their completeness and accuracy properties. They also showed that the eventual strong failure detector, $\diamond S$, was the weakest required for consensus to be solved[CT96][1]. $\diamond S$ is defined by the following properties:

Strong Completeness : Eventually every crashed node is permanently suspected by every correct node

Eventual Weak Accuracy : There is a time after which some correct node is never suspected by any correct node.

Intuitively these two properties allow the correct nodes to elect a coordinator which has not crashed (by strong completeness) and stay with that coordinator until consensus is reached (by eventual weak accuracy). The weak accuracy property may seem highly unrealistic; "some correct node is *never* suspected", but in fact "never" in this case actually only means until a decision is made, which in a real system may be a tolerable assumption.

Failure detectors are typically implemented in one of two ways; either using periodic "heartbeat" messages to all nodes in the system, or through gossiping about perceived view of suspected nodes[vMH98].

Of these, sending periodic "heartbeat" messages to all nodes is the simplest and most widely used, though also the approach which scales worst and incurs the greatest overhead ($O(n^2)$ heartbeat messages). Gossip based failure detection goes some way to alleviate this. In the basic gossiping protocol, each failure detector module picks another failure detection module randomly (without concern for the network topology) and sends it its list of known nodes with their heartbeat counters after incrementing its own heartbeat counter. The receiving failure detector will merge its local list with the received list and adopts the maximum heartbeat counter for each node. Occasionally each node broadcasts its list to recover from network partitions. If a heartbeat counter for a node N_i maintained at a failure detector at another node N_j has not increased after some timeout, node N_j suspects node N_i to have crashed.

A number of consensus protocols have been developed using failure detectors. A failure detector based consensus protocol also lies at the heart of group communication toolkits such as Horus[RBM96], Ensemble[RBH98].

Most of these protocols (e.g. [CT96][Lam98][MR99]) use the *rotating coordinator* paradigm to solve consensus. The rotating coordinator paradigm works in consecutive, asynchronous rounds. Each round is coordinated by a single, *coordinator*, node. The coordinator of round r, node N_c, is

[1]Not strictly true; the eventual weak failure detector, $\diamond W$, is the weakest, but they also showed how to transform the weak completeness property into a strong one, so we focus on $\diamond S$ here

predetermined. One way to determine this is by choosing the coordinator id, c, as $c = (r \bmod n) + 1$. Each node, N_i keeps an estimate of the decision value, est_i to champion. Initially est_i is set to node N_i's input to the consensus protocol. Nodes update their estimate during the execution of the consensus protocol as follows:

In a round r, all nodes send their estimate to the current coordinator, N_c. N_c then proposes one of these estimates and sends it to all nodes. If a node does not suspect N_c, and N_c does not crash, it eventually receives this proposal and champions it as its estimate. The proposal is decided upon when a majority of nodes champion the proposal. If a node suspects a coordinator has crashed, it moves on to the next round with a new coordinator. Due to the strong completeness of $\diamond S$, eventually every correct node suspects a crashed coordinator.

4.3.2 Randomization

Using randomization was the first known approach, proposed by Ben-Or[Ben83], of circumventing the FLP impossibility result. This approach assumes there is a source of randomness in the system such that the choice of execution paths at certain points is not deterministic, but rather drawn from some probability distribution.

Adding randomness to the system in this way means no longer looking at a single worst case execution of a consensus protocol, as is done to prove the FLP impossibility result, but rather studying a probability distribution of bad executions. If the liveness property of the protocol is weakened to only requiring eventual termination, the FLP impossibility result no longer holds. By only requiring eventual termination, non-terminating executions continue to exist, but they exist with probability 0, so are in practice irrelevant.

Carrying forward the uniform agreement, and the slightly weaker termination property discussed in section 4.2.1, we get the following properties for randomised consensus in ad-hoc networks:

Validity If a node decides v then v was proposed by some node.

Uniform Agreement No two nodes (correct or faulty) decide differently

Termination With probability 1, at least one correct node eventually decides some value.

Again, the safety properties are unaffected by weakening the liveness property in this way.

There are two ways to add the required randomness to the system; (i) assume the environment the protocol operates in has sufficient level of randomness and (ii) add randomness to the protocol itself.

Assuming the environment provides a sufficient level of randomness was first proposed by Bracha and Toueg[BT85]. The approach requires that, at any given state, particular operations occur with

some probability. This differs from the definition of asynchronous systems where any operation is possible in each state. This has been called the *fair scheduling* assumption.

A fair scheduler assumes that:

1. In any round r, there is a positive constant $\epsilon > 0$, such that the probability that a node N_i receives a message from some other node N_j in round r is $> \epsilon$.

2. For any distinct nodes, N_i, N_j and N_k, in any round r, the event that N_k receives a message from N_i in round r, and the event that N_j receives a message from N_i in round r, are independent.

The protocol works in consecutive rounds by having each node broadcast (i) its preferred value, and (ii) the number of nodes it knows of which also have this as their preferred value (the vote for this value). In each round, every node receives $n - f$ messages. The nodes then change their preferred value according to which value has the most votes. This continues until, in a single phase, a node receives f or more messages of a single value each with at least $n/2$ votes, a which point it knows that the algorithm will decide on that value in the next two phases and terminates after broadcasting enough messages for the next two phases.

Given a fair scheduler, the probability that consensus is reached goes to 1 as the number of rounds go to infinity.

The caveat of relying on randomness in the environment is that these assumptions may not hold in certain real world scenarios. In addition, adding randomness to the protocol itself, the approach discussed next, is usually simple.

Adding randomness to the consensus protocol is typically done by associating a randomised oracle with each node. The oracle when invoked returns the result of a random coin-flip (or a random number from some probability distribution). The first protocol to do this was Ben-Or's binary randomised consensus protocol, which is presented in figure 4.1.

Ben-Or's protocol works in asynchronous consecutive rounds, consisting of two phases. Each node N_i maintains an estimate of the decision value, est_i initially set to the node's input to the consensus protocol (line 4). In the first phase of each round, r, each node reliably broadcasts a phase one message containing its current estimate to all nodes (line 7). A node then waits to receive $n - f$ phase one messages (it cannot wait for more as f nodes may have crashed). If more than $n/2$ of the received messages carry the same value, v, the node reliably broadcasts a phase two message containing v (line 11). Otherwise it reliably broadcasts a special value $\perp$. A $\perp$ value is by definition not a valid estimate/decision and indicates that no majority has been reached. The node then moves into phase two.

In phase two, each node waits to receive $n - f$ phase two messages (line 17). If a node, N_i receives *at least one* phase two message containing some value $v \neq \perp$, it sets its current estimate,

```
1   BenOrConsensus(v_i)
2   {
3           round = 1;
4           est_i = v_i;
5           while(true)
6           {
7                   send PHASEONE(round, est_i) to all nodes;
8                   wait to receive n − f PHASEONE(round, *) messages;
9                   if(more than n/2 carry the same value, est ≠ ⊥)
10                  {
11                          send PHASETWO(round, est) to all nodes;
12                  }
13                  else
14                  {
15                          send PHASETWO(round, ⊥) to all nodes;
16                  }
17                  wait to receive n − f PHASETWO(round, *) messages;
18                  if( received at least one PHASETWO(round, est ≠ ⊥) message)
19                  {
20                          est_i = est;
21                          if(received more than f PHASETWO(round, est ≠ ⊥) messages)
22                          {
23                              decide(est);
24                          }
25                  }
26                  else (received all PHASETWO(round, ⊥) messages)
27                  {
28                          est_i = coinFlip();
29                  }
30                  round++;
31          }
32  }
```

Figure 4.1: Ben-Or's binary randomised consensus protocol. Adapted from [Ben83]

$est_i = v$. If a node N_i receives *more than* f phase two messages with the same value $v \neq \perp$, then N_i `decide(v)` (line 23). If all received phase two messages contain only $\perp$, N_i queries the random oracle to adopt either a 1 or a 0 as its est_i (line 28). N_i then moves into the next round.

Ben-Or's guarantees agreement because:

- At most one value can receive a majority of votes in phase one, so for any phase two message with $v \neq \perp$, the value of v is the same.
- If some node receives $f + 1$ phase two message with $v \neq \perp$, *all* nodes will see at least one message with $v \neq \perp$.
- If every node sees a phase two message with $v \neq \perp$ in round r, every node adopts v as its current estimate to be used as input into round $r + 1$. Thus every process decides v in round $r + 1$.

Termination must also occur with probability 1; nodes do not wait for more messages then they are guaranteed to receive $(n - f)$, and thus proceed through consecutively increasing rounds forever unless a decision is reached. Further, at the end of each round, if no value receives a majority vote, all nodes query the random oracle. Thus it follows that in each round, r , with some probability, $\rho > 0$, all nodes choose the same value. When this happens, termination is guaranteed in round $r + 2$.

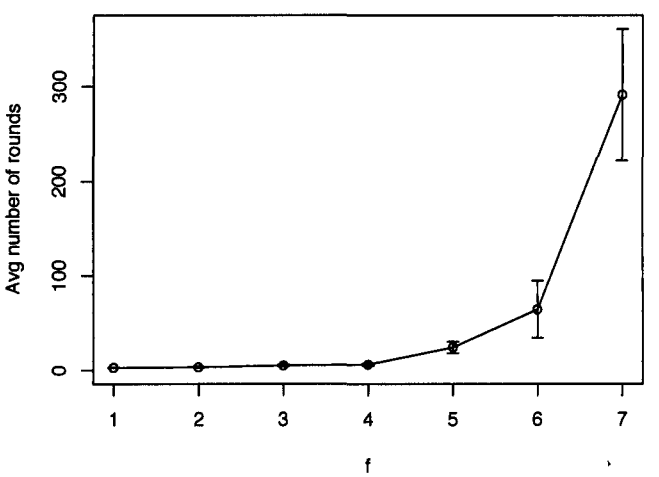

Figure 4.2: Ben-Or's consensus performance: average number of rounds vs. increasing the bound on the number of failures (f) for $n = 16$ nodes.

Figure 4.2 is an example of how Ben-Or's protocol performs in an ad-hoc network with increasing values of f. The simulations were run with the number of nodes (n) kept constant at 16. Note that

the nodes were forced to execute in lockstep; that is, all nodes progressed through the phases and rounds at the same pace.

The simulations show that the number of rounds grows exponentially versus the the number of failures, f. The reason the number of rounds increase as f increase is that increasing f reduces the number of phase one values a node waits for (on line 8 in figure 4.2). Reducing the number of values makes it less likely that a majority can be found for any single value (on line 9).

For example, when $n = 16$ and $f = 7$, all values received must be the same to reach a decision. However, if $f = 3$, the protocol can tolerate up to 4 dissenting values in a given round and still reach a decision.

Ben-Or's protocol is binary; that is, the input to the consensus protocol can only be 1 or 0. A *multi-value* consensus protocol allows the set of input values to be arbitrarily large. This is beneficial if the items to be agreed upon are not naturally binary, yes/no, decisions.

Ezhilchelvan, Moustefaoui and Raynal (EMR)'s consensus protocol is a generalization of Ben-Or's protocol to allow multi-value consensus. The full pseudo code for EMR's consensus protocol is presented in figure 4.3.

The core difference between EMR's and Ben-Or's consensus protocol is the introduction of a reliable broadcast of a node's initial proposal (line 4) before the main consensus protocol is started. This implies that eventually all correct nodes will have received the same set of initial proposals. This set of initial values is called a node's local *bag*. Upon receiving only $\perp$ values at the end of a round (line 37), a node makes its random choice from this bag (line 39) instead of flipping a coin as in Ben-Or's protocol (on line 28 in figure 4.1).

The reason the reliable broadcast is necessary is that the proof of correctness requires that eventually all correct nodes make a random choice from bags containing the *same* values. If nodes can have bags containing different values, then all the nodes may not have at least one common value. If there is no common value, then the nodes can never choose the same value and the protocol will never terminate. In Ben-Or's consensus protocol this is not an issue, as all possible initial inputs to the consensus protocol (1 or 0) is implicitly known by all nodes.

The benefit of having a multi-value consensus protocol is offset by the deteriorating performance when the number of distinct initial proposals is increased. Figure 4.4 shows how EMR's protocol performs in an ad-hoc network as the number of distinct proposals is increased. The simulations were run with the number of nodes kept constant at $n = 16$, and no failures allowed ($f = 0$). Note that $f = 0$ is the best case scenario for Ben-Or's binary protocol (see figure 4.2).

The figure indicates that the number of rounds increases exponentially as the number of distinct choices available increase. The simulation where each node proposing distinct values (16 distinct values in figure 4.4) was omitted as the time each simulation took meant it was almost impossible to get statistically meaningful results for this.

```
1   EMRConsensus(v_i)
2   {
3           bag[n] = {⊥,⊥,...,⊥};
4           RBroadcast VAL(v_i);
5           activate tasks T1, T2;
6   }
7   task T1
8   {
9           when receive VAL(v_j) from node j; do bag[j] = v_j;
10          when receive DEC(v); return (v);
11  }
12  task T2:
13  {
14          round = 1;
15          est_i = v_i;
16          while(true){
17                  send PHASEONE(round, est_i) to all processes;
18                  wait to receive (n − f) PHASEONE(round, *) messages;
19                  if(more than n/2 carry the same value, est ≠ ⊥)
20                  {
21                          send PHASETWO(round, est) to all processes;
22                  }
23                  else
24                  {
25                          send PHASETWO(round, ⊥) to all processes;
26                  }
27                  wait to receive (n − f) PHASETWO(round, *) messages;
28                  if( received at least one PHASETWO(round, est ≠ ⊥) message)
29                  {
30                          est_i = est;
31                          if(more than f received are PHASETWO(round, est ≠ ⊥) messages)
32                          {
33                              RBroadcast DEC(est);
34                              return(est);
35                          }
36                  }
37                  else (received all PHASETWO(round, ⊥) messages)
38                  }
39                          est_i = bag[random()];
40                  }
41                  round++;
42          }
43  }
```

Figure 4.3: Ezhilchelvan, Mostefaoui and Raynal's randomised consensus protocol. Adapted from [EMR01]

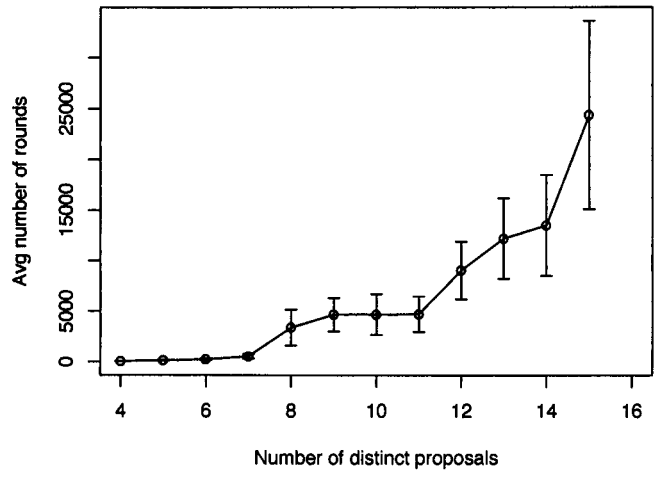

Figure 4.4: EMR's consensus protocol: average number of rounds vs. increasing number of distinct proposals for $n = 16$ nodes.

Finally, we note that the message complexity per round for both EMR's and Ben-Or's consensus protocol is around $2n$ (1 broadcast per node per phase).

4.4 A randomised consensus protocol for ad-hoc networks

Randomised consensus is an appealing solution for ad-hoc networks for a number of reasons:

- A random oracle incurs no message passing overhead. A failure detector inevitably incurs some, which is undesirable in an ad-hoc network; not only is the creation of routes for all the heartbeat messages expensive, but the exchange of these heartbeat messages may in extreme cases cause interference to the extent that connectivity is lost.

- A randomised consensus solution is completely decentralised. Most failure detector based solutions uses a centralised coordinator, thus potentially reducing availability when the coordinator is in a different network partition.

- A randomised consensus solution tolerates completely asynchronous networks. A failure detector based solution requires periods of network stability where no nodes suspect the current coordinator to make progress. In a local area network the stability assumption is probably justified, but in an ad-hoc network it may not be, as appropriate timeouts might be difficult to choose.

For these reasons, and as it has not been done specifically for ad-hoc networks before, the randomised consensus approach was adopted in this dissertation.

However, as we have seen, the randomised consensus protocols as presented in section 4.3.2 have a few drawbacks. First, as the number of potential failures, f, is increased as a fraction of the total number of nodes in the system, the expected number of rounds increases exponentially (see figure 4.2). Second, the message overhead per round is relatively high for ad-hoc networks, requiring $O(n)$ broadcasts per round.

Finally, extending the consensus to allow input values from an arbitrarily large set appears to further increase the expected number of rounds as seen in figure 4.4, and require an additional reliable *broadcast* to start with. As shown in theorem 2.3, guaranteeing all correct nodes receive a given message is impossible if node crashes are undetectable and storage subsidence is essential (see section 2.3.2).

A further complication is that the bound on the number of failures, f, must be revisited. In the wired network context, where it is typically assumed that failures are detectable, and thus all correct nodes can be guaranteed to receive a message, each node can expect to receive $n - f$ values at the start of each phase (e.g. on line 8 and 17 in Ben-Or's protocol). As long as $f < n/2$, receiving $n - f$ nodes will always ensure that a node receives a majority of values.

However, if the dissemination primitive can only guarantee that $n - f$ nodes will receive any given message, a node can only expect to receive $n - 2f$ values at the start of each phase. This is because of the $n - f$ nodes which receive a message, f can crash, thus leaving just $n - 2f$ nodes to send messages in the next phase. If a node can only receive $n - 2f$ messages, then if $f > n/4$, $n - 2f$ may not be a majority of n. If a node does not receive a majority of values in EMR's and Ben-Or's protocol, it can never reach a decision.

For this reason, the safe option is to let f be less than $n/4$. We will assume $f < n/4$ up until section 4.4.4 where we will show how the protocol can still reach a decision with up to $f < n/2$. Note here that an implication of theorem 2.3 is that maximum $n - f$ (and thus $n - 2f$ correct) nodes can ever be assumed to have received a decision, though this limitation is dealt with in chapter 5.

The contributions of this chapter is to show how these drawbacks can be alleviated. We will:

Replenish optimisation: Present an optimisation to EMR's protocol which reduces the increase in the expected number of rounds as the number of distinct inputs is increased from exponential to linear. This optimisation is not ad-hoc network specific, and is presented in section 4.4.1.

Adopt optimisation: Show how the highly variable message delivery latencies associated with ad-hoc networks can be used to reduce the number of rounds to between 2 and 4, irrespective of the number of distinct proposals or the number of failures. This optimisation is presented in section 4.4.2.

Cross layer optimisation: Show how combining the reliable dissemination protocol and the randomised consensus protocol we can reduce the per round message overhead by 1-2 orders of magnitude. This optimisation is presented in section 4.4.3.

Benefit of combining optimisations: Show how combining the adoption and cross layer optimisations produces a protocol which can tolerate $f < n/2$ failures instead of just $f < n/4$. This is shown in section 4.4.4.

Proof of correctness: Present a new proof of correctness of a protocol combining all these optimisations. The proof is presented in section 4.4.5, and crucially removes the need for an initial reliable broadcast.

4.4.1 Using randomization to reduce choice

The sharp increase in expected number of rounds for EMR's protocol shown in figure 4.4 makes intuitive sense; as the number of distinct values is increased, the chance that a majority of nodes choose the same value in any given round decreases correspondingly.

It is the reliable broadcast of all nodes' initial proposals which is why the number of possibilities is so large. As discussed in section 4.3.2, this reliable broadcast is a core part of the proof of correctness of EMR's protocol, which requires all correct nodes to eventually have the same set of values to choose from in their local bags.

