POLITECNICO DI MILANO

V Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

EPIDEMIC ALGORITHMS FOR RELIABILITY IN MOBILE CONTENT-BASED ROUTING

Relatore: Prof. Gian Pietro PICCO
Correlatore: Prof. Gianpaolo CUGOLA

Paolo COSTA Matr. 630935
Matteo MIGLIAVACCA Matr. 632026

Anno Accademico 2001-2002

Acknowledgments

We would like to thank our advisors Prof. Gian Pietro Picco and Prof. Gianpaolo Cugola for trusting, encouraging, and supporting us all throughout our thesis work. Through advice and lively discussions, they have contributed a lot to the development of this work. We also want to thank Prof. Carlo Ghezzi for arousing our interest in Software Engineering during his lectures and for his support in many occasions.

We developed our research at the Software Engineering Laboratory of Politecnico di Milano. In this environment we had the opportunity to meet people that in so many ways contributed to this work. In addition to the ones we already mentioned, we would like to express our gratitude to Davide Balzarotti, Claudio Casoli, Mirco Cesarini, Davide Frey, Vincenzo Martena, Roberto Tedesco, Luca Zanolin and many others.

We are especially grateful to our class mates for their friendship and support throughout our university career. In particular, we would like to thank Andrea D'Angelo, Claudio Defferara, Daniele Dellafiore, Lorenzo Della Vedova, Paolo Falzoni, Matteo Giaconia, Stefano Guazzi, Marco Lambri, Luca Ludovico, Davide Merlani, Giuseppe Milani, Fabio Misani, Mauro Misino, Daniela Montoli, Matteo Morelli, Filippo Moriggia, Mauro Mosca, Fabio Negroni, Liliana Nenov, Gianluca Palermo, Sara Pasqui, Stefano Perfetti and Luigi Petruccelli.

Milano, December 2002

Contents

Chapter 1

Introduction

In recent years, scenarios characterized by an extreme degree of dynamicity and reconfiguration have attracted the attention of the research community.

For instance, in mobile computing, devices can come and go continuously, e.g.,because they are free to move since connected through wireless links or because they are employed by a user, and hence plugged in and out of the infrastructure according to his/her movements. The situation is emphasized in pervasive computing whereby a component can often be replaced by another providing equivalent services, e.g., because a location change determined a different context populated by a different set of components, or because the component is no longer operational and is replaced by a new one.

Another kind of reconfiguration can be found in peer to peer computing, whereby the physical network, in contrast to mobile settings, is fixed but bears at its top a logical network known as overlay network. Reconfiguration occurs at logical level and is triggered by peers joining and leaving the

network.

In these reconfigurable scenarios, communication and coordination of components is often a challenging task, given the unreliability of the environment, including the communication network.

In this scenario, the ability to decouple the internal behavior of a single device, application, or service, from the rest of the system becomes of paramount importance. Decoupling enables a component to work with little or no dependency on the current availability of other components.

Recently, the publish-subscribe communication paradigm has become popular precisely because of its decoupling characteristics. This paradigm revolves around the notion of *event*. Components interested in the occurrence of a given class of events *subscribe* expressing their interest. Components observing an event *publish* it[1] to the rest of the system. The decoupling is obtained through the fact that publishers and subscribers do not know each other: publish-subscribe operations, and in particular the delivery of an event to all the subscribers, are mediated by a component, the *event dispatcher*, whose architecture can be either centralized or distributed.

A number of publish-subscribe systems have been proposed to date. In this thesis we focus on systems that seek increased scalability and flexibility by exploiting a distributed event dispatcher, and empower the programmer with maximum expressiveness by using a content-based scheme for determining the match between an event and a subscription. Examples of systems of this sort can be found in [5, 39, 36, 4, 10]. The typical scenarios we target ,

[1]More precisely they publish an event notification.

with a large number of components that need to communicate in a dynamic environment justify this choice in preference to more conservative, less flexible, and less scalable solutions, adopting a centralized event dispatcher or less expressive subscription schemes.

This said, we may observe that while the publish-subscribe *model* appears very well-suited for the needs of those scenarios, the current state of the art in publish-subscribe *systems* is such that exploitation of publish-subscribe in this scenario is essentially hampered. Two are the main limitations:

1. the lack of mechanisms to deal with topological reconfiguration of the distributed dispatcher;

2. the lack of reliability, in terms of guaranteed event delivery in presence of "unreliable" environments characterized by wireless links, mobile devices, and so on.

Little or no published work tackles these two problems. Clearly, the two issues are intimately intertwined, in that topological reconfiguration of the dispatcher, e.g., induced by mobility or peer disconnections, is likely to disrupt the operation of the publish-subscribe system, and lead to the loss of events.

Previous works of the software engineering research group at Politecnico di Milano examined solutions for the first problem [11, 12, 30]. In this thesis, we focus instead on the second one. The ultimate goal of our research is to be complemented with the results obtained for reconfiguration so as to lead to the implementation of a new-generation distributed content-based publi-

sh-subscribe system capable of tolerating arbitrary reconfiguration and yet minimizing the number of events lost.

The approach we investigate in this thesis uses *epidemics algorithms* for recovering lost events. Epidemics algorithms [1, 23, 15] are a new breed of distributed algorithms that find inspiration in the theory of epidemics. They aim at providing a lightweight, scalable, and robust means of reliably disseminating information to a group of recipients, by providing guarantees only in probabilistic terms. Given their characteristics, epidemics algorithms are amenable to being used in highly dynamic and unreliable scenarios like those we target in this thesis.