However, reliably broadcasting all initial proposals appears to be unnecessary if we introduce the concept of *replenishing a node's bag* in each round. Figure 4.5 details how the replenishment works.

R1: Each node, N_i, starts in round $r = 1$ with its local bag containing only the node's input (it's initial preference value, v_i).

R2: In phase one of round r, each node adds the $n - 2f$ values it guaranteed to receive to the local bag.

R3: At the end of phase two of round r, if no decision is reached, a new est_i is chosen randomly from the local bag and the bag is emptied. After this, the newly adopted estimate is added to the bag and round $r + 1$ is entered into.

Figure 4.5: The replenish optimisation.

What this optimisation affords us is an alternate proof of termination. All we need to show to prove eventual termination is that the set of distinct values from which nodes make their random choice eventually contains only one value.

The full new proof of termination is presented when proving the correctness of the combined protocol in section 4.4.5, but the following proof sketch outlines the intuition behind it:

If there is more than one distinct value in a round r, then it is not possible for more than one of those distinct values to be the only choice available to a node. This follows from the fact that all values are disseminated to at least a majority of nodes, and that all nodes wait for at least a majority set of values. If some node happens to receive a majority of values all equal to some distinct value, v, then by the properties of majority sets, that value v exists in all other nodes' majority sets. Thus any other distinct value in round r could not be the only available choice to any node.

Because of this, in every round r where there exists more than one distinct value, there is some probability, p, (strictly greater than 0) that at least one distinct value is not chosen in that round r. Further, when a value v is not chosen by some node in round r, it never appears in any round $r' > r$. For this reason the set of distinct values shrinks in each round with some $p > 0$.

As the protocol does not terminate until a decision has been made, the number of rounds keeps increasing. In the limit as the number of rounds tend to infinity, the probability that the set of distinct values only contains one value is equal to 1. When there is only one distinct value to chose, all nodes adopts this value as their estimate and the protocol is guaranteed to terminate in the next round. Hence eventual termination is ensured without requiring an initial reliable broadcast.

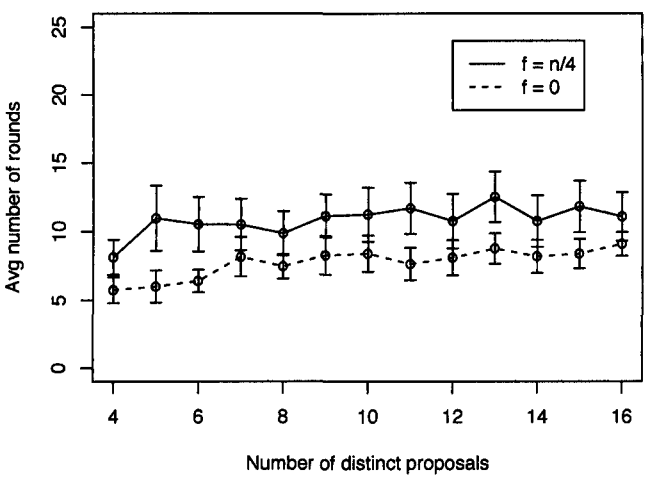

Figure 4.6: Average number of rounds vs. increasing number of distinct proposals for 16 nodes.

Recall that the number of rounds until a decision is reached in the unmodified version of EMR's protocol increases exponentially as the number of distinct proposals increase (figure 4.4). We are interested to see if the replenish optimisation can reduce the effect of increasing the number of distinct proposals.

Figure 4.6 shows how increasing the number of distinct proposals impacts EMR's protocol when using the replenish optimisation. The simulations were run with $n = 16$ nodes for $f = 0$ and $f = n/4$.

The number of distinct proposals were varied between 4 and 16 (all nodes proposing distinct values). As in figure 4.4, all nodes were forced to act in lockstep; that is, all nodes proceeded through the phases and rounds at the same rate.

The simulations indicate that the replenish optimisation is effective. The number of rounds now appears to increase only slightly as the number of distinct proposals increase. This is in sharp contrast to the performance of the unmodified version of EMR's protocol with the same simulation parameters.

For example, using the unmodified version of EMR's protocol with 15 out of 16 nodes proposing distinct values meant it took on average of 26000 rounds to reach a decision. With the replenish optimisation, the average number of rounds is roughly 7. Further, increasing f to $n/4$ also appears to have a very limited effect.

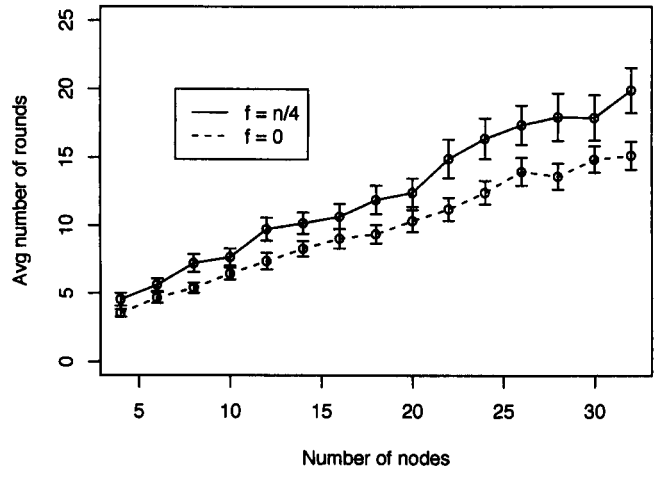

Figure 4.7: Average number of runs with increasing n with each node proposing a distinct value.

Figure 4.7 suggests that this benign behavior is not peculiar to $n = 16$. In these simulations each node proposed a distinct value, and the number of nodes was varied between 4 and 32. The average number of rounds until a decision is reached appears to grow linearly with a factor of less than 1, at least for $f = 0$ and $f = n/4$.

In all, we have seen how the replenish optimisation: (i) reduces the impact of increasing the number of distinct proposals in terms of the number of rounds required until a decision is reached, and (ii) removes the need for an initial reliable broadcast.

Finally, we note that the optimisation could probably work well in wired networks as well, as the changes described here are not specific to ad-hoc networks; recall that all nodes were forced to act in lockstep, thus removing the impact of the underlying network being ad-hoc.

All nodes acting in lockstep is unlikely to happen in a real ad-hoc network, but as we shall see next, this is not necessarily a bad thing.

4.4.2 Taking advantage of a noisy environment

Ad-hoc networks are *noisy*; transient network partitions, radio interference, node mobility and MAC layer scheduling algorithms all contribute to the level of noise.

The effect of this noise is highly variable message latencies, so that nodes no longer proceed through phases and rounds in lockstep.

The histogram in figure 4.8 backs this up; it shows an example of how long it takes for various nodes to complete a phase (receive $n-2f$ messages) in a run with $n = 50$, $f = 10$ and 250m wireless range. In this experiment all nodes started the phase at the same time.

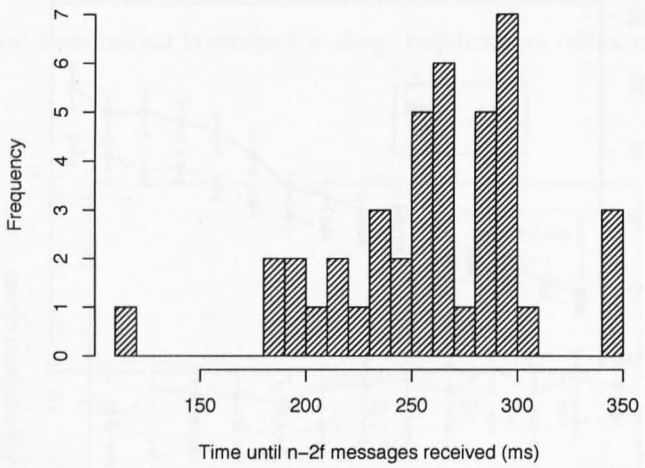

Figure 4.8: A histogram of the time it takes various nodes to receive $n - 2f$ messages have been received (a phase is complete) in a typical simulation run. All nodes started the phase at the same time. Wireless range = 250m.

We can take advantage of this disparity by making slow nodes *adopt the choice* of quicker ones. The change required to the protocol is outlined in figure 4.9.

Neither the safety nor the termination properties are affected by this change: A slower node which receives a message with a higher round or phase number can safely abandon its current estimate, as the higher round or phase can only have been entered into after some node has received a majority of values. Therefore, the estimate received in the higher round/phase message is a value which the slower node could have adopted itself, if it was quicker. Further, as long as at least one node makes a random choice, FLP is still circumvented; there is no need for *every* node to make a random choice

A1: A node receiving a message with higher round or phase numbers than it is currently in, stops executing the lower round/phase and sets it its round/phase numbers to the received numbers.

A2: After changing its round/phase numbers, the node adopts the estimate, *est*, in the received message as its own. The protocol then starts executing the adopted phase/round by disseminating the (adopted) *est* to $n - 2f$ nodes.

Figure 4.9: The adopt optimisation.

as long as at least one does so.

Making slower nodes adopt the values chosen by quicker ones can be seen as electing (a small number of) de-facto coordinators or leaders which impose their choice on others. However, unlike the coordinators in failure detector based approaches where coordinators are predetermined in each round, these de-facto leaders are always the nodes which obtain a majority of messages in the shortest time.

The effect of this change can be seen in figures 4.10 and 4.11, which shows how increasing n and varying the number of distinct initial proposals impacts the optimised protocol.

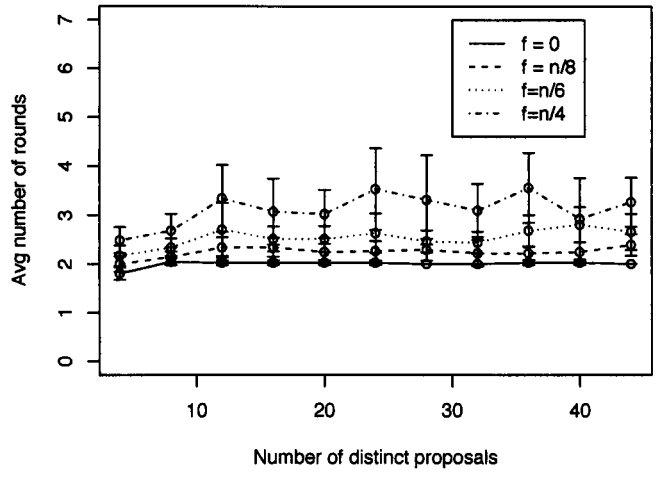

Figure 4.10: Average number of rounds vs Number of distinct proposals for $n = 50$. Wireless range = 250m, max speed = 5m/s.

The optimised consensus protocol now appears almost unaffected by either increasing the number of nodes or the number of distinct initial proposals. Contrast this with the exponential increase seen by the unmodified version of EMR's protocol in section 4.3.2. The only variable with any impact is the value of f, and even here the impact appears to be a small constant. This makes multi-value

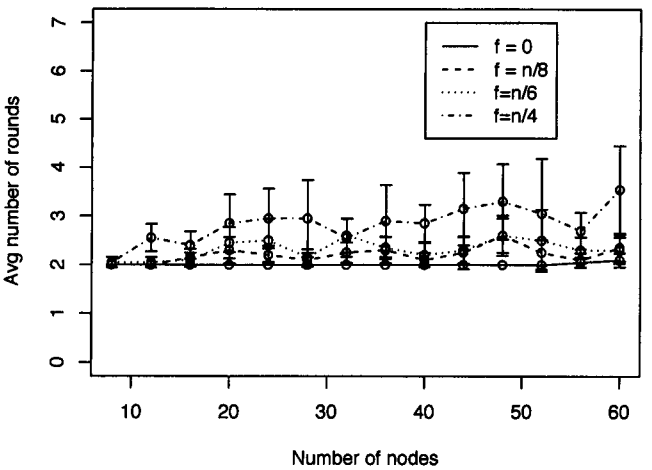

Figure 4.11: Average number of rounds vs Number of nodes (n), with all nodes proposing distinct values. Wireless range = 250m, max speed = 5m/s

randomised consensus a practical possibility in ad-hoc networks.

The improvement seen by this optimisation is dependent on how much noise there is in the environment. Unlike with a failure detector based approach which require periods of stability to make progress, increasing the level of noise appears to have a benign effect on the expected number of rounds. That being said, the results reported in this section are perhaps somewhat artificial, as they rely on the randomness in the simulator used to conduct the experiments, and not "real-world" noise. However, it is probably reasonable to assume that a real ad-hoc network has at least some level of noise associated with it, in which case this optimisation should work well.

4.4.2.1 An interesting possibility

Given that the ad-hoc networking environment has an inherent level randomness associated with it, an interesting possibility arises; one could design a completely deterministic consensus protocol which could circumvent the FLP impossibility result by taking advantage *only* of randomness in the environment.

For example, if EMR's protocol with the adoption optimisation outlined above was changed so that each node reverted to the estimate it held at the start of the round instead of choosing randomly when no majority is received, then the EMR protocol itself would no longer contain any randomness. However, given a sufficient level of randomness in the environment, the protocol would still terminate.

The proof of correctness would have to be changed so that there was a probability, $p > 0$, in each

round that the fastest node in the previous round is able to "impose" their estimate on at least a majority of nodes in phase one before these nodes revert to their own estimate. A benefit of this approach is its simplicity; there is no need to keep a bag of other nodes' values as they are never needed.

Such an approach is very similar to the "fair/noisy scheduling" approach as discussed in section 4.3.2. As with these approaches, the probability of termination depends crucially on the randomness inherent in the environment being sufficient, something which may not always be the case. Further, embedding a random choice in the actual algorithm typically incurs very low overheads, and ensures eventual termination should the randomness in the environment be insufficient.

4.4.3 Reducing overhead using a cross layer optimisation

The two optimisations presented so far has shown how making further use of randomness in the protocols and embracing the randomness inherent in the environment can dramatically reduce the number of rounds required to reach a decision. However, the cost in terms of message passing overhead per round remains high (around $2n$ broadcasts per round). In an ad-hoc network this large number of messages can cause serious problems, even when the underlying dissemination protocol is relatively efficient.

In this section we will see how the per round message passing overhead can be reduced based on the following observations:

- There is no need to know which node voted for what value.
- Only one correct node needs to be guaranteed to receive a majority of votes to progress through rounds and phases until the eventual decision.

Both EMR's and Ben-Or's consensus protocols are stacked on top of a reliable broadcast primitive unaware of how it implements the service it provides. Similarly, the reliable broadcast primitive has no semantic information indicating the significance of any message it is disseminating. A *cross layer optimisation* puts protocols side by side instead of stacking them, making them aware of each others functionality and thus able to perform more efficiently overall.

We now present a cross layer optimisation which works by putting the reliable dissemination primitive described in the previous chapter side by side with the randomised consensus protocol. The full description of the changes is given in figure 4.12, and assumes familiarity with the reliable dissemination protocols presented in chapter 3.

The gist of the optimisation is to have *one* reliable dissemination per phase instead of n. The nodes participate in this dissemination and add their estimates to the message body (if not already present) prior to signing the message. In this way a signature acts as a vote for one of the values in a given message. By the properties of the reliable dissemination protocol at least one correct node will

C1: The exchange of estimates is done solely through special *consensus messages*. A consensus message's unique id, *uid*, consists of the round and phase in which the message was originated.

C2: A node initiating a phase (whether phase one or two) does so by initiating a reliable dissemination of a consensus message with its current estimate in the body of the message, and $k = \lceil (n+1)/2 \rceil$ (i.e. the consensus message is guaranteed to be received by a majority of nodes).

C3: A node receiving a consensus message with a given phase/round number for the first time participates in the dissemination of that message. The node adds its estimate to the message body (if not already present) and only then signs the message. This replaces initiating a separate dissemination of its estimate at the start of phase one or two. A signature acts as a *vote* for at least one of the values in the message body.

C4: Every time a node receives a consensus message, it adds all estimates it has received in the current phase to the message body, prior to merging the signatures as in the dissemination protocol.

C5: Upon realising a consensus message, a node inspects the contents of the message body to decide which action to take. This replaces inspecting all messages after having received $n - 2f$ of them, at the end of phase one or two.

C6: If a node realises a message, there are two possibilities: (i) a decision is reached or (ii) the next phase/round is entered into. Only in the first instance is a realisation message sent (containing also the decided value). In the second case, the node initiates the dissemination of the subsequent round/phase consensus message. This removes the need for a realisation message.

Figure 4.12: The cross layer optimisation.

realise the consensus message. This correct node therefore has at least a majority of votes and can legitimately decide which action to take and start the next dissemination of the higher phase/round consensus message. This will continue until `decide()` is invoked.

This optimisation is included in the full proof of correctness given in section 4.4.5. Note that the push-pull optimisation presented in section 3.3.1 cannot be used in conjunction with this optimisation.

It would be perfectly feasible to require $k = n - 2f$ instead of only waiting for a majority of votes (see **C2**). As we saw with Ben-Or's protocol; waiting for more values makes it easier to find a majority which agree, and thus should reduce the number of rounds until a decision.

However, for this to make a difference each vote would have to be associated with a particular value in the message body so that a majority of votes could be associated with a specific value. For purely pragmatic reasons this was not done; the dissemination protocol is obviously much quicker with lower values of k and leads to smaller transmission overheads. Also, the optimisations described in the previous two sections, which were tailored to reduce the number of rounds, render the additional reduction unnecessary.

The following figures shows how the optimised version of EMR's protocol (including both the replenish and the adopt optimisations), performs with and without the cross layer optimisation. We are interested in how much the cross layer optimisation can reduce the per round transmission overhead, and also if this reduction avoids increasing the average number of rounds until a decision.

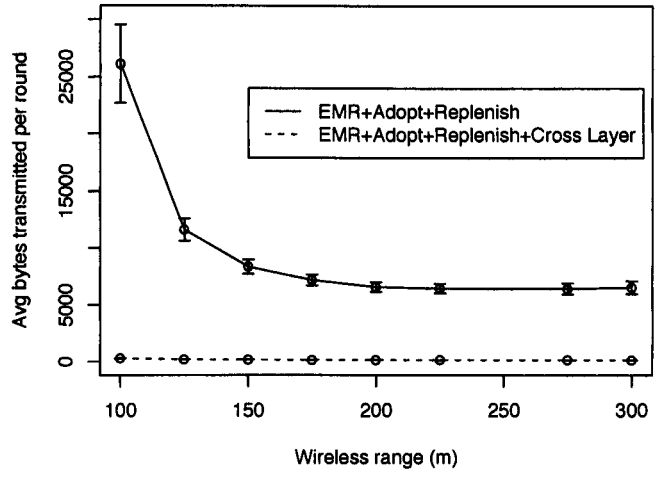

Figure 4.13: Average transmission overhead per node vs Wireless range. Max speed = 5m/s.

Figure 4.13 shows the transmission overhead per node per round both before and after the cross layer optimisation, as density is varied. The number of nodes, n, and the number of failures, f, was

fixed at 50 and 10 respectively. The number of distinct proposals was varied between 1 and 40.

The figure indicates a 1-2 orders of magnitude reduction in transmission overhead. The improvement is large enough so that the overhead associated with the cross layer optimised protocol barely registers on the graph.

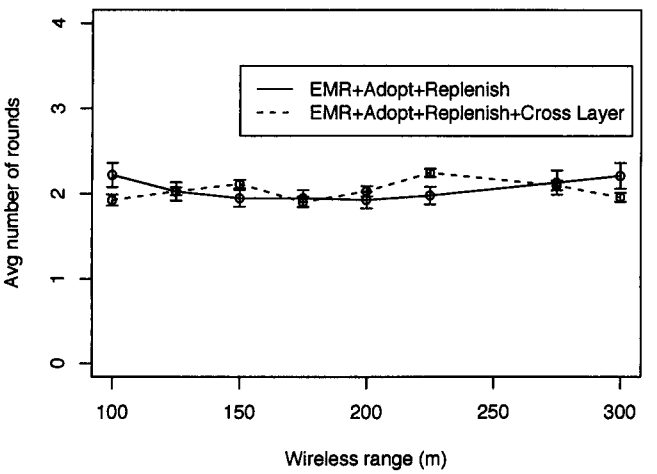

Figure 4.14: Average number of rounds vs Wireless range. Max speed = 5m/s.

Figure 4.14 indicates that this reduction in overhead has not come at a significant increase in the number of rounds; it shows the average number of rounds for the same simulation parameters as figure 4.13. A further improvement, as seen in figure 4.15, is that the latency (the time until `decide()` is invoked) is reduced at lower densities as a result of the optimisation. This is because the cross layer optimisation only needs a majority of votes to proceed, and because it reduces interference by reducing the number of messages being transmitted.

To summarise, the cross layer optimisation appears to reduce the transmission overhead per round by 1-2 orders of magnitude without impacting the number of rounds and even slightly decreasing the latency of the protocol. This makes the protocol more usable in ad-hoc networks where bandwidth is at a premium.

4.4.4 Putting it all together

We have seen how the 3 optimisations presented benefits the protocol in various ways. However, a further benefit arises when the adoption and cross layer optimisations are combined; we can now (again) tolerate up to $f < n/2$, instead of just $f < n/4$ failures.

The reason is as follows: The cross layer optimisation ensures that at least one correct node will receive a majority of votes (because of the properties of the reliable dissemination protocol).

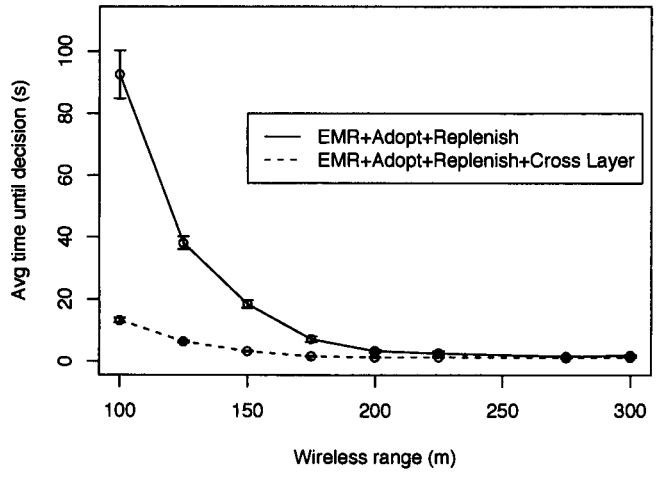

Figure 4.15: Average time until `decide()` is invoked vs Wireless range. Max speed = 5m/s.

That correct node will initiate the dissemination of the next round or phase consensus message (see **C6**). Further, adding the adoption optimisation means that it is sufficient to have one correct node starting in each phase. That correct node will keep disseminating the consensus message until it has a sufficient number of votes, making nodes which are "stuck" in previous rounds adopt its value as discussed earlier.

This is in contrast to the case when there is no adoption or cross-layer optimisation. Here the progress of nodes is dependent on being able to deliver the current round's messages to f nodes more than it can expect to hear back from in the next round. This is because f of the nodes can crash.

A comprehensive study of the combined protocol with up to $n/2$ failures has been carried out, and the results can be seen in appendix B.2. The results generally speaking show the same benefits as indicated when studying each optimisation. In addition, the trends associated with changing variables such as number of nodes, wireless range or node speeds, closely mirrors that of the reliable dissemination protocol reported in the previous chapter. This is as expected.

Figure 4.16 shows the pseudo code for the combined protocol. Note that the adoption optimisation has been left out, as it is easy to describe, but less easy to present elegantly; the full pseudo code with this included is given in the appendix.

The next section gives the proof of correctness of the combined protocol.

4.4.5 Proof of correctness

Roadmap to proof:

```
1
2   Consensus(preference)
3   {
4           bag[] = ∅;
5           round = 1;
6           phase = 1;
7
8           while(true)
9           {
10                  est[] ={preference};
11                  disseminate consensusMsg(phase, round, est[]) to n/2.
12                  wait until consensusMsg has more than (n/2) signatures;
13                  bag[] = est[];        //replenish bag with received values and current estimate
14                  if( |est[]| > 1)
15                  {
16                          est[] ={⊥};          //then more than one value proposed
17                  }
18                  phase = 2;
19                  disseminate consensusMsg(phase, round, est[]) to n/2 nodes;
20                  wait until consensusMsg has more than (n/2) signatures;
21                  if( est[] contains at least one value, v  ≠ ⊥)
22                  {
23                          preference = v;
24                          if(  est[] contains only one value, v  ≠ ⊥)
25                          {
26                              decide(v); //sends a realisation packet with decided value.
27                          }
28                  }
29                  else (est[] only contains ⊥)
30                  {
31                          preference = bag[random];
32                  }
33                  bag[] = ∅; //empties bag
34                  round++;
35                  phase = 1;
36          }
37  }
38
```

Figure 4.16: Pseudo code for the combined protocol

1. We first show that either at least one correct node is guaranteed to participate in a phase one of round 1, or all nodes which participated in the round crashes before entering phase two (lemma 4.1).

2. Lemmas 4.2 and 4.3 then show that if at least one correct node participates in phase one of round 1, then at least one correct node will participate in every phase and round until a decision is reached.

3. Lemma 4.4 then shows that if all nodes start phase one of some round with the same value, at least one node will decide on that value at the end of that round.

4. Using the replenish optimisation, lemmas 4.5 and 4.6 shows how there is a non-zero probability that some value is not chosen in each round, and how if a value is not chosen it will not appear in any later round.

5. Theorem 4.1 then proves eventual termination by using lemmas 4.1-4.3 to show that if a decision is not reached in some round, the next round is always entered into (progression), and showing how as the number of rounds tend to infinity, lemmas 4.5-4.6 ensure that eventually there will be only one distinct value in all nodes' bags. The proof of the theorem then concludes by using lemma 4.4 to show that in the round when there is only one distinct value in all nodes' bags, all nodes will chose the same value and at least one node will decide on that value by the end of the round.

6. Finally, theorem 4.2 proves uniform agreement by showing how if a node has made a decision in some round, that decision is the only possible decision that can be reached by any other node (whether correct or not). This is done by pointing out the relevant parts of the algorithm which ensures this behaviour.

The proof assumes $f < n/2$, which is a necessary requirement for randomised consensus protocols[Ben83].

Definition (vote). *A* vote *is a node's signature plus the node's estimate and other received values in the current phase added to the message body if different from values already in message.*

Definition (participate). *A node is said to* participate *in a phase if it actively disseminates a phase 1 or phase 2 message after having added its vote to the message.*

Definition (initiate). *To* initiate *a round or phase is defined as starting the dissemination of new message after having received a majority of votes in the previous round or phase.*

Lemma 4.1. *If a phase 1 of some round r is initiated by some node, faulty or not, either a correct node participates in the phase or all nodes crash without moving into phase 2.*

Proof. As $f < n/2$, in order for any node to move into phase 2, at least one correct node must participate in phase 1. Hence the lemma.

□

Lemma 4.2. *If a correct node participates in phase 1 of some round r, eventually at least one correct node participates in phase 2 of round r.*

Proof. By contradiction. Assume there is a correct node in phase 1, but never a correct node in phase 2. The only way a correct node in phase 1 will stop participating in phase 1 without moving into phase 2 is if at least $f + 1$ other nodes are participating in phase 1. In this set of participating nodes there must be another correct node. Hence if ever at least one correct node participates in a phase 1, at least one correct node will keep participating in phase 1 unless it moves into phase 2.

As $f < n/2$, and the dissemination protocol is reliable, a correct node in phase 1 will eventually receive a phase 1 message with a majority of votes and stop waiting at line 12. When this happens, the correct node moves into phase 2, becoming the correct node in phase 2. This is a contradiction, hence the lemma. □

Lemma 4.3. *If at least one correct node participates in phase 2 of round r, eventually at least one correct node participates in phase 1 of round $r + 1$, or decides.*

Proof. By a similar argument to Lemma 2; assume there is never such a correct node which decides in round r or moves into phase 1 of round $r + 1$. For this to hold the correct node (of which there must always be at least one) must block forever waiting for a majority of votes on line 20. As a majority of nodes are correct, eventually the correct node will receive a phase 2 message with a majority of votes and stop waiting. When this happens the correct node will either move into phase 1 of round $r + 1$ or decide. This is a contradiction, hence the lemma. □

Lemma 4.4. *If all nodes which initiate phase 1 of round r do so with their estimates equal to the same value $v \neq \perp$, and at least one correct node participates in phase 1 of round r, then at least one correct node decides v at the end of round r.*

Proof. As per the lemma assumption, all nodes which initiate the round do so with their estimates equal to the same value v. These estimates are added to the phase 1 message. Any nodes which did not initiate round r, but which receives the phase one message will adopt v as their estimate and add their vote to the phase 1 message. As there is only one distinct value added to the phase one messages, it follows that any node which moves into phase 2 after receiving a majority of votes will evaluate the comparison on line 14 to be false and leave v as the only value added to any phase two message of round r. By Lemma 2 eventually at least one correct node will participate in phase 2. By lemma 3 as at least one correct node is participating in phase 2, eventually at least one correct node will receive a phase 2 message with a majority of votes and either participate in phase 1 of

round $r+1$ or decide. As there is only distinct value, $v \neq \perp$, in any phase 2 message, any node which receives a phase 2 message with a majority of votes will evaluate the statement on line 24 to be true and will thus decide v on line 26. □

Definition ($\mathcal{B}_i$). *$\mathcal{B}_i$ is node N_i's local bag of values which it replenishes on line 13, chooses randomly from at line 31 and empties at line 33. Initially $\mathcal{B}_i$ contains only node N_i's input value to the consensus protocol.*

Definition ($\mathcal{C}_r$). *$\mathcal{C}_r$ is the set of* distinct *available values over* all *local bags, $\mathcal{B}$, in some round r. This constitutes the set of possible values which nodes that initiate round r can choose their preference value from. Initially $\mathcal{C}_1$ = all the distinct initial inputs to the consensus protocol.*

Lemma 4.5. *If a value, $v \neq \perp$, is not chosen by any node in phase one of round r, it will not appear in any $\mathcal{C}_{r'}$, for any $r' > r$.*

Proof. By definition; the local bag, $\mathcal{B}_i$, that a node N_i chooses its preference from if it initiates a round $r+1$ is made up of a majority of the choices made by other nodes in round r. As $\mathcal{B}_i$ is emptied at the end of round r (line 33), if no node choses v and disseminates it in phase 1 of round $r+1$, no node can receive v on line 12, thus $v \notin \mathcal{C}_{r+1}$. □

Lemma 4.6. *If $|\mathcal{C}_r| > 1$ during some round r, there is a probability, p (strictly greater than 0), that some value $v \in \mathcal{C}_r$ is not chosen by any node.*

Proof. By contradiction; assume there is a round, r, where $p = 0$. For this to hold each value, $v \in \mathcal{C}_r$ has to be guaranteed to be chosen by at least one node in r. For each value $v \in \mathcal{C}_r$ to be guaranteed to be chosen, each v has to be the *only* choice available to at least one node. However, if a value v is the only distinct choice available to a node N_i, then v must also be in every other node's bag, as the bags contain at least a majority of values and all majority sets intersect. If v exists in all bags, then clearly another value w cannot be the only available choice to any node. This is a contradiction, hence the lemma. □

Theorem 4.1 (Eventual Termination). *If at least one correct node participates in a consensus, then at least one correct node eventually decides with probability 1.*

Proof. First note that if no correct nodes participate in a given consensus then by Lemma 1, all nodes which participate will crash in phase 1 of round 1, in which case no other node will ever know about the consensus and it can be ignored.

Further, by Lemma 2, as a correct node participates in phase 1 or round r, eventually a correct node will participate in phase 2 of round r. By Lemma 3, if there is no decision in round r, there will be at least one correct node participating in phase 1 of round $r+1$. Thus, starting with a correct

node participating in round 1, if no node decides before round r, it follows that the protocol does not block in any round $r' < r$.

Now, if in a round r there is $d > 1$ distinct value to choose from ($|C_r| > 1$), then there is a probability $p > 0$ that some value $v \in C_r$ is not chosen in r (Lemma 6). If this happens, v will not appear in any $C_{r'}$, for any $r' > r$(Lemma 5). As a result, for each round r where $|C_r| > 1$ there is a non-zero probability that $|C_{r+1}| < |C_r|$; that is, after 1 round there is a non-zero probability that at least one value is removed.

Hence there is a probability $P(\alpha) = p + p(1-p) + p(1-p)^2 + ... + p(1-p)^{\alpha-1} = 1 - (1-p)^\alpha$ that $|C_{r+\alpha}| < |C_r|$. As $\lim_{\alpha \to \infty} P(\alpha) = 1$, it follows that at least one values is removed after a finite number of rounds with probability arbitrarily close to 1.

Repeating this argument $d-1$ times, we conclude that all nodes which participate in a consensus will only have one distinct values to adopt as their estimate, with probability arbitrarily close to 1.

Once this happens, according to Lemma 4, a correct node will make a decision. □

Theorem 4.2 (Uniform Agreement). *No two nodes decide distinct values.*

Proof. Let r be the first round during which nodes decide. There are two cases:

1. Two nodes, N_i and N_j both decide during r. They decide v_i and v_j respectively. (Note that due to the test on line 24 these are necessarily not $\perp$). Because of lines 12 and 14 we conclude that N_i received a majority of votes for its single decision value v_i in phase 1. Similarly N_j received a majority of votes for its single decision value v_j in phase 1. As nodes do not change their vote during a phase 1, it follows that there is a node N_k which added its vote to both N_i and N_j phase 1 message. Hence $v == v_j$.

2. A node N_i decides during r, while another node N_j decides during some later round $r' > r$. As node N_i received phase a 2 message with at least a majority votes containing only v in round r, the phase 2 message with at least a majority of votes received by any node not deciding in round r must contain v. For this reason, all nodes which do not decide during r will adopt v as a preference on line 23 and proceed to round $r + 1$. Therefore, v is the only estimate value present in any round $r' > r$, and thus the only value which N_j can decide on.

□

4.5 Related work

There is a large amount of literature about the consensus problem in the wired networking context (see [GHM00] for a concise overview), but relatively little has been done specifically for ad-hoc networks. A number of papers has addressed related agreement problems, including mutual

exclusion[MVW01][WWV01] and leader election[VKT04], but none of these have attempted to provide both strong guarantees and tolerate crash failures.

An example of this is Vasudevan et al.'s leader election protocol[VKT04] which tolerates crash failures, but only eventually guarantees a distinct leader in each "connected component"; a much less useful abstraction than electing a global leader. Conversely, Welch et al. [MVW01][WWV01] uses a token circulating in the ad-hoc network which ensures mutual and k-mutual exclusion, but their algorithms are not guaranteed to work if nodes can crash.

Another popular approach has been to weaken the safety properties (as opposed to the liveness property) to be only probabilistic; Luo et al.[LEH04] uses their probabilistic gossiping protocol[LEH03b] to provide a lightweight probabilistic group communication system for ad-hoc networks. This system provides "tunable reliability", but is unable to provide anything more than "probabilistic safety" properties; that is, there is always a possibility that two correct nodes decide *different* values.

The simplest way to provide a consensus solution in ad-hoc networks is probably to use an existing wired network solution. There has been a lot of work in providing best-effort unicast and multicast protocols suitable for ad-hoc networks, and the Internet Engineering Task Force (IETF) is currently working towards standardizing these. With the availability of such protocols, a naive solution would be to simply use a group communication toolkit like Ensemble or NewTOP[EMS95] (which contain failure detector based consensus solutions) unmodified. This would probably be unwise, as the assumptions underlying the design of these toolkits may not hold in ad-hoc networks. For example, most wired network group communication toolkits assume that unicast is a relatively straightforward, low overhead operation, and thus use it extensively. However, in ad-hoc networks unicast may very well incur substantial overheads, as routes must be discovered and maintained in face of a potentially very dynamic network topology.

An interesting approach taken by Friedman et al.[Fri03][FT03] is to use an existing group communication toolkit and modifying it to better suit ad-hoc networks. Their group communication toolkit, *JazzEnsemble*, uses Ensemble as its base, and aims to provide both strong guarantees and fault tolerance in a manner better suited to ad-hoc networks.

JazzEnsemble is the only other approach designed specifically for ad-hoc networks which provides both strong safety guarantees and tolerates crash failures. Further, this work can be seen as a feasibility study of the failure detector approach in ad-hoc networks, and nicely complements the use of randomization to provide consensus adopted in this dissertation. For these reasons JazzEnsemble is further discussed in the next section.

4.5.1 JazzEnsemble

The JazzEnsemble group communication toolkit has adopted the layered approach of Ensemble, replacing the layers deemed unsuitable for the ad-hoc network environment. The main changes

which make up JazzEnsemble is the introduction of *fuzzy group membership* and the provision of *middleware level routing.*

The need for so called *fuzzy group membership* was based on their study of the feasibility of using failure detectors in ad-hoc networks [FT03]. In this study a novel gossip based failure detector specifically designed for ad-hoc networks is introduced and evaluated. The main change compared to the basic gossip based failure detection service discussed in section 4.3.1 is that instead of gossiping with any node in the network (regardless of topology), a node gossips with one of its neighbours. The study showed that failure detection in ad-hoc networks is possible, although not straight forward. For example, in sparse networks the failure detector frequently falsely suspected nodes due to route disconnections, and in general, a relatively high timeout had to be set to avoid too many false suspicions.

Fuzzy group membership is introduced to alleviate some of these issues by introducing a level of "fuzziness" associated with a node's membership to a group. This is opposed to the binary, either a node is connected or it is faulty, relationship found in Ensemble. The fuzziness level of a node can relate to simple things like the number of missed heartbeats, or more advanced measures such as measured bandwidth, estimated round trip time, sending success rate, etc.

The benefit of fuzzy group membership is that nodes are not dropped immediately on becoming slightly fuzzy. Instead the group communication toolkit's components can decide to behave differently towards fuzzy nodes. For example, the flow control mechanism may decide to continue sending new messages if it is lacking acknowledgments for older messages from fuzzy nodes only. The buffer management mechanism may choose to compress these older messages, while the reliable retransmission mechanism could decide to stop proactively sending these. Either way, the scheme avoids immediately dropping fuzzy nodes which are not actually faulty, though of course once a fuzziness level reaches a given threshold the node is removed from the current view.

In Ensemble, routing is considered a network layer operation and it does not concern itself with how messages are forwarded. In JazzEnsemble, an application developer may choose to use the underlying routing mechanisms in the same way, if these are available. However, JazzEnsemble also includes its own routing mechanism, called *middleware level routing*, as the authors argue that JazzEnsemble can sometimes do its own routing more efficiently. For example, middleware level routing can make use of the periodic heartbeat messages, already required by the failure detection mechanism, to obtain accurate neighbourhood information. Further, unicast messages in JazzEnsemble typically only follows a multicast message where the multicast source is the unicast destination (the unicast is typically an acknowledgment), and if reverse paths are kept, the number of unicast routes which must be maintained by the network layer is reduced.

As of writing, no performance measurements of JazzEnsemble has been published, though it is a promising approach and nicely complements the approach taken in this dissertation. However,

even with the modifications to suit ad-hoc networks, JazzEnsemble still uses at its core a group communication system which assumes quite a high level of stability. It is therefore unlikely to work well in the most challenging types of ad-hoc networks considered in this dissertation, such as for example frequently partitioned networks.

4.6 Conclusion and summary

In this chapter we have argued that a randomised consensus solution is appealing for mobile ad-hoc networks, as these are decentralised in nature, require no failure detectors, and are fully asynchronous.

We have presented the first such randomised consensus protocol specifically designed for mobile ad-hoc networks, and studied its performance through simulations. The base of the protocol is Ezhilchelvan, Mostefaoui and Raynal's multi-value consensus protocol, but we have added 3 important optimisations:

Replenish optimisation Alleviates the effect on the number of rounds required to reach a decision incurred by increasing the number of distinct initial proposals. This optimisation works by emptying the local bag of values from which each node makes a random choice at the end of each round, and replenishing the bag during each round with only the values received in that round.

Adopt optimisation Further reduces the average number of rounds by having slower nodes adopt the choices made by faster ones. This optimisation takes advantage of the highly variable network latencies which occur in mobile ad-hoc networks.

Cross layer optimisation Reduces the message passing overhead without impacting the number of rounds or the decision latency. This optimisation works by integrating the optimised consensus protocol and the reliable dissemination primitive presented in chapter 3.

The first two optimisations combine to reduce the average number of rounds required to reach a decision by several orders of magnitude. The observed average number of rounds required is between 2 and 4 rounds, irrespective of the number of nodes in the system, the number of distinct initial proposals made, or even the number of failures tolerated. The third, cross layer optimisation reduces the per round transmission overhead by between 1 and 2 orders of magnitude, requiring O(1) instead of O(n) invocations of the reliable dissemination primitive per round. A further benefit of combining the cross layer and adopt optimisations is that the protocol can now (again) tolerate up to $f < n/2$ instead of just $f < n/4$ failures.

Just as importantly, we provided a **proof of correctness**. Crucially, this proof allows the randomised protocol to work despite the fact that only $k = n - f$ nodes could be guaranteed to

receive any given message. The original proof requires a reliable broadcast of the initial proposals made by nodes to reach *all* correct nodes, something we showed to be impossible in theorem 2.3. The new proof shows how the replenish optimisation makes this requirement redundant.

As a result, we now have a consensus solution suitable for ad-hoc networks; the solution reaches a decision fast and has a low transmission overhead. In the next chapter we will see how this consensus protocol can be combined with the reliable dissemination primitive described in the previous chapter to design a fault-tolerant middleware solution for ad-hoc networks.

Chapter 5

A design for a fault-tolerant tuple space for mobile ad-hoc networks

5.1 Introduction

Linda[Gel85] is a simple, yet powerful *coordination language* which can be used to coordinate the actions of distributed entities. There have been numerous attempts to bring the Linda model to ad-hoc networks, as Linda makes implementing sophisticated distributed collaborative applications and services much easier. However, all attempts so far have either weakened the semantics of the Linda primitives or provided no fault-tolerance. This removes much of the benefit of the Linda model in a number of interesting scenarios.

In this chapter we will argue that the reason for this weakening of semantics or lack of fault-tolerance is because prior to the work described in this dissertation (and the ongoing work by Friedman et al.[FT03]) no consensus solution suitable for mobile ad-hoc networks existed. However, without a consensus solution, the replication required to provide fault-tolerance cannot be achieved while still maintaining the original Linda semantics.

This chapter will describe the design of a fault-tolerant Linda system which maintains the original Linda semantics. In particular, we will show how the two protocols introduced in previous chapters can be used to implement the system. As these two protocols have been shown to perform well through extensive simulations, we can reasonably conclude that the design, if implemented, would render a system suitable for mobile ad-hoc networks.

The availability of suitable reliable dissemination and consensus protocols does not make the design of such a system trivial. A number of important design issues remain and the central contribution of this chapter lies in carefully addressing these.

For example, the result in theorem 2.3 implies that any decision reached by the group as a whole cannot be guaranteed to be received by all nodes in the group. This implies that a node cannot automatically assume that any node it communicates with has the same view of the state of the

system as it. This has far reaching implications.

This problem is exacerbated when nodes are allowed to join and leave the group. The design must ensure that sufficient replication is maintained when nodes leave, and that joining nodes are given enough information about the state of the system in order to be able to participate fully in the group without violating Linda semantics. This last issue is closely related to the classical "state transfer" problem in fault-tolerant wired network systems (see for example chapter 18, section 3.2 in [Bir05])

Almost the most trivial contribution of this chapter is to show how to assign node identifiers upon nodes joining, showing how an assumption that has been made throughout this dissertation can be met; that is, how all nodes can be assigned node identifiers which are sequentially rising integers from 0 to n.

5.1.1 Linda in a nutshell

Linda was originally developed as a means for programmers to coordinate multiple execution threads to create parallel programs on multiprocessors. The Linda model provides a globally accessible *tuple space*, through which independent and possibly heterogeneous processes communicate by means of adding, reading and removing *tuples*.

A tuple is a sequence of typed fields, such as `<''foo'', 23, 10, 1978>`, containing the information to be communicated. A tuple is anonymous, and may exist independently of the process which created it. Tuples are referenced associatively using tuple *templates* or *patterns*. A tuple pattern can contain *actuals* and/or *formals*. Actuals are typed values; all the fields in the previous tuple are actuals. Formals are typed "wild cards"; the tuple pattern `<''foo'', ?integer, ?integer, ?integer>` contains one actual (the string "foo") and 3 formals of type integer. A tuple pattern *matches* a tuple if each actual field of the pattern equals those of the tuple, and the types of the formals in the pattern match the types of the actuals in the tuple. Therefore the tuple pattern `<''foo'', ?integer, ?integer, ?integer>` matches the tuple `<''foo'', 23, 10, 1978>`.

The original Linda model defined 4 primitives available to the processes using the tuple space:

`out(t)` takes a tuple, `t`, and places it into the tuple space, returning immediately.

`in(p)` takes a tuple pattern, `p`, and if it matches a tuple in the tuple space, the matching tuple is removed from the tuple space and returned to the calling process. If there is no matching tuple available, this operation will block. This operation is *atomic*, i.e. a node which gets returned a tuple can be sure that no other node will be returned the same tuple.

`read(p)` takes a tuple pattern, `p`, and if it matches a tuple in the tuple space, a copy of the matching tuple is returned to the calling process. The matching tuple, if any, remains in the tuple space.

`eval(a)` is Linda's process creation mechanism. This functionality is orthogonal to the communication primitives which is the focus of this work, so this primitive is ignored.

If two or more tuples match a given tuple pattern, which tuple is returned as the result of a `read(p)` or an `in(p)` is undefined. Similarly, if two operations are invoked in parallel on the tuple space, it is undefined which of the processes gets executed first. For example, if a `read(p)` and an `in(p)` matching the same (and only) tuple in the tuple space are executed in parallel, it is undefined whether the `read(p)` will block or not.

Linda provides a simple and powerful way to coordinate parallel processes. The authors of Linda provided as an example an implementation of the classic "Dining Philosophers Problem" implemented in a Linda system developed for the C programming Language. Briefly, the Dining Philosophers Problem involves a number of philosophers, usually 5, seated around a circular table. In front of each one is a bowl of rice. Between each philosopher there is a chopstick. It takes two chopsticks to eat rice, so while philosopher n is eating neither philosopher $n+1$ nor philosopher $n-1$ can be eating. The problem arises if each philosopher starts by picking his left chopstick, thus causing deadlock (all philosophers have a chopstick each). Assuming there are 5 philosophers, a typical solution to this problem is to admit only 4 at a time that try to eat. Then in the worst case at least 1 philosopher can always eat when the other 3 are holding 1 chopstick.

```
1   phil(int i)
2   {
3      while(1) {
4         think();
5         in(''room ticket'');
6         in(''chopstick'', i);
7         in(''chopstick'', (i+1)%Num);
8         eat();
9         out(''chopstick'', i);
10        out(''chopstick'', (i+1)%Num);
11        out(''room ticket'');
12     }
13  }
14
15  initialise()
16  {
17     int i;
18     for(i = 0; i < Num; i++){
19        out(''chopstick'', i);
20        eval(phil(i));
21        if ( i < (Num-1))
22                out(''room ticket'');
23     }
24  }
```

Figure 5.1: C-Linda implementation of the Dining Philosophers Problem.

The C-Linda implementation of this solution is reproduced from [CG89] in its entirety in figure 5.1. Num chopstick tuples and $Num-1$ room ticket tuples are put into the tuple space by the initiating process on line 19 and 22. In addition, Num philosopher processes are spawned (line

20). A philosopher will first attempt to get a room ticket, then pick up the left chopstick, then the right, followed by eating and then putting down the chopsticks and releasing the room ticket (lines 5 - 11). As the authors note, there is no fairness guaranteed in this solution (because of the non-deterministic choice of which parallel process gets returned a tuple during parallel `in(p)`'s), but assuming the implementation is reasonably fair, no philosopher will starve. A solution to the same problem implemented in the then state of the art parallel programming language Parlog86, required in excess of 70 lines of code, as well as 5 diagrams to explain the code structure[Rin88].

The simplicity of the implementation arises from two of the properties provided by the Linda model:

Space decoupling Two processes communicating through the tuple space do not need to know each other's identity, as tuples are referenced associatively. This allows processes to be fully decoupled in space, enabling for example reconfiguring an application by changing the processes which implement it; the parts of the application not being modified can be left unchanged.

Time decoupling A tuple added to the tuple space by `out()` remains in the tuple space until it is removed by an `in()`. If it is never removed by an `in()` then in principle it will remain in the tuple space forever. This allows processes to be distributed in time; that is, a process A can generate some tuples and terminate, followed by another process B at some later time consuming the generated tuples.

These two properties combined implies that a single tuple is functionally equivalent to a semaphore [Dij68]; `in(sem)` is functionally equivalent to the semaphore operation `P(sem)` and `out(sem)` is functionally equivalent to `V(sem)`. Further, these are in fact *distributed semaphores*[Sch80], that is semaphores defined over an arbitrary number of disjoint processes[Gel85], which are very powerful primitives as we shall see.

5.1.2 Why Linda in mobile ad-hoc networks?

The Linda model has been adopted for the distributed systems setting, with Linda-like systems being developed both in academia and industry. The features provided by the Linda model match well with the requirements of distributed systems, as often the problems facing programmers of physically distributed applications and parallel programs are similar. Prominent examples from industry include TSpaces from IBM[WML98] and JavaSpaces from Sun Microsystems[FAH99]. The EventHeap[JFW02] developed at Stanford as part of their "iRoom" pervasive computing environment is a good example from academia. However, these systems are implemented as centralised tuple spaces on a single server accessible by remote clients. Such implementations are unsuitable to ad-hoc networks, where network connectivity is highly variable and the fault tolerance and availability of a single server is by no means guaranteed.

Distributed implementations of Linda providing fault tolerance[BS95] and data availability [Pin93] have been proposed. The main disadvantage with these approaches is that they were designed with relatively stable network topologies in mind, and thus assume a high degree of network connectivity. These approaches are thus unsuitable in ad-hoc networks for the same reasons that the unicast and multicast protocols designed for wired networks are not well suited to the ad-hoc environment; in ad-hoc networks connectivity can be patchy and topology highly dynamic.

None the less, if Linda semantics (or something equivalent) could be provided despite the obstacles presented by the ad-hoc environment, a number of issues currently facing developers of sophisticated, fault-tolerant applications and services using ad-hoc networks would become much more manageable. For example, researchers working on multi-robot systems have identified coordinating the robots' actions as being important:

> The key to utilizing the potential of multi-robot systems is coordination. How can we achieve coordination in systems composed of failure-prone autonomous robots operating in noisy, dynamic environments?[GM02]

The "coordination" problem being alluded to is the so called "Dynamic Multi-Robot Task Allocation" [GM01] problem; which can be stated as follows: There are a number of robots, each looking for one task, and a number of tasks, each requiring exactly one worker. Clearly there are other issues besides communication inherent in this problem; optimal assignment of tasks to robots being one. However, even just coordinating the robots such that exactly one robot gets assigned each task in such a "failure-prone ... noisy, dynamic environment" is a challenge; one which would be greatly reduced by the availability of something like a fault-tolerant version of Linda running over a mobile ad-hoc network.

Another example is dynamic address assignment in mobile ad-hoc networks. Briefly this is the problem of assigning a unique address (such as an IPv4 address) to nodes joining the network. A centralised service, such as that provided by a DHCP server in LAN environments is unusable in ad-hoc networks; a decentralised, replicated service is deemed necessary to provide availability. The difficulty arises in ensuring that no duplicate addresses are assigned. There are two types of solution to this problem; schemes which guarantee that a duplicate IP address will never be assigned, and schemes which may supply duplicates, but eventually detect and handle this. Descriptions and a study of the relative performance of a number of these approaches can be found in [SB04].

Using a fault-tolerant implementation of Linda on ad-hoc networks, the solution is trivial; available IP addresses is simply added to the tuple space using something like `out(''IPv4 address'', 128.134.23.74)`. The addresses are then stored and duplicated transparently on the nodes implementing the tuple space; no separate address assignment protocol is necessary. To be assigned a unique IP address a node simply issues an `in(''IPv4 address'', ?IPv4Address)`.

5.2 Previous efforts to bring Linda to ad-hoc networks

A number of middleware systems for ad-hoc networks inspired by the Linda model has recently been proposed; e.g. LIME (Linda in a Mobile Environment)[PMR99],Limbo[DWF97], Tuple Board[KB05] and RUSSIAN (Resilient and Unified Shared Spaces in Ad-hoc Networks)[CQ04]. However, none of the proposed systems have provided "strict" Linda semantics and still tolerated crash failures; all have either weakened or modified the properties slightly, or do not tolerate crash failures at all.

The easiest approach is to weaken the primitives themselves. An example is RUSSIAN where a tuple is replicated as widely as possible providing some fault-tolerance, but the `in(p)` operation is only "best effort". This means tuple removal is no longer atomic, and the same tuple can be returned as the result of multiple `in(p)` operations. The drawbacks of this approach becomes clear in scenarios where atomic `in(p)` is necessary, though the authors argue that a number of interesting applications can built using this abstraction.

A more common modification is to introduce some form of "tuple ownership". Limbo, LIME and Tuple Board are examples of this type of modification, where a tuple is associated with a specific owner node. The idea is to maintain Linda semantics by only allowing the current owner of a tuple to withdraw it. Other nodes can get a copy of the tuple using `read(p)`, but not `in(p)`. The Tuple Board simply assigns ownership based on which node inserted the tuple, while Limbo has the ability to transfer ownership of a tuple upon request and thus can allow other nodes to withdraw it. The LIME model only allows `in(p)` operations on tuples which are local to the host, but extends the Linda semantics with the ability to execute operations on remote nodes when nodes are connected.

Linking the existence of tuples and semantics of operations to specific owner nodes in this way is undesirable in a number of scenarios as space decoupling is lost (the owner node must be identified). Another downside to this approach is the lack of availability which arises when a node crashes; even if a tuple is replicated on more than one node, if the current owner of a tuple crashes, it cannot be consumed by an `in(p)`.

Consider the examples of assigning IP addresses given in the previous section; which nodes should "own" the pool of IP addresses? Why should a whole batch of IP addresses become unusable simply because the node which happened to own these addresses crashes? The multi-robot example is another case in point; the necessity for a task to be assigned may not go away simply because the robot which deposited the task description has crashed. In extreme cases it may be even more pertinent that the task is assigned *because* it has crashed. The robot may have detected a fire and deposited a request for a robot with fire-fighting capability to deal with the fire. The fact that the robot which detected the fire now has failed should certainly not stop the task being assigned.

The reason these Linda implementations weaken semantics is that tuples must be replicated to tolerate crash failures. Implementing strict Linda semantics when tuples are replicated is a difficult

task. The main problem lies in ensuring the `in(p)` operation remains atomic when a tuple is replicated onto several nodes. Guaranteeing only one node is returned a given tuple as a result of an `in(p)` requires consensus between the nodes implementing the distributed tuple space about which process gets the tuple. As argued in chapter 4, prior to the work by Friedman et. al and the work in this dissertation, low bandwidth consensus solutions suitable for ad-hoc networks did not exist.

5.3 Design overview

The tuple space design was driven by the following objectives:

Availability : The tuple space should be able to tolerate up to f crash failures and be completely distributed; the system should not have a single point of failure. This requires all tuples to be replicated on at least $f + 1$ physically distinct nodes

Consistency : The semantics of tuple space operations `in(p)`, `out(t)` and `read(p)` should be the same as if there was a single tuple space; the tuple space should provide "strict" Linda semantics. Achieving this when tuples are replicated requires consensus between the replica holders.

A system adhering to these design goals would remove the shortcomings outlined in the previous section, thus providing the full benefit of the Linda model to the applications which require it. Of course, not all applications require the fault-tolerance and strict semantics that a fault tolerant tuple space provides, and it is perfectly feasible to run something like LIME or RUSSIAN alongside it, using the stronger semantics only when required.

The next section describes how consistency and availability is achieved in the case when the group membership of the nodes implementing the tuple space is *static*. That is, nodes do not join or leave the group implementing the tuple space, though crashes may of course still occur.

A static group is useful in a number of scenarios. For example, in the case of a group of robots cooperating there is no requirement to handle joins and leaves; the group is determined at the start of the mission. However in certain situations, being able to have a dynamic group membership may be beneficial. How this can be handled is described in detail in section 5.5.

5.4 A fault-tolerant tuple space with static group membership

A distributed, fault-tolerant tuple space implementation needs to address 3 core issues; how to write tuples to replicas, how to read tuples from replicas and how to remove tuples from replicas.

The next section describes how tuples can be written to replicas using `out(t)` and read from replicas using `read(p)`. The reliable dissemination protocol described in chapter 3 is used in the former, while a reliable *aggregation protocol* is used to achieve the latter.

The subsequent section describes how tuples can be removed from replicas using `in(p)`, including the modifications required to the `read(p)` operation to maintain consistency once this capability is added.

5.4.1 Reading from and writing to replicas

An `out(t)` operation requires replicating the tuple, t, onto a number of nodes. The number of replicas of t must be greater than f, and numerous enough so that a subsequent matching `read(p)` operation is guaranteed to find a copy of t. This can be achieved by replicating the tuple onto a *write quorum, WQ*. A *quorum* is a subset of nodes such that any two quorums in a *quorum system* intersects. A *read-write quorum system* is a quorum system with two kinds of quorums; write and *read quorums, RQ*, any two of which intersects. More background on quorum systems can be found in [Maled].

Figure 5.2 shows pseudo-code for the `out(t)` operation. Once the `out(t)` operation is invoked,

```
1  out(Tuple t)
2  {
3          t.uid = generateUid();
4          TupleStore.put(t.uid, t);
5          disseminate.write(t.uid, |WQ|, blockUntilrealisation=FALSE); //non-blocking
6  }
7
```

Figure 5.2: The `out(t)` operation with static group and no destructive reads.

a unique identifier, *uid*, is generated and tagged to the tuple, and the tuple is added to the node's local *Tuple Store*. The Tuple Store contains all the tuples a node has received. The tuple is then disseminated to at least $|WQ|$ nodes using the reliable dissemination protocol (line 5). This invocation of the dissemination protocol is non-blocking (as indicated by the boolean value passed to the protocol), as when the message is realised is immaterial to the node performing the `out(t)`; the Linda semantics require that a call to the `out(t)` operation returns immediately. Also note how the dissemination protocol is only passed a reference (the *uid*) of the message. The *uid* is used by the dissemination protocol to query the Tuple Store immediately before it transmits the packet, thus avoiding duplicating the tuple. A node receiving a tuple adds it to its local Tuple Store (not shown).

In order to perform a `read(p)` operation, an *aggregation protocol* is required. An aggregation protocol queries at least k nodes for a match to a *template*. A template is in this case the same as a tuple pattern, but could in principle be anything. The reliable dissemination protocol can act as an

aggregation protocol through some fairly straight forward modifications. The resulting protocol is similar to the first phase of the optimised consensus protocol described in section 4.4.4. The main differences between the aggregation and the dissemination protocols are as follows:

1. The aggregation message contains the template and a placeholder for a match to the template if one has been found.

2. On reception, if the placeholder is empty, the template is passed up to the higher layer and if a match to the template is found, is returned to the aggregation protocol and added to the placeholder.

3. The node(s) which realise the message adds the match contained in the placeholder (if any) to the realisation packet.

4. As with the cross layer optimisation for the randomised consensus protocol, the push-pull optimisation is disabled (see section 4.4.3).

When the aggregation message is realised, at least k nodes will have checked for a match to the template. If a match exists, it will be contained in the aggregation message (by 1 and 2 above). If no match exists, no further action is necessary. If a match is found, this match should be returned to the node which initiated the aggregation.

There are two ways in which the match could be returned to the originator; either make it the responsibility of the node which initiated the aggregation, or put the onus on the node(s) that realised the aggregation message. The former was chosen as putting the onus on the realising node(s) has a number of drawbacks; the main problem is that there is no way a realising node can know if the originator of the aggregation has since crashed. This implies that any technique to inform the originator of a match has to be best-effort only, as an attempt to reliably deliver a message to a crashed node will never terminate. For this reason, the approach outlined by modification 3 above was chosen.

It is worth noting a couple of things about this protocol. First, it could be optimised by using only the *uid* of any match found, in a similar way to the push-pull optimisation in section 3.3.1. Second, it could easily be modified to return *all* matches instead of only one.

A `read(p)` operation uses the aggregation protocol to query a read quorum as shown in figure 5.3. The `read(p)` operation blocks until a match is found, so if a call to the aggregation protocol returns a null result indicating no match has been found (line 9), the thread sleeps for *readInterval* seconds after which the condition in the while loop still holds (line 5) and the aggregation protocol is called again. The *readInterval* should be configurable. If there are many tuples that match a given tuple pattern received by the aggregation protocol, one can be chosen at random from the Tuple Store.

```
1
2  read(tuple-pattern p)
3  {
4          resultTuple = null;
5          while(resultTuple == null)
6          {
7                  initiated = Now();
8                  p.uid = generateUid();
9                  resultTuple = aggregate.read(p.uid, p, |RQ|); //blocking
10                 sleep(readInterval - (Now() - initiated));
11         }
12         return resultTuple;
13 }
14
15
```

Figure 5.3: The `read(p)` operation with static group and no destructive reads.

5.4.2 Removing items from replicas

The reading and writing of tuples from and to replicas using `read(p)` and `out(t)` can be achieved by the means described in the previous section. However, if the tuple space is to support the `in(p)` operation, the following two issues arise:

- The `in(p)` operation is *atomic*. That is, the same tuple cannot be returned as the result of (consumed by) two different `in(p)` operations.
- The semantics of a Linda `read(p)` operation does not allow a tuple to be returned which has previously been consumed by an `in(p)`.

How to guarantee that the `in(p)` operation remains atomic even though tuples are replicated is covered in the next section. Section 5.4.2.2 explains why Linda semantics do not allow `read(p)`s to return tuples which have been consumed by an `in(p)`, and presents the modifications required to ensure such semantics.

5.4.2.1 Ensuring tuple consumption using `in(p)` is atomic

The consensus protocol tailored for ad-hoc networks presented in chapter 4 can be used to guarantee the atomic nature of `in(p)`. This is done by requiring a node to obtain agreement that a tuple can be consumed prior to returning the tuple to the calling application. Obtaining such agreement necessitates that the node is aware of all other decisions made up to the point where it proposes to consume the tuple, as otherwise it may make a proposal which contradicts an earlier decision.

When the consensus protocol is used to make multiple decisions it can be thought of as working in *epochs*. Epochs are consecutive, monotonically increasing integers representing the order in which decisions were made. Each decision has an associated epoch number (which should not be confused with round numbers used in each execution of the consensus protocol).

The state of a consensus protocol includes the decisions reached in all epochs up to and including the current epoch. If a node wants a decision on something new, it needs to initiate the consensus protocol in the "correct" epoch, where the correct epoch is the epoch which is one higher than the latest decision made in the system.

The pseudo code in figure 5.4 shows how the consensus protocol can be used in conjunction with the `read(p)` operation described in previous section to yield the `in(p)` functionality.

```
1  in(tuple-pattern p)
2  {
3          candidateTuple = null;
4          matchingTuple = null;
5          while(matchingTuple == null)
6          {
7                  if(candidateTuple == null)
8                  {
9                          candidateTuple = read(p); //blocking
10                         TupleStore.put(candidateTuple.uid, candidateTuple);
11                 }
12                 decision = consensus.propose(''I get candidateTuple'');
13                 if(decision implies candidateTuple no longer available)
14                         candidateTuple = null;
15                 else if(decision is agreed)
16                         matchingTuple = candidateTuple;
17         }
18         disseminate.write(decision, |WQ|, blockUntilrealisation=TRUE); //blocking
19         return matchingTuple;
20 }
```

Figure 5.4: The `in(p)` operation with static group.

A node performing an `in(p)` operation enters a loop (line 5) which it returns from only when a suitable tuple has been found and agreement that it can consume it has been reached. The node first issues a `read(p)` to get a candidate tuple (line 9). Once this has been obtained, it is stored in the Tuple Store and the node initiates a consensus in what it believes to be the correct epoch proposing that it gets to consume the candidate tuple (line 12). There are three possible decisions reached when the consensus returns; (i) the candidate tuple is no longer available, (ii) an unrelated decision has been reached or (iii) agreement is reached that the node can consume the tuple.

The first of these possibilities, the candidate tuple no longer being available, occurs when another node has been given the right to consume the tuple. In that case, the tuple should perform another `read(p)` and initiate another consensus on the new candidate tuple. This is achieved by resetting the candidate tuple to null on line 14.

The second possibility arises because *all* decisions have to be reached in order. For this reason a completely unrelated decision could have been made in the current epoch. For example, another node could have proposed that it should get to consume some other tuple. When this happens the node does not need to do another `read(p)`, as the candidate tuple is still valid; it simply makes the same proposal in the new epoch by looping around with the same candidate tuple and making a new proposal on line 12.

Finally, once an agreement that a node can consume a given tuple is reached, the node which makes this decision needs to disseminate it to a write quorum. The originating node must block on the invocation of this dissemination until the decision message has been realised. This is achieved by making a blocking call to the dissemination protocol on line 18. As other nodes receive this decision, they can delete the tuple from their Tuple Store. Once the decision message has been realised, the originating node can return the matching tuple to the calling process (line 19).

A node must block and not return the match until it knows the decision message has been realised, as not waiting may result in the tuple (which has now been returned by an `in(p)`) being returned as the result of a *later* `read(p)`. This can arise because there are no guarantees made about the relative speed at which dissemination and aggregation invocations are performed. This means nodes in subsequent read quorums may not be aware of the decision that the tuple has been consumed and thus erroneously return the consumed tuple.

The next section describes why returning a consumed tuple as result of a `read(p)` is incorrect, and outlines the functionality (in addition to the blocking dissemination of consensus decisions described above) required to ensure the correct semantics.

5.4.2.2 Guaranteeing `read(p)`s do not return consumed tuples

Returning a tuple as the result of a `read(p)` which has previously been consumed by an `in(p)` is wrong because the node which originally consumed the tuple may have performed a subsequent `out(t)` based on the contents of the first tuple. A node performing another `read(p)` may now be returned either of the two tuples, which may be incorrect.

The following example, adapted from [XL89], shows why this is wrong when two nodes, A and B, are using a tuple space. Assume the variables u, v and x have previously been declared as integer variables in A and B's programs.

Node A	Node B
`in(''count'', ?x)`	`read(''count'', ?u)`
`out(''count'', x+1)`	`read(''count'', ?v)`

In this example, the integer value in the second field of tuples with the name "count" increases with time. Node A modifies the count in a manner which satisfies this constraint; node B only reads the count and should not observe a violation of the constraint.

The operations in this example could easily be executed in the following order: Node A's `in(p)` returns `<''count'', 1>`, node A then increments and performs the `out(t)` adding `<''count'', 2>` to the tuple space. If a tuple which has previously been `in(p)`'ed is allowed to be returned as the result of a subsequent `read(p)`, node B's first `read(p)` could return `<''count'', 2>` and its second `<''count'', 1>`, which is incorrect. Removing tuples from replicas thus requires a consensus

protocol to reach agreement on what nodes gets which tuples, as well a way of synchronizing `read(p)`s with completed `in(p)`s.

Making the node which has obtained agreement that it can consume a tuple wait to return the tuple until after the realisation of the decision message is not enough to avoid the above situation. This is because not all nodes which participate in subsequent `read(p)`s can be guaranteed to have received the decision that the tuple has been consumed. Recall that theorem 2.3 dictates that the maximum value for $k = (n - f)$ (section 2.3.2).

The following example shows how the incorrect semantics can come about: Assume the tuple space is being implemented by 5 nodes: A, B, C, D and E. Assume also that $f = 2$ (thus max $k = 3$) and for simplicity that $|WQ| = |RQ| = majority = 3$. Consider if a tuple Z which has been replicated onto all three nodes is consumed by node A in epoch 1. Assume that a write quorum consisting of nodes A , B and C are aware of this and are now in epoch 2, and that nodes D and E remain in epoch 1 and still keep a copy of Z. If node D now initiates a `read(p)` which happens to match the consumed tuple Z, it may happen that node E is the last node to be queried about a match. Node E could then add in Z as the matching tuple and inform D. D would then return the previously consumed tuple, which is incorrect.

However, the reliable dissemination protocol *is* able to guarantee that at least one of the nodes which participate in any possible later `read(p)` will have received any given decision. This is because of the intersection property of read and write quorums; the decisions that a tuple has been consumed must be disseminated to at least a $|WQ|$ nodes, and a `read(p)` must query at least $|RQ|$ nodes using the aggregation protocol.

We can take advantage of this property to ensure that by the time any `read(p)` returns it will not contain a tuple that has been consumed. This is achieved by requiring any node participating in a `read(p)` to tag all messages with what it believes to be the most recent epoch. Nodes can then compare the epoch numbers they receive, request any decisions they are missing and take the appropriate action; this guarantees that any `read(p)` will be performed in the correct epoch by the time it completes.

Ensuring that all `read(p)`'s are performed in the correct epoch can be handled transparently by a *State Manager*. A State Manager maintains the cache of all decisions reached by the consensus protocol so far (the *Decision Cache*), as well as all group related variables, such as the values for *current_epoch* n, f, $|WQ|$, etc. The State Manager sits between the group communication protocols and the network and tags all outgoing packets with what it believes to be the correct epoch (the *current_epoch*). It also intercepts all incoming messages before they are passed up to the protocol.

In its most basic form, the State Manager performs the following actions when it receives an incoming packet tagged with the epoch *incoming_epoch*:

incoming_epoch < current_epoch: Transmit the decision *incoming_epoch* + 1 contained in the

Decision Cache. Discard the packet.

incoming_epoch > current_epoch and the packet is *not* a decision: Transmit a request for decision *current_epoch* + 1. Discard the packet.

incoming_epoch > current_epoch and the packet is a decision: If the decision is for *current_epoch* + 1, add the decision to the Decision cache and terminate the ongoing aggregation and consensus. If the decision is for > *current_epoch* + 1, cache the decision (for later, in order, processing) and request decision *current_epoch* + 1.

incoming_epoch = current_epoch: If the packet is a decision, discard the packet. Else pass the packet to the correct protocol.

With the introduction of a State Manager, the aggregation protocol (and thus the `read(p)` operation) can no longer return tuples which have previously been `in(p)`'ed. From the example above, node E would have been aware of the epoch mismatch once it got queried, and would have requested its "missing decisions". On reception of these the tuple Z would have been deleted from E, and the correct behavior would have been ensured.

It is worth noting that epoch mismatch can be handled for the dissemination protocol as well. We will make use of this ability in the next section.

5.5 Handling dynamic group membership

In order to handle nodes joining and leaving the group which implements the tuple space, two main issues must be addressed: Transferring a consistent state to a joining node and ensuring the replication of tuples is sufficient upon nodes joining or leaving. We will deal with these issues in turn.

5.5.1 State transfer to joining nodes

The state transfer problem is a classic problem in reliable distributed computing, and is explicitly dealt with in for example the Virtual Synchrony[BJ87] or Paxos[Lam98] models. The crux of the problem is that a joining node needs to be told of past events in order to be able to participate in future group related decision making. For example, in our design, if a node is allowed to join without being given the past decisions made by the consensus protocol (the *Decision Cache*), it may propose or agree with proposals that contradict earlier decisions. This can easily lead to inconsistencies.

Assume a node wishing to join can pull the state from some current group member. While an important step, this is not the end of it; there is the real possibility that after the state has been transfered, but before the node has been admitted to the group properly, further updates are made to

the group state. If the joining node simply uses the stale state it received some time ago, the updates which happened just before the node joined may be lost. This can also lead to inconsistencies.

In general there are three broad approaches to this problem. The first and most heavy handed approach is to lock down the whole group when a node wants to join, transfer the state to the joining node and then unlock the group and resume group operations. The second approach involves a joining node receiving a checkpoint of the state offline (before the node joins the group) and then joins. The joining node then reconciles any missing updates, doing something like: "I have the state until checkpoint 323"; "This is what happened since then...", etc. The third and final approach is to attempt to do as much as possible concurrently. The Horus[RBM96] system for example has a 5 stage process for servers joining, involving initially joining as a client, then requesting to become a server, being installed as a special kind of server with "all rights, but no obligations", requesting the state and finally becoming a "normal" server. The general gist is to gradually allow the server to do more and more as it becomes more capable (see [RBM96] for further details).

The chosen design for managing node joins is depicted in figure 5.5. It is very similar to the second approach in that the state transfer is first done offline, followed by a subsequent update of any decisions reached after the offline state transfer. We assume the joining node has some way of getting various configuration information relating to the relevant communication channel used (e.g. multicast address and port number or similar). A joining node then pulls the state (on line 5) using a modified form of the aggregation protocol; the *getState()* function acts as a normal aggregation except it is handled by the State Manager and is not related to the Tuple Store in any way. The State Manager updates the epoch as it would any other aggregation message and adds the relevant state. This state includes: the Decision Cache, n, $|WQ|$, $|RQ|$, f, the current epoch number and a *Member Cache*. The Member Cache is a list of group node ids and the binding between these node ids and the underlying (also unique) address (e.g. a MAC address) of already joined group members. The joining node initiates its state to the received state as seen on lines 7-12.

After the joining node has pulled the recent state from the read quorum, it uses this state to initiate a consensus on line 24 proposing "Node X joins as Z", where X is the node's unique address and Z is an available group node id, *nid* from the Member Cache. During this consensus invocation, as with any invocation of the consensus protocol, if the joining node is in too low an epoch, the State Manager is used to give it any decisions made between it pulling the state while offline and now. Note that the node has not yet been assigned a *nid*, so cannot participate in the consensus, though it can initiate it and find out about its outcome. From the time a node first proposes to join, we consider it a part of the group for failure purposes (i.e. if the joining node crashes after this, only $f - 1$ other nodes in the group may crash).

If there are available *nids* in the group, the node will eventually be allowed to join. If there is no room, the group membership should be expanded, a topic which is not covered by the current

```
1
2   join(Address addr)
3   {
4           //Get the current state while offline
5           state = aggregate.getState(addr); //blocking.  addr used as uid.
6           //Assign the states
7           decisionCache = state.getDecisionCache();
8           n = state.getN();
9           epoch = state.getEpoch();
10          |RQ| = state.getRQ();
11          |WQ| = state.getWQ()
12          memberCache = state.getMemberCache();
13
14          canParticipateInReads = false;
15
16          //Get a candidate tuple
17          candidateNodeId = null;
18          assignedNodeId = null;
19
20          while(assignedNodeId == null)
21          {
22                  if(candidateNodeId == null)
23                          candidateNodeId = memberCache.pickFree();  //assume one exists..
24                     decision = consensus.propose(''addr joins as candidateNodeId'');
25                  if(decision implies candidateNodeId no longer available)
26                          candidateNodeId = null;
27                  else if(decision is agreed)
28                          assignedNodeId= candidateNodeId;
29          }
30
31          n++;
32          |WQ|++
33
34          //Can now participate in out(t)s, and in consensus,  but not read(p)s
35
36          allTuples[] = aggregation.readAll(generateUid(), |RQ|); //blocking.
37          TS.add(allTuples[]); //discarding duplicates of tuples stored after joining.
38          canParticipateInReads = true;
39
40          //Is now a fully fledged group member
41  }
42
43
```

Figure 5.5: The `join(addr)` operation which increases write quorum size on success

design. Once the the decision that it can join has been received, the joiner will assign itself the assigned *nid* (line 28), and the joiner and other nodes which know of the decision can update n.

However, the other main issue now arises; simply increasing n by itself means that read and write quorums no longer necessarily intersect (they are necessarily fixed values, and not defined in terms of n).

5.5.2 Ensuring sufficient replication on nodes joining or leaving

The simplest technique for dealing with this lack of quorum intersection would be to increase the read quorum size, $|RQ|$, at the same time as n is increased. There are a number of drawbacks to this approach however. First, the overhead of performing a `read(p)` grows with the increase in read quorum size. More seriously, simply increasing $|RQ|$ implies that availability is not increased by a node joining (as tuple replication is not increased). Finally, if this approach was adopted and nodes are also allowed to leave, the following undesirable scenario may arise: No matter how many nodes join, no nodes can ever leave the group(!).

For these reasons the desirable and indeed intuitive alternative is for availability to increase upon another node joining the group. What is required is increasing the value of $|WQ|$ on receiving the decision about a node join. The joiner increases the value of $|WQ|$ on line 32.

This will work for tuples which are added to the tuple space by subsequent `out(t)` calls, as long as dissemination protocol calls are also synchronised within a given epoch. Such synchronization can easily be achieved using the State Manager as discussed previously. However, availability is not increased for tuples already in the tuple space. Indeed, because the *old* write quorum size is now insufficient to ensure intersection with the read quorum, these tuples may never be found by subsequent `read(p)` or `in(p)` calls. The remedy is to disallow joining nodes from participating in aggregations before a *readAll()* has been performed by the joining node. The boolean value *canParticipateInReads* is used by the joining node's State Manager for this purpose, and is initially set to false on line 14.

A readAll() is a modification to the aggregation protocol's read() as hinted at in the previous section; instead of simply adding *one* matching tuple, *all* tuples are added to the aggregation. Once the readAll() aggregation has returned, the joining node adds the tuples to its local Tuple Store (discarding duplicates) and can now participate in the aggregation protocol as a fully fledged group member (lines 36-38).

Managing nodes leaving the group is much simpler, the main issue is ensuring that $|WQ|$ is still greater than or equal $f+1$. For this reason the node must initiate a consensus proposing "I leave". If it gets a negative response (as it leaving would result in the write quorum being less than f), it cannot leave until a new node has joined the group[1]. When the decision that a node can leave is

[1]This also shows why increasing $|RQ|$ instead of $|WQ|$ on nodes joining is a very bad idea..

received, nodes will decrease n and $|WQ|$, and update the Member Cache appropriately.

5.6 Changing quorums sizes and number of failures tolerated

The strategy adopted in our design is to replicate all tuples as widely as possible, thus increasing availability and fault-tolerance and making read quorums as small as possible. This may not be suitable in all situations however, and the system should therefore have the ability to change the sizes of the read and write quorums within the allowable bounds. Further, it may be desirable to be able to vary the number of failures a system can tolerate. An example is if the tuple space is bootstrapped from a single node. In this scenario, the number of tolerable failures, f, must be zero initially, but should be able to increase as new nodes join the group.

These two issues are very similar; it is about how to reach agreement so that a system wide parameter can be changed a within some allowable bounds. The number of tolerable failures, f can not be set to higher than n or indeed $|WQ|$ or $|RQ|$. In the same way, you would not want to set either quorum size to be less than the tolerable number of failures, or higher than the number of nodes in the system. However, within these bounds, changing the values is fairly straightforward given that all dissemination, aggregation and consensus protocol messages are synchronised in the right epoch (using the State Manager as explained in previous sections).

For example, one might want to keep the tolerated number of failures, f, to roughly 10% of the number of nodes making up the group. In that case, one could employ a policy which, upon receiving a join decision implying that f was less than 10%, a node would initiate a consensus proposing "RQ is increased", after the success of which a node would propose "f is increased to RQ-1". Something similar could be done to change the write quorum size if that was desired.

5.7 Conclusion and summary

Previous attempts at bringing the Linda model to mobile ad-hoc networks have either had to weaken the original semantics of the Linda primitives, or have not been able to provide fault-tolerance. Weakening the semantics of Linda makes it less powerful, while not providing fault-tolerance makes it less usable in scenarios where failures are likely.

However, in order to keep the Linda `in(p)` primitive atomic while replicating tuples to provide fault-tolerance, a consensus solution is necessary. This explains why none of the previously proposed Linda systems for mobile ad-hoc networks were able to do so; prior to the work in this dissertation (and the parallel work by Friedman et al.[FT03]) consensus was considered too heavyweight in terms of transmission overheads to be feasible in mobile ad-hoc networks.

This chapter has shown in detail how a fault-tolerant Linda system which maintains the original

semantics of the Linda model could be implemented using the protocols presented in the previous two chapters. As extensive simulations have shown that both these protocols perform well in a wide range of network conditions, it is safe to assume that the system, if implemented, would be very well suited to ad-hoc networks; the resulting system would not only have a reasonable transmission overhead, but also be able to tolerate highly dynamic and frequently disconnected network conditions.

The design has addressed a number of issues in addition to keeping `in(p)` atomic including; how to maintain the semantics that a tuple which has already been removed by an `in(p)` cannot be returned by a later `read(p)`, how to ensure sufficient tuple replication upon nodes leaving, how to perform state transfer to nodes joining so that these can participate in group decision making, and how to vary the number of failures tolerated based on the current number of nodes in the group.

These issues are particularly challenging given that not every node in the system can be assumed to receive the most recent decision made by the group, even though a decision is reliably disseminated by the node which reaches it. This naturally arises from theorem 2.3.

This chapter has described how the issue which arises from theorem 2.3 can be resolved by tagging all messages with an epoch number, where an epoch number signifies the latest decision the transmitter of the message is aware of. This works because all decisions reached by the consensus protocol are assigned monotonically increasing integer identifiers. Nodes which receive a message with a higher epoch than it is currently in can thus directly request its missing decision(s).

Chapter 6

Summary and conclusions

This dissertation has introduced two families of reliable group communication protocols, one solving the reliable dissemination problem, the other the consensus problem. The protocols have been shown to work in even transiently disconnected, highly mobile ad-hoc networks, and extensive simulations confirm that the overheads associated with these are not too high to be feasible in ad-hoc networks. The dissertation has also presented a design for a fault-tolerant tuple space which is able to provide the same semantics as one would expect from such a system implemented on a single machine.

This chapter summarises the contributions made in this dissertation, and describes some avenues for future research.

6.1 Summary

In chapter 2, a system model for mobile ad-hoc networks was presented. The system model precisely defines the requirements on network connectivity and the assumption made that the resources available to any protocol are finite. The requirement on network connectivity was designed to be as minimal as possible, and essentially only requires the ad-hoc network not to be permanently partitioned. The assumption about finite available resources disallowed any protocol to, for example, require a node to keep a copy of a message for ever. This assumption was justified by the definition of mobile ad-hoc networks as resource constrained.

Chapter 3 presented a family of reliable dissemination protocols suitable for mobile ad-hoc networks. The protocols were shown to guarantee delivery to a minimum number of nodes in any ad-hoc network which meets the requirement on network connectivity. The protocols make use of a novel, distributed method of detecting when enough nodes have received a message and their design were guided by the foundational results presented in chapter 2. One of the protocols was shown to have transmission overheads on par with a simple, unreliable flooding protocol in relatively normal network conditions.

Chapter 4 presented a family of consensus protocols. The protocols are based on an existing

consensus protocol designed for wired networks, but include three important optimisations which remedies what make the existing protocol infeasible for mobile ad-hoc networks. The first optimisation removes the requirement that a message must be delivered to all nodes in the network, a requirement which was shown to be impossible to meet in chapter 2. The second optimisation takes advantage of the highly variable network latencies found in mobile ad-hoc networks to reduce, by several orders of magnitude, the number of rounds until a decision is reached . The third optimisation reduces the per round transmission overhead by between 1 and 2 orders of magnitude. The protocol which combines all these optimisations is shown to only require between 2 and 4 rounds to reach a decision, where, each round only requires O(1) invocations of the type of reliable dissemination protocol presented in chapter 3.

Finally, chapter 5 has demonstrated how these protocols can be combined to implement a fault-tolerant tuple space for mobile ad-hoc networks with the same semantics as one would expect from a tuple space implemented on a single machine. This design, if implemented, is the first tuple space system designed specifically for mobile ad-hoc networks which is able to both provide such semantics and also be fault tolerant.

6.2 Conclusion

This dissertation has demonstrated that reliable, fault-tolerant group communication protocols can be implemented in a wide range of mobile ad-hoc networks, and further that doing so does not incur too high transmission overheads to be feasible. It has done so by first defining a precise system model which a wide range of mobile ad-hoc networks fit into. Then, a family of reliable dissemination protocols and a family of consensus protocols have been introduced and proven correct in all ad-hoc networks covered by the system model. Finally, extensive simulations has been presented which demonstrate that these protocols do not incur excessive transmission overheads.

6.3 Future work

It has often been remarked that mobile ad-hoc networking research suffers from a lack of real world experimentation. This is a problem which afflicts the work in this dissertation as well. In general, any real world testing of the protocols is likely to throw up interesting and unexpected issues, and although the protocol's correctness will not suffer as long as the minimum requirements on connectivity hold, the protocols' parameters may require some tweaking to get the best possible performance.

For example, the maximum time between periodic transmissions of the reliable dissemination protocols presented in chapter 3 was fixed at ($\beta = 5s$) throughout this dissertation. Naturally, this

value is unlikely to be best value for all scenarios, and experimentation with this variable is likely to be needed in a real world deployment. (Note that actual studies of both human and animal movements are beginning to appear and may be of some use in this, see for example [HCSar] and [JOW02]).

More radical changes to the dissemination strategies of the reliable dissemination protocols is also possible (a protocol for when there are no failures is given in appendix A for example). Any such optimisation would still have to satisfy theorem 2.1 in that the resulting protocol would have to include executions where all nodes were actively participating in the dissemination of a message. This naturally lends itself to a two phase protocol; the first phase attempting to deliver the message in a more efficient manner (the optimised phase), and then reverting to one of the protocols described in this dissertation to guarantee delivery in less benign network scenarios (the basic phase). Proving termination of such an optimised protocol would thus be reduced to proving that if the protocol does not succeed, it reverts the basic phase.

Extending the reliable dissemination protocols presented in chapter 3 to the *multicast* case, that is, where there are a subset of "server" nodes to deliver a message to and the remainder of the nodes only aid in this task, is a worthwhile endeavour. It would extend the usability of the consensus and tuple space work to the case where there is a relatively small group of capable server nodes and a large number of less capable client nodes which connect to the server nodes. Such a heterogeneous system configuration is perhaps more common than the homogenous configuration assumed in this dissertation.

A careful implementation of the design of the tuple space from chapter 5 would be interesting and potentially useful. The main problem would be to decide on which platform to perform the work; in mobile ad-hoc networks there is no standardised hardware or software platform which everyone uses, unlike what the PC is for wired networks.

A more ambitious endeavour would be to build more a adaptive, intelligent network service using the protocols here as a base. Once wireless networking and wirelessly networked devices become ubiquitous, the ability of a group of devices to adapt to, learn from, and be aware of, their changing environment all the while maintaining fault-tolerance, is challenging. This is even more true if the devices which collaborate have different capabilities. A step in that direction may be to use the work here enhanced with some form of reinforcement learning algorithm, where for example the dissemination protocol "learns" when and how often to transmit a message.

Bibliography

[ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. "On Quiescent Reliable Communication." *SIAM J. Computing*, **29**(6):2040–2073, 2000. pages 21

[Ben83] Michael Ben-Or. "Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols." In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pp. 27–30, New York, NY, USA, 1983. ACM Press. pages 51, 53, 71

[BHR05] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. "JiST: An efficient approach to simulation using virtual machines." *Software Practice and Experience*, **35**(6):539–576, May 2005. pages 15, 16

[Bir05] Kenneth P. Birman. *Reliable Distributed Systems - Technologies, Web Services and Applications.* Springer-Verlag, 2005. pages 80

[BJ87] Kenneth P. Birman and T. Joseph. "Exploiting virtual synchrony in distributed systems." In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, pp. 123–138. ACM Press, 1987. pages 92

[BS95] David Edward Bakken and Richard D. Schlichting. "Supporting Fault-Tolerant Parallel Programming in Linda." *IEEE Transactions on Parallel and Distributed Systems*, **6**(3):287–302, 1995. pages 83

[BT85] Gabriel Bracha and Sam Toueg. "Asynchronous consensus and broadcast protocols." *Journal of the ACM*, **32**(4):824–840, 1985. pages 51

[Cer05] V. Cerf. "IETF Internet draft: Delay-Tolerant Network Architecture.", July 2005. pages 109

[CF99] Flaviu Cristian and Christof Fetzer. "The Timed Asynchronous Distributed System Model." *IEEE Trans. Parallel Distrib. Syst.*, **10**(6):642–657, 1999. pages 49

[CG89] Nicholas Carriero and David Gelernter. "Linda in context." *Communications of the ACM*, **32**(4):444–458, 1989. pages 81

[CQ04] Daniel Cutting and Aaron Quigley. "Resilient, Unified, Shared Spaces in Ad hoc Networks." In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware*, 2004. pages 84

[CRB01] R. Chandra, V. Ramasubramanian, and K. Birman. "Anonymous Gossip: Improving Multicast Reliability in Mobile Ad-Hoc Networks." In *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001. pages 2

[CSS02] David Cavin, Yoav Sasson, and André Schiper. "On the Accuracy of MANET Simulators." In *Proceedings of the Workshop on Principles of Mobile Computing (POMC'02)*, pp. 38–43. ACM, 2002. pages 15

[CT96] Tushar Deepak Chandra and Sam Toueg. "Unreliable failure detectors for reliable distributed systems." *Journal of the ACM*, **43**(2):225–267, 1996. pages 46, 47, 49, 50

[DDC97] C. Diot, W. Dabbous, and J. Crowcroft. "Multipoint communication: A survey of protocols, functions and mechanisms." *IEEE Journal of Selected Areas in Communications*, **15**(3):277–290, April 1997. pages 2

[Dij68] E.W. Dijkstra. "The Structure of the "THE" Multiprogramming System." *Communications of the ACM*, **11**(5):341–346, May 1968. pages 82

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the Presence of Partial Synchrony." *Journal of the ACM*, **35**(2):288–323, 1988. pages 49

[DWF97] N. Davies, S. Wade, A. Friday, and G. Blair. "Limbo: A tuple space based platform for adaptive mobile applications." In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms*, pp. 291–302, May 1997. pages 84

[EMR01] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal. "Randomized multivalued consensus." In *Proceedings of the 4th International Symposium on Object-Oriented Real-Time Computing*, pp. 195–200, 2001. pages 56

[EMS95] Paul Ezhilchelvan, Raimundo Macedo, and Santosh Shrivastava. "NewTOP: a fault-tolerant group communication protocol." In *Proceedings of 15th IEEE International Conference on Distributed Computing Systems*, pp. 296–306, 1995. pages 75

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *Javaspaces Principles, Patterns and Practice*. Pearson Education, June 1999. pages 82

[Fal03] Kevin Fall. "A Delay Tolerant Network Architecture for Challenged Internets." In *Proceedings of the annual conference of the Special Interest Group on Data Communication (SIGCOMM)*, pp. 27–34, Karlsruhe, Germany, 2003. pages 108

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." *J. ACM*, **32**(2):374–382, 1985. pages 49

[Fri03] Roy Friedman. "Fuzzy Group Membership." In *Future Directions in Distributed Computing: Research and Position Papers*, 2003. pages 19, 75

[FT03] Roy Friedman and Galya Tcharny. "Evaluating Failure Detection in Mobile Ad-Hoc Networks." *Technical Report, Computer Science Department, Technion*, **CS-2003**(06):0–22, October, 2003. pages 75, 76, 79, 96

[Gel85] David Gelernter. "Generative communication in Linda." *ACM Transactions on Programming Languages and Systems*, **7**(1):80–112, 1985. pages 79, 82

[GHM00] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. "Consensus in Asynchronous Distributed Systems: A Concise Guided Tour." *Advances in Distributed Systems*, **LNCS 1752**:33–47, 2000. pages 2, 74

[GM01] Brian P. Gerkey and Maja J. Matarić. "Principled Communication for Dynamic Multi-Robot Task Allocation." In D. Rus and S. Singh, editors, *Experimental Robotics VII, LNCIS 271*, pp. 353–362. Springer-Verlag, Berlin, 2001. pages 83

[GM02] Brian P. Gerkey and Maja J. Matarić. "Sold!: Auction methods for multi-robot coordination." *IEEE Transactions on Robotics and Automation*, **18**(5):758–768, 2002. pages 83

[GS99] S. Gupta and P.K. Srimani. "An Adaptive Protocol for Reliable Multicast in Mobile Multi-Hop Radio Networks." In *IEEE Workshop on Mobile Computing Systems and Applications*, 1999. pages 20, 41, 42

[HCSar] Pan Hui, Augustine Chaintreau, James Scott, Richard Gass, Jon Crowcroft, and Christophe Diot. "Pocket Switched Networks and the Consequences of Human Mobility in Conference Environments." In *Workshop on Delay Tolerant Networking, ACM SIGCOMM*, 2005 (to appear). pages 100

[JFW02] Brad Johanson, Armando Fox, and T. Winograd. "The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms." *IEEE Pervasive Computing*, **1**(2), April 2002. pages 82

[JMB01] David B. Johnson, David A. Maltz, and Josh Broch. "DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks." In Charles E. Perkins, editor, *Ad Hoc Networking*, chapter 5, pp. 139–172. Addison-Wesley, 2001. pages 1

[JOW02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. "Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet." *SIGOPS Oper. Syst. Rev.*, **36**(5):96–107, 2002. pages 100

[KB05] Alan Kaminsky and Chaithanya Bondada. "Tuple Board: a new distributed computing paradigm for mobile ad-hoc networks." In *Extended Abstracts - 2005 GCCIS Conference on Computing and Information Sciences*, 2005. pages 84

[KC02] Thomas Kunz and Ed Cheng. "On-demand multicasting in ad-hoc networks: Comparing AODV and ODMRP." In *International Conference on Distributed Computing Systems*, 2002. pages 112

[KNE03] David Kotz, Calvin Newport, and Chip Elliott. "The mistaken axioms of wireless-network research." Technical Report TR2003-467, Dept. of Computer Science, Dartmouth College, July 2003. pages 15

[Lam98] Leslie Lamport. "The part-time parliament." *ACM Transactions on Computer Systems*, **16**(2):133–169, 1998. pages 50, 92

[LEH03a] Jun Luo, Patrick Th. Eugster, and Jean-Pierre Hubaux. "PAN: Providing reliable storage in mobile ad hoc networks with probabilistic quorum systems." In *Proceedings of the 4nd ACM/SIGMOBILE Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc'03*, pp. 1–12, 2003. pages 2

[LEH03b] Jun Luo, Patrick Th. Eugster, and Jean-Pierre Hubaux. "Route Driven Gossip: Probabilistic Reliable Multicast in Ad Hoc Networks." In *Proceedings of the 22nd Conference of the IEEE Communications Society*, 2003. pages 2, 75

[LEH04] Jun Luo, Patrick Th. Eugster, and Jean-Pierre Hubaux. "Pilot: probabilistic lightweight group communication system for ad hoc networks." *IEEE Transactions on Mobile Computing*, **03**(2):164–179, April 2004. pages 75

[LG97] Chunhung Richard Lin and Mario Gerla. "Adaptive Clustering for Mobile Wireless Networks." *IEEE Journal of Selected Areas in Communications*, **15**(7):1265–1275, 1997. pages 41

[LSG00] S. Lee, W. Su, and M. Gerla. "On-Demand Multicast Routing Protocol in Multihop Wireless Mobile Networks.", 2000. pages 109, 112, 113

[LSH00] Sung-Ju Lee, William Su, Julian Hsu, Mario Gerla, and Rajive Bagrodia. "A Performance Comparison Study of Ad Hoc Wireless Multicast Protocols." In *Proceedings of the 19th Conference of the IEEE Communications Society*, pp. 565–574, 2000. pages 47, 112

[LW04] Wei Lou and Jie Wu. "Double-Covered Broadcast (DCB): A Simple Reliable Broadcast Algorithm." In *Proceedings of the 23rd Conference of the IEEE Communications Society*, 2004. pages 2

[Maled] Dahlia Malkhi. "Quroum Systems." In *The Encyclopedia of Distributed Computing.* Kluwer Academic Publishers, To be published. pages 86

[MR99] Achour Mostefaoui and Michel Raynal. "Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: A General Quorum-Based Approach." In *Proceedings of the 13th International Symposium on Distributed Computing*, pp. 49–63, London, UK, 1999. Springer-Verlag. pages 50

[MVW01] Navneet Malpani, Nitin H. Vaidya, and Jennifer L. Welch. "Distributed Token Circulation on Mobile Ad Hoc Networks." In *Proceedings of the Ninth International Conference on Network Protocols (ICNP'01)*, p. 4, Washington, DC, USA, 2001. IEEE Computer Society. pages 75

[NTC99] S Ni, Y Tseng, Y Chen, and J Sheu. "The Broadcast Storm Problem in a Mobile Adhoc Network." In *the proceedings of the ACM Int. Conf. on Mobile Computing and Networking (MOBICOM)*, pp. 151–162, 1999. pages 35, 36, 45

[Per97] Charles E. Perkins. "Ad-hoc on-demand distance vector routing." In *Proceedings of MILCOM*, 1997. pages 1

[Pin93] J. Pinakis. *Using Linda as the Basis of an Operating System Microkernel.* PhD thesis, The University of Western Australia, 1993. pages 83

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. "LIME: Linda Meets Mobility." In *Proceedings of the 21st International Conference on Software Engineering*, pp. 368–377, 1999. pages 84

[PR97] Elena Pagani and Gian Paolo Rossi. "Reliable Broadcast in Mobile Multihop Packet Networks." In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 1997. pages 20, 41

[RBH98] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. "Building adaptive systems using ensemble." *Softw. Pract. Exper.*, **28**(9):963–979, 1998. pages 50

[RBM96] Robbert Van Renesse, Kenneth P. Birman, and Silvano Maffeis. "Horus, a flexible Group Communication System." *Communications of the ACM*, **39**(41):76–83, 1996. pages 50, 93

[Rin88] G. A. Ringwood. "Parlog86 and the dining logicians." *Communications of the ACM*, **31**(1):10–25, 1988. pages 82

[RK03] Prashant Ratanchandani and Robin Kravets. "A Hybrid Approach to Internet Connectivity for Mobile Ad Hoc Networks." In *Proceedings of IEEE Wireless Communications and Networking*, pp. 1522–1527, 2003. pages 2

[SB04] Yuan Sun and Elizabeth M. Belding-Royer. "A Study of Dynamic Addressing Techniques in Mobile Ad hoc Networks." In *Wireless Communications and Mobile Computing*, April 2004. pages 83

[Sch80] Fred B. Schneider. "Ensuring Consistency in a Distributed Database System by Use of Distributed Semaphores." In *Symposium on Distributed Data Bases*, pp. 183–189, 1980. pages 82

[SRB01] A. Striegel, R. Ramanujan, and J. Bonney. "A Protocol Independent Internet Gateway for Ad-Hoc Wireless Networks." In *Proceedings of Local Computer Networks (LCN)*, Tampa, Florida, November 2001. pages 2

[TOL02] Ken Tang, Katia Obraczka, Sung-Ju Lee, and Mario Gerla. "Reliable, Congestion Controlled Multicast Transport Protocol in Multimedia Multi-hop Networks." In *Proceedings of WPMC*, 2002. pages 20, 41, 43

[TS92] John Turek and Dennis Shasha. "The Many Faces of Consensus in Distributed Systems." *Computer*, **25**(6):8–17, 1992. pages 46

[VKT04] Sudarshan Vasudevan, Jim Kurose, and Don Towsley. "Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks." In *12th IEEE International Conference on Network Protocols*, pp. 350–360, 2004. pages 2, 75

[vMH98] Robbert van Renesse, Yaron Minsky, and Mark Hayden. "A Gossip-Based Failure Detection Service." In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pp. 55–70, 1998. pages 50

[WAF04] Peng-Jun Wan, Khaled M. Alzoubi, and Ophir Frieder. "Distributed construction of connected dominating set in wireless ad hoc networks." *Mobile Networks and Applications*, **9**(2):141–149, 2004. pages 44

[WML98] Peter Wyckoff, Stephen McLaughry, Tobin Lehman, and Daniel Ford. "TSpaces." *IBM Systems Journal*, August 1998. pages 82

[WWV01] Jennifer E. Walter, Jennifer L. Welch, and Nitin H. Vaidya. "A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks." *Wireless Networks*, **7**(6):585–600, 2001. pages 2, 75

[XL89] Andrew Xu and Barbara Liskov. "A design for a fault-tolerant, distributed implementation of Linda." In *Ninteenth International Symposium on Fault-Tolerant Computing, Digest of Papers*, pp. 199–206, June 1989. pages 90

[YB04] Eiko Yoneki and Jean Bacon. "An Adaptive Approach to Content-Based Subscription in Mobile Ad Hoc Networks." In *IEEE International Conference on Pervasive Computing and Communications - Workshop on Mobile Peer-to-Peer Computing*, 2004. pages 112

[ZBG98] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. "GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks." In *Workshop on Parallel and Distributed Simulation*, pp. 154–161, 1998. pages 15, 112

Appendix A

Responsibility transfer for reliable broadcast

A.1 Introduction

The main part of this dissertation has concerned itself with developing *fault-tolerant* protocols for mobile ad-hoc networks. When fault-tolerance is a key design issue, replication becomes important. For example, the tuple space in chapter 5 was designed to replicate each tuple as widely as possible, thus maximising both fault-tolerance and availability of each tuple. However, in certain scenarios failures may be highly unlikely, and we can avoid the replication required to tolerate failures.

Even when failures are not considered, the coverage a best-effort multicast or broadcast protocol can achieve may fall catastrophically below what the end user might expect. This can cause problems if the message to be delivered contains important information (such as for example code updates which need to get to all nodes).

End-to-end acknowledgements between the originator of a message and all the receivers is a possibility in cases where guaranteed delivery is important, but this can result in long delays for the originator before it can discard each message. This is particularly true when the network is frequently partitioned and end-to-end paths between nodes are sporadic at best.

An alternative to end-to-end acknowledgements is to pass the responsibility for ensuring delivery of a message between nodes. This is the approach taken to achieve reliable unicast between nodes in so called Delay Tolerant Networks (DTNs)[Fal03], where it is called *custody transfer*. DTNs are networks where no end-to-end path is assumed to exist, and includes the kind of transiently partitioned mobile ad-hoc networks considered in this dissertation.

Using custody transfer, a "better" node to transfer custody to is searched for, and when found, a transaction-like handshake takes place to transfer custody between the two nodes. The metric by which a node to transfer custody to is chosen is typically dependent on how likely the node is to get the message closer to the destination

However, the most recent IETF internet draft covering the DTN custody transfer mechanism states that the exact meaning and design of custody transfer for *multicast* delivery remains to be fully explored [Cer05]. For example, when delivering to more than one destination one might want more than one node to be able to have custody of a given message. In such cases, the quite heavy-weight method of transferring responsibility between two nodes may not be suitable.

This chapter presents an alternate method for transferring responsibility between nodes to ensure reliable broadcast in mobile ad-hoc networks. A reliable, though not fault-tolerant, broadcast protocol called *Scribble*, is derived by combining this method with the basic proactive dissemination strategy from section 3.2.3. The performance of Scribble is compared and contrasted to the best-effort multicast protocol ODMRP[LSG00].

Note that as there are no failures, assumptions such as each node being assigned a unique id and being aware of the value of n can be trivially met by having a single centralised node maintaining group membership.

A.2 The Scribble protocol

The next section describes the responsibility transfer mechanism, including explaining how to make use of the mechanism to implement the Scribble protocol.

The assumption that failures do not happen allows Scribble to guarantee delivery to all nodes in the system. However, even with no failures, the theorem that shows how possibly all nodes may have to become responsible for this to be achieved still holds (theorem 2.1 in section 2.3.1).

Scribble handles this by gradually making all nodes responsible if the protocol appears not to be terminating. Section A.2.2 explains how this is handled.

A.2.1 Responsibility transfer mechanism

The responsibility for message dissemination initially rests with the broadcast originator and is subsequently passed on to other nodes (as in a relay-race). Consequently, a node can be in one of two states regarding the broadcast of m: *responsible* or *passive*. A responsible node behaves as in PDP (section 3.2.3); it transmits m once, and then repeats this transmission every β seconds if required. A node in the passive state does not transmit, nor retain m. Note that a node which has not received m is considered to be passive.

Ideally, the number of nodes simultaneously responsible should be kept low, particularly when a higher number does not provide any further coverage. The mechanism used for transferring responsibility achieves this objective by striving to keep at most one node responsible in any subset of nodes that are in each others wireless range (a *fully connected* subset); moreover, at least one node is kept responsible in the group as a whole at any time.

The initiator is the first node to know of, and to be responsible for, m. The responsibility transfer mechanism makes use of *logical clocks*. A logical clock is just an integer counter whose value can only increase, though not necessarily in relation to the passage of real-time. A node N_i constructs a logical clock $L_i(m)$ with the initial value of zero when N_i first knows of m. Whenever N_i transmits a message, $L_i(m)$ is added to it; this (logical) time-stamp is denoted as $m.l$ of the transmitted m.

In addition, the mechanism makes use of a random assessment delay, RAD. Recall that a RAD is a short time interval randomly chosen between 0 and some maximum value by a node which is about to transmit. During the RAD a node will assess the action of neighbouring nodes to see if their actions change what it will transmit.

Suppose that a node N_j is passive on m and receives m. N_j becomes responsible for m if :

R1 : $m.l$ of the received m is greater than, or equal to $L_j(m)$, and

R2 : N_j has not received m in the past β-δ seconds.

The newly responsible node's first transmission of m is preceded by the following activities: N_j adopts the incoming $m.l$ as its $L_j(m)$ and chooses a RAD uniformly distributed on (0, MAX_RAD). The first transmission of m is *scheduled* for when the RAD has expired. During the RAD, N_j assesses any incoming values of $m.l$, and sets $L_j(m)$ to the highest received such value. When the RAD expires, N_j increments $L_j(m)$ by 1, stamps m with the value of $L_j(m)$ and transmits m.

A responsible node N_i becomes passive if it receives m such that:

P1 : $m.l$ of the received $m > L_i(m)$.

Upon becoming passive, N_i cancels any any pending transmission of m and deletes it.

Remark 1. If N_j that has just become responsible, receives *another* transmission of m that meets **P1**, then it instantly becomes passive canceling any RAD that it had just set and any transmission scheduled. This means a responsible node could become passive before ever transmitting m.

Remark 2. When two responsible nodes receive each other's m, this method does not permit both nodes to become passive. However, one of them is likely to become passive except in the case where both have identical logical clock values and transmit m nearly at the same instant despite RAD.

An Example. Consider a (fully connected) subset $\mathcal{C}$ of nodes that are in wireless range of each other. Let us suppose that $|\mathcal{C}| < n$ and that $\mathcal{C}$ contains c_0 which initiates a dissemination of m. Also suppose that every node in $\mathcal{C}$ other than c_0 receives m with $m.l = 1$ and becomes responsible after setting $L(m)$ to 1. If $c_1 \in \mathcal{C}$ chooses the smallest RAD it ends up transmitting m with $m.l = 2$. Upon receiving m from c_1, c_0 will become passive (by **P1**). If we assume that all other nodes receive m from c_1 before their respective RAD expires (i.e., before they transmit m which would result in increasing their $L(m)$ to 2), only c_1 will be responsible in $\mathcal{C}$. That is, the responsibility for m has now been passed on from c_0 to c_1.

Consider next a node $d \notin \mathcal{C}$ that is in wireless range with only c_1 in $\mathcal{C}$ and receives m with $m.l = 2$. When d becomes responsible and transmits m with $m.l = 3$, c_1 becomes passive. Note that $\mathcal{C}$ now has no active node in it, while another fully connected subset (which contains c_1 and d) gains a responsible node.

A.2.2 Protocol termination

Scribble ensures that any responsible node is able to deduce the achievement of the desired coverage once the latter is obtained, and thus terminate, using the same distributed detection mechanism as described in section 3.2.2.

The only modification to accommodate passive nodes is that if a node which receives an $m.K$ which does not contain its signature, and if it subsequently decides not to transmit m, it transmits a small *acknowledgment packet* for m. It is worth noting that this does not cause an ack-implosion, as the acknowledgment packets only ever travel one hop.

However, as discussed in section 2.3.1, the network can act *like an adversary*, enabling the direct connectivity between nodes which have received m and which has not received m, only when the former is passive. This means that a protocol which keeps only a subset of nodes responsible, however cleverly designed, cannot guarantee that the right nodes are responsible at the right time.

To illustrate this in the context of a protocol which allows nodes to transfer responsibility, let us revisit the example from the previous section. The fully-connected subset $\mathcal{C}$ contains c_0 which initiates a dissemination of m. Since $|\mathcal{C}| < n$, some more nodes, such as $d \notin \mathcal{C}$, must receive m. In the absence of any such d which enters the wireless range of some $c_i \in \mathcal{C}$, it is easy to see that the desired coverage cannot be obtained.

Putting the above differently, the desired coverage is guaranteed only if some $c_i \in \mathcal{C}$ has in its wireless range one or more nodes, such as $d \notin \mathcal{C}$, for a period of at least $\beta + 2\delta$ (i.e. c_i and d are in direct connectivity) *and* if c_i is responsible during this period of direct connectivity. The former will occur since the network meets the NLR (see section 2.2) and when it does occur, the protocol must ensure the latter.

Note that the network can choose *any* $c_i \in \mathcal{C}$ and place the chosen c_i and d in direct connectivity at arbitrarily chosen timing instants. Indeed, the network can behave *like an adversary*, enabling the direct connectivity between c_i and d only when the former is passive. This means that a protocol which keeps only a subset of nodes in $\mathcal{C}$ responsible, however cleverly designed, cannot guarantee that the right nodes in $\mathcal{C}$ are responsible at the right time.

So, the measure taken by Scribble involves gradually allowing all nodes in $\mathcal{C}$ to become responsible when the broadcast appears not to be terminating. This is achieved by requiring each node, N_i, to have a parameter θ_i. If $\mathrm{L}_i(m) \geq \theta_i$ or if N_i receives m with $m.l \geq \theta_i$, then N_i becomes responsible (if it is not already) and does not become passive until it realises m.

If this happens, the execution of Scribble is almost indistinguishable from the (fault-tolerant) proactive dissemination protocol, PDP, introduced in section 3.2.3. The discussion above, coupled with theorem 2.1, indicates that *any* reliable protocol must contain the possibility of executions similar to PDP. Otherwise, termination cannot be guaranteed even when the network satisfies the NLR, but behaves like an adversary.

Choosing a suitable value for θ is highly dependent on the scenario in which the protocol is deployed. Transient, and even prolonged, partitions are common in some ad-hoc networks, and their occurrences should not necessarily be misread by the protocol as adversarial network behaviour. θ is therefore left as a configurable parameter.

A.3 Performance evaluation

This section compares Scribble to ODMRP[LSG00], a best-effort multicast protocol. Although ODMRP does not provide any delivery guarantees it should act as an interesting benchmark. It has fared well in various simulation studies[LSH00, KC02], and is also widely used; for example acting as the underlying transport protocol for the event-based middleware proposed for pervasive computing environments by Yoneki and Bacon[YB04].

In addition, comparing Scribble to ODMRP provides the first comparison between a best-effort and a reliable broadcast protocol. This comparison is an additional contribution as it provides an insight into the cost associated with affording the protocol user *certainty* that a message will be delivered to the specified number of nodes.

A.3.1 Simulation Model

The simulation parameters used are the same as those described in section 2.1 in section 2.4, though the simulator used was GloMoSim[ZBG98] not Jist/SWANS as in the main dissertation.

In all experiments Scribble set aside 64 bytes for signatures in each data header. All simulations used a byte for each node signature. θ was chosen to be a high value such that no simulations included executions where all nodes were forced by Scribble to be responsible.

Scribble's performance was studied in both sparse (when wireless range is 150m), as well as dense networks (wireless range = 350m). The following performance metrics were considered:

1. *Transmission Overhead*: Measures the total number of bytes transmitted by each node in order to complete one reliable broadcast. This includes both control (ack, topology control, realisation, etc.) and data packets (including packet headers). It is measured in bytes transmitted per byte broadcast, and was measured at the MAC layer for both protocols.

2. *Latency:* Measures the average time from a node initiates a the broadcast of a message until a node receives that message. This was measured at the application layer.
3. *Percentage Successful Runs (PSR):* Measures the percentage of protocol runs where a message was delivered to all nodes. This will always be 100% for Scribble as it is a reliable broadcast protocol. This was also measured at the application layer.

A.3.2 Simulation Results

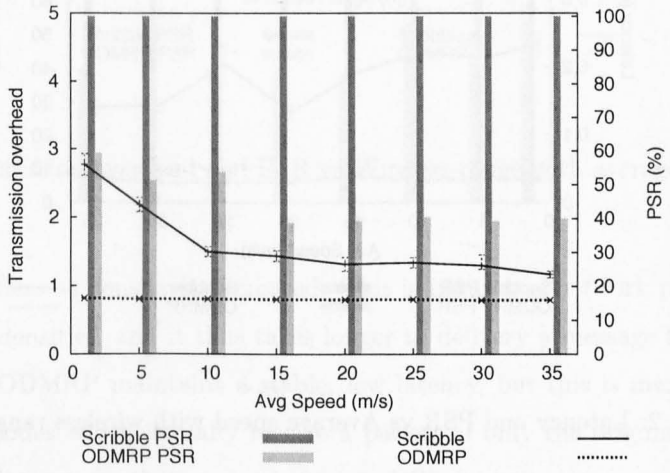

Figure A.1: Transmission overhead and PSR vs Average speed with wireless range = 250m

Figure A.1 shows the impact that variable node speeds has on the two protocols, when the wireless range of the nodes fixed at 250m. PSR is shown as bars on the right y-axis, while the transmission overhead is drawn as lines relative to the left y-axis.

As expected, ODMRP performs relatively well in terms of overhead with these parameters (these conditions are in fact very similar to those chosen by the ODMRP authors themselves in [LSG00]), with mobility having little impact on the transmission overhead, as there are no reconstruction efforts in place in case the routing mesh breaks. However the average number of times ODMRP is able to deliver a message to all intended recipients is below 60% even in the most static scenario (1m/s), with the PSR dropping to just under 40% in the most mobile case (35m/s). Further, when the number of successful recipients of individual packets is studied, one can observe that in extreme cases, when the originating node is partitioned from the rest of the network, the protocol terminates with only the originating node itself having received the packet.

Scribble on the other hand, provides its delivery guarantees with little additional overhead com-

pared to ODMRP, with the additional overhead being higher when the mobility is relatively low. The reason for this is that Scribble guarantees delivery to all nodes including those which might be transiently partitioned from the rest. When the mobility is low, these partitions takes longer to heal, so the cost of guaranteeing delivery to partitioned nodes is higher, thus increasing the overall cost of guaranteeing delivery.

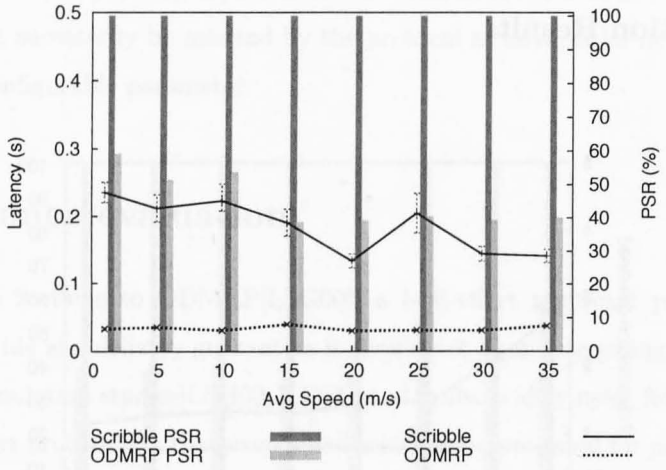

Figure A.2: Latency and PSR vs Average speed with wireless range = 250m

Figure A.2 shows the average latency of ODMRP and Scribble for the same scenario as above. What is immediately obvious is that Scribble has a higher latency than ODMRP. The reason for this is mainly that Scribble, as part of its structure-less dissemination mechanism, has to observes its neighborhood for a small amount of time before deciding whether or not to retransmit a packet. A node in ODMRP, on the other hand, knows instantly whether to retransmit a packet, as this decision is based simply on whether it is part of the routing mesh or not.

Figure A.3 shows how transmission overhead and PSR vary as function of node wireless ranges. These results show very clearly the cost of attaining certainty that a message will be delivered to enough nodes. As one would expect, the cost of guaranteeing delivery is not excessive when the network is fairly dense (remember we do not consider very congested conditions). In fact ODMRP and Scribble have almost identical overheads in the densest case considered, though note that even here ODMRP is not able to provide delivery to all nodes in more than 72% of cases. As the density decreases, the PSR ODMRP achieves drops dramatically, going as low as 0.2% of cases in the sparsest network (this is too small to show up on the graph). Scribble on the other hand maintains its 100% PSR, though at what can be considered considerable costs in the sparsest cases.

Figure A.4 shows how latency is impacted by variations in density. The general trend is that the

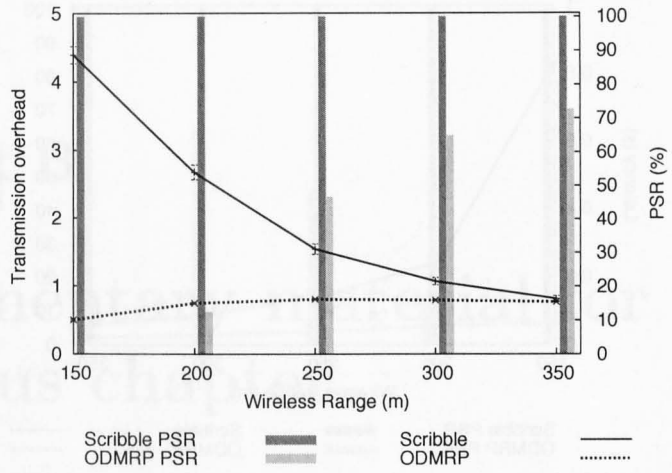

Figure A.3: Transmission overhead and PSR vs Wireless range with average speed = 5 m/s

latency of Scribble suffers as density is decreased. This is natural as network partitioning becomes more frequent in low densities, and it thus takes longer to delivery a message to nodes partitioned from the originator. ODMRP maintains a stable, low latency, but this is merely because latency is only measured to nodes which actually receive a packet; if only the originator receives a given message, the average latency for that packet is 0 seconds(!).

A.4 Conclusion and summary

This chapter has introduced a novel mechanism for transferring responsibility for ensuring delivery of a message to a group of nodes. This responsibility transfer mechanism is analogous to the custody transfer mechanism proposed for reliable unicast in delay tolerant networks, though is designed for reliable broadcast in mobile ad-hoc networks.

A reliable broadcast protocol, Scribble, which makes use of the responsibility transfer mechanism has been introduced, and its performance compared to that of the best-effort multicast protocol ODMRP. The resulting performance study showed that Scribble overall had a higher transmission overhead as well as latency than ODMRP in most cases.

However, in a number of scenarios the difference was not very great. Further, the simulations showed that in none of the scenarios studied was ODMRP able to deliver a message to all nodes every time it was invoked. Indeed, in some sparse scenario the percentage of times ODMRP was successful in this was less than 0.2% of the times it was invoked.

This result highlights the inherent trade-off between a reliable and a best-effort protocol; a

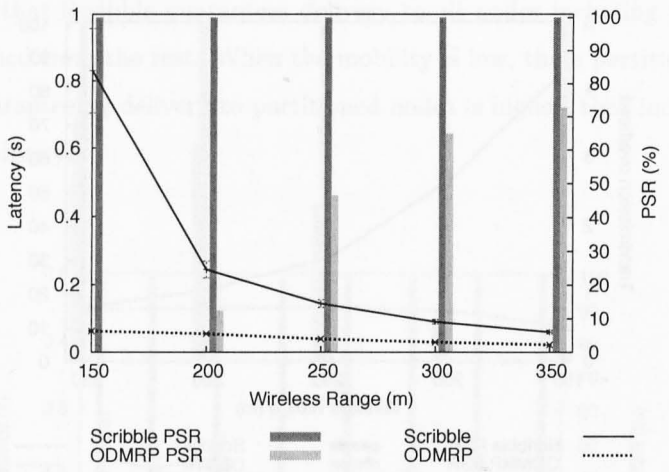

Figure A.4: Latency and PSR vs Wireless range with average speed = 5 m/s

reliable protocol might be more expensive in terms of overhead and latency *per invocation*, however with a reliable protocol the end user only ever needs to invoke it once. With an unreliable protocol the end user may have to invoke it several times to get anywhere near acceptable level of coverage.

Appendix B

Supplementary material for the consensus chapter

B.1 Pseudo code for the combined consensus protocol

The combined consensus protocol is in two main threads. The core is in the consensus thread, depicted in figure B.1. It contains the replenish optimisation. The replenishing and emptying of the local bag of values is seen on lines 14 and 34.

The cross layer and adoption optimisations is in the receive thread depicted in figure B.2. The cross layer optimisation involves nodes adding any values they know if in the message body prior to the dissemination protocol signing the message (lines 27 and 29).

How a node should behave when it receives a higher round message is shown on lines 4-12, and how it should behave when it receives a higher phase message is shown on lines 13-26.

Receiving a higher round message involves emptying your local bag, setting the round number to the received round number, setting the phase number to 0 and stopping the ongoing dissemination of the (lower round) consensus message. The `onReceiveConsensusMessage()` function is then called recursively.

Receiving a higher phase message involves adopting the incoming phase number and starting to disseminate the received dissemination message in the appropriate place (involves setting the consensus thread control to the appropriate line).

One further change is necessary; on line 32 in the consensus thread, it is possible that a node has no values to choose randomly from it receives a higher round, phase two message. In this case, the consensus blocks on this call, and the dissemination protocol keeps disseminating the phase two message. This means that eventually a correct node which has values in its bag realises the phase two message, makes a choice, and then moves into the next phase (thus unblocking any nodes which had empty bags).

```
1
2
3   Consensus(preference)
4   {
5           bag[] = ∅;
6           round = 1;
7           phase = 1;
8
9           while(true)
10          {
11                  est[] ={preference};
12                  disseminate consensusMsg(phase, round, est[]) to n/2.
13                  wait until consensusMsg has more than (n/2) signatures;
14                  bag[] = est[];        //replenish bag with received values and current estimate
15                  if( |est[]| > 1)
16                  {
17                          est[] ={⊥};          //then more than one value proposed
18                  }
19                  phase = 2;
20                  disseminate consensusMsg(phase, round, est[]) to n/2 nodes;
21                  wait until consensusMsg has more than (n/2) signatures;
22                  if( est[] contains at least one value, v  ≠ ⊥)
23                  {
24                          preference = v;
25                          if(  est[] contains only one value, v  ≠ ⊥)
26                          {
27                              decide(v); //sends a realisation packet with decided value.
28                          }
29                  }
30                    else (est[] only contains ⊥)
31                  {
32                          preference = bag[random];  //maybe empty; block in that case
33                  }
34                  bag[] = ∅; //empties bag
35                  round++;
36                  phase = 1;
37          }
38  }
39
40
```

Figure B.1: The main consensus thread of the combined protocol

```
1
2  onReceiveConsensusMsg(incomingMsg)        //the Receive Thread
3  {
4          if(incomingMsg.round > round)
5          {
6                  bag[] = ∅
7                  round = incomingMsg.round;
8                  phase = 0;
9                  consensusMsg = null; //stops any ongoing dissemination
10                 onReceiveConsensusMsg(incomingMsg); //recursive
11                 return;
12         }
13         if(incomingMsg.phase > phase)
14         {
15                 phase = incomingMsg.phase;
16                 consensusMsg = incomingMsg;
17                 if(incomingMsg.phase == 1)
18                 {
19                         set Consensus thread to line 11
20                 }
21                 else
22                 {
23                         set Consensus thread to line 20;
24                 }
25
26         }
27         if(incomingMsg.est[] contains values not in consensusMsg.est[])
28         {
29                 add values not found in consensusMsg.est[]
30         }
31         return; //When this returns, the dissemination protocol signs the message as before
32 }
33
34
```

Figure B.2: The receive thread of the combined protocol

B.2 Performance study of the combined consensus protocol

The simulation environment for this performance study is as described in section 2.4. The measures of interest to us are the number of rounds, the time until a decision is reached and the message passing overhead per round. We will see how these three measures are impacted by varying nodes' wireless ranges (and thus varying the network density), the nodes' average speeds (and thus varying the rate of topological change), the number of nodes in the system, n, and finally the ratio of tolerated failures, f. In all simulations , the number of actual failures induced was half the number of tolerated failures.

Figures B.3, B.4 and B.5 shows the impact (or lack of) when changing the maximum speed of the nodes. For 36 nodes, with 250m wireless range, the node speeds appear to have no impact either on the number of rounds, the average overhead per round or the time until decision. This is a good result for ad-hoc networks where increasing node speeds often leads to deteriorating performance.

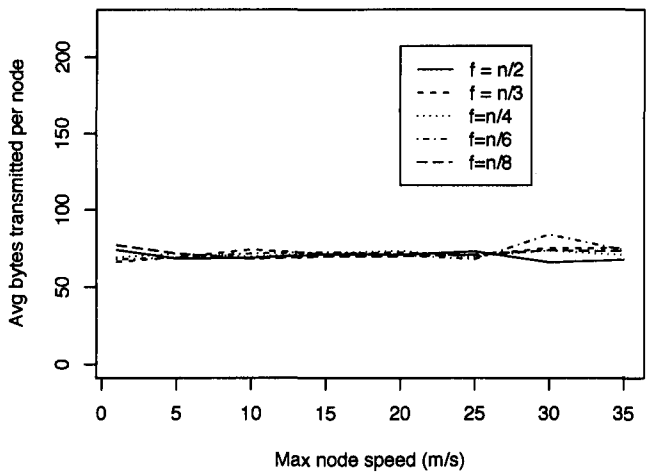

Figure B.3: Average number of bytes transmitted per node vs. Node speeds for $n = 36$ and 250m wireless range

Figures B.6, B.7 and B.8 shows the impact of varying the number of nodes while keeping the wireless range and node speed fixed at 250m and 10m/s respectively. It appears that when the number of nodes is low, the overhead per round is slightly higher (figure B.6). This is likely due to the properties of the dissemination protocol; for example, the dissemination protocol will suppress a transmission after hearing a number of equivalent transmissions. If there are few nodes in the system, this is unlikely to happen, and a lot of nodes (relatively speaking) will end up transmitting.

The number of rounds appear to be unaffected by the increasing number of nodes, even for $f = n/2$ (figure B.7). This shows that the replenishment and adoption optimisations work well, and

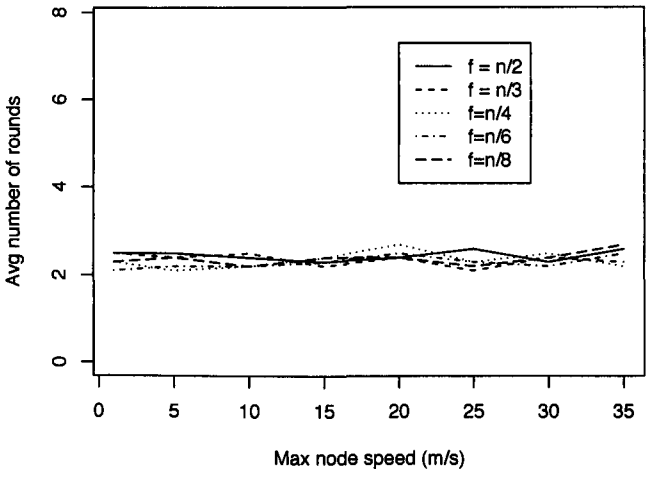

Figure B.4: Average number of rounds vs. Node speeds for $n = 36$ and 250m wireless range

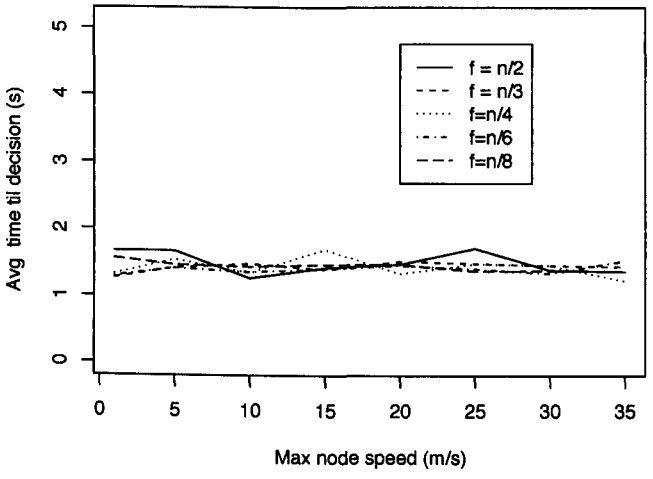

Figure B.5: Average time till decision vs. Node speeds for $n = 36$ and 250m wireless range

is a significant improvement on the EMR and Ben-Or protocols when they are run in lockstep.

Figure B.8 shows how the time till a decision is reached is increased as the number of nodes increases. This is natural, as it takes longer to get a majority of votes when the number of nodes is higher. The increase appears to be only linear, and again, unaffected by the number of tolerated failures.

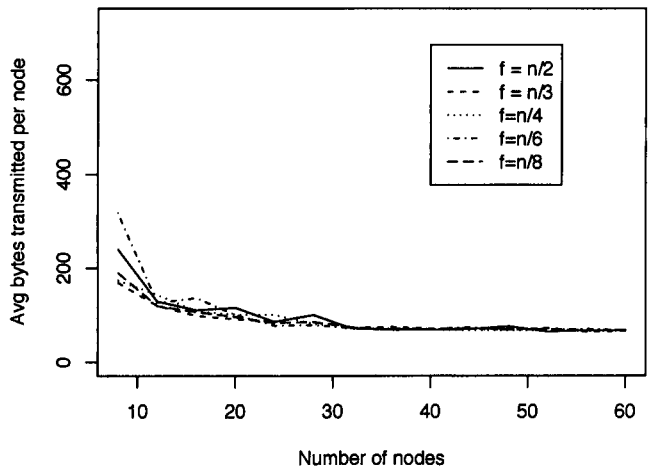

Figure B.6: Average number of bytes transmitted per node vs. Number of nodes for 10m/s node speeds and 250m wireless range

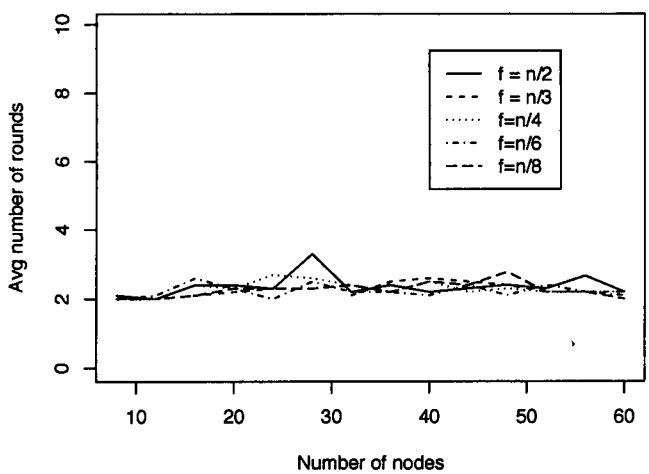

Figure B.7: Average number of rounds vs. Number of nodes for 10m/s node speeds and 250m wireless range

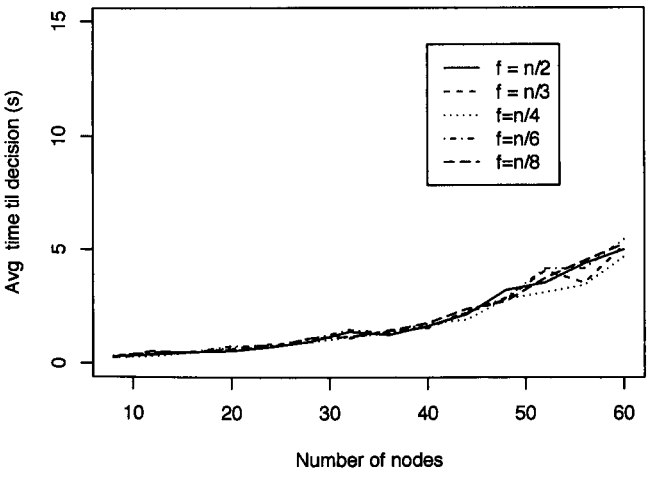

Figure B.8: Average time till decision vs. Number of nodes for 10m/s node speeds and 250m wireless range

Finally, figures B.9, B.10 and B.11 shows the impact of varying the node density by varying the nodes' wireless ranges. Again, the number of rounds appears unaffected, while the latency and the transmission overhead per round is higher at lower densities. This mirrors the result for the dissemination protocol where lower densities cause higher overheads. This is because at low densities the network is likely to be partitioned more frequently and for longer durations.

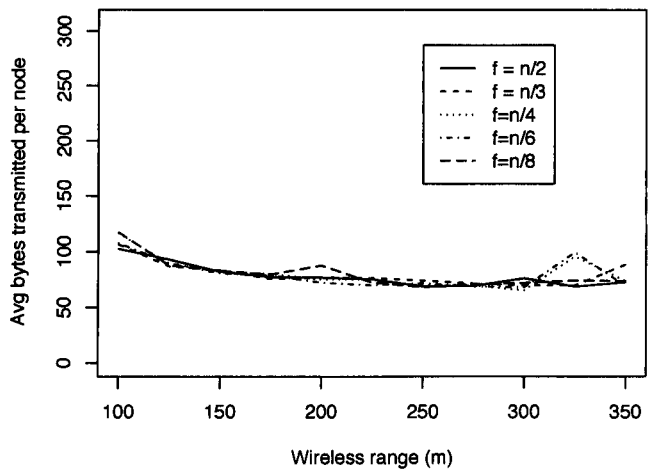

Figure B.9: Average number of bytes transmitted per node vs. Wireless range for 10m/s node speeds and $n = 36$

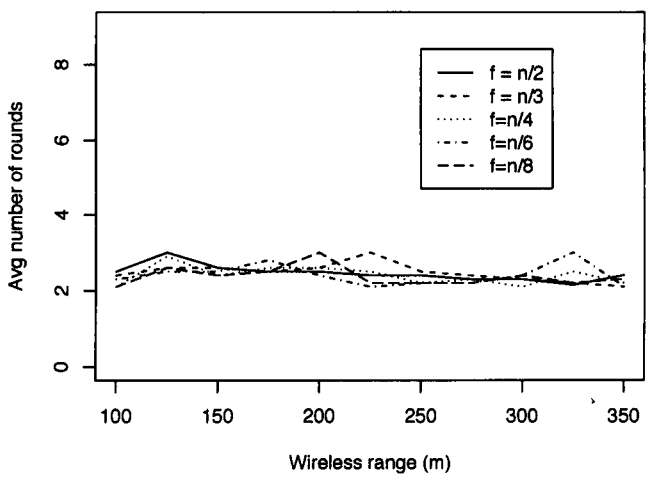

Figure B.10: Average number of rounds vs. Wireless range for 10m/s node speeds and $n = 36$

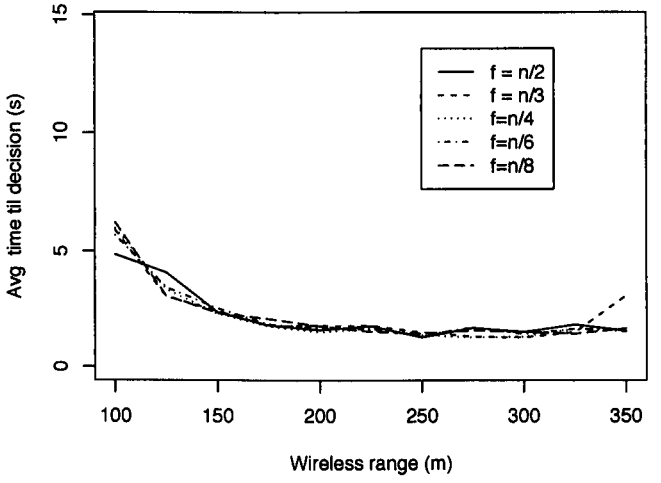

Figure B.11: Average time till decision vs. Wireless range for 10m/s node speeds and $n = 36$