1.1 Contribution of the Thesis

In this thesis, for the first time, a solution is presented to address reliability in such dynamic environments. The work consists of two parts:

Design of algorithms to achieve probabilistic reliability in distributed content-based publish subscribe systems. Our algorithms, adopt two different approaches *push* and *pull*, to recover events and suit different scenarios. The design of recovery mechanisms exploiting a synergy between epidemic technique and content-based publish subscribe has never been tackled before and requires the solution of several challenging issues.

Validation and Evaluation through simulation, of the aforementioned algorithms. We define our simulations as having the goal of verifying the

performance in reasonable scenarios and more critical ones and of evaluating their behavior against the change of environment and algorithm parameters. We also measure the overhead introduced in the network as the system size grows.

For an in-depth understanding of the mechanisms underlying epidemic algorithms and to acquire the necessary foundations to design our algorithms, we identified key aspects of epidemic techniques by analyzing published works exploiting them and built up an, albeit partial, classification. Since, to our knowledge, none of the existing works in literature presents a systematic review of epidemic algorithms, this classification can be regarded as a contribution per se.

1.2 Structure of the Thesis

The thesis is structured as follows. Chapter 2 provides the reader with the necessary background information related with content-based publish-subscribe systems. Chapter 3 presents epidemic algorithms with a classification of their relevant aspects and an overview of application areas. Chapter 4 analyzes the challenges posed by the application of epidemic algorithms in the specific context of content-based communication. Chapter 5 presents our main contribution, constituted by a number of algorithms that are designed to provide reliability in content-based publish-subscribe systems. Chapter 6 reports the simulation results evaluating our algorithms. Chapter 8 compares our approach against related work. Finally, Chapter 9 ends the thesis with

brief concluding remarks.

Chapter 2

Publish-Subscribe Middleware

2.1 Publish-Subscribe Paradigm

The recent demand in distributed systems has created the need for more flexible communication models and systems, reflecting the dynamic and decoupled nature of the applications.

The *publish/subscribe* interaction scheme has recently received increasing attention because it well-adapts to the loosely coupled nature of distributed communication in large scale applications over the Internet.

The *publish/subscribe* paradigm provides subscribers with the ability to express their interest in an event or pattern of events, in order to be notified of any event fired by a publisher, matching their interest.

An event is asynchronously notified to all subscribers that registered interest in that given event.

Applications exploiting publish-subscribe middleware are organized as a

collection of autonomous components, the *clients*, which interact by *publishing* events and by *subscribing* to the classes of events they are interested in. A component of the architecture, the *event dispatcher*, is responsible for collecting subscriptions and forwarding events to subscribers.

The communication and coordination model that results from this schema is inherently *asynchronous*; *multi-point*, because events are sent to all the interested components; *anonymous*, because the publisher need not know the identity of subscribers, and vice versa; *implicit*, because the set of event recipients is determined by the subscriptions, rather than being explicitly chosen by the sender; and *stateless*, because events do not persist in the system, rather they are sent only to those components that have subscribed before the event is published.

These characteristics result in a strong decoupling between event publishers and subscribers, which greatly reduces the effort required to modify the application architecture at run-time to cope with different kinds of changes in the external environment.

Given the potential of this paradigm, the last few years have seen the development of a large number of publish-subscribe middleware, which differ along several dimensions[1]. Two are usually considered fundamental: the expressivity of the subscription language and the architecture of the event dispatcher.

[1]For more detailed comparisons see [5, 10, 35].

2.2 Publish-Subscribe Systems Overview

2.2.1 Subscription Language

The expressivity of the subscription language draws a line between *subject-based* systems, where subscriptions identify only classes of events belonging to a given channel or subject, and *content-based* systems, where subscriptions contain expressions (called *event patterns*) that allow sophisticated matching on the event content.

A content-based approach enhances the underlying publish-subscribe middleware with unprecedented levels of flexibility, which in turn simplify significantly the programmer's task.

2.2.2 Architecture of the Event Dispatcher

The architecture of the event dispatcher can be either centralized or distributed. In this paper, we focus on publish-subscribe middleware with a distributed event dispatcher. In such middleware (see Figure 2.3) a set of interconnected *dispatching servers*[2] cooperate in collecting subscriptions coming from clients and in routing events, with the goals of reducing network load and increasing scalability.

The systems exploiting a distributed dispatcher can be further classified

[2]Unless otherwise stated, in the following we will refer to *dispatching servers* simply as *dispatchers*, although the latter term refers more precisely to the whole distributed component in charge of dispatching events, rather than to a specific server that is part of it.

according to the interconnection topology of dispatching servers, and the strategy exploited to route subscriptions and events. In this work we consider a subscription forwarding scheme on a undirected acyclic graph topology, as this choice covers the majority of existing systems.

2.2.3 Subscription Forwarding Scheme

In a subscription forwarding scheme [5], subscriptions are delivered to every dispatcher, along a single undirected acyclic graph connecting all dispatchers, and are used to establish the routes that are followed by published events. When a client issues a subscription, a message containing the event pattern is sent to the dispatcher the client is attached to. There, the event pattern representing the subscription is inserted in a subscription table, together with the identifier of the subscriber. Then, the subscription is propagated by the dispatcher, which now behaves as a subscriber with respect to the rest of the dispatching graph, to all of its neighboring dispatchers. In turn, these record the subscription and re-propagate it towards all their neighboring dispatchers, except for the one that sent it. This scheme is typically optimized by avoiding propagation of subscriptions to the same event pattern in the same direction[3]. The propagation of a subscription effectively sets up a route for events, through the reverse path from the publisher to the subscriber. Requests to unsubscribe from a given event pattern are handled and propagated

[3]Other optimizations are possible, e.g., by defining a notion of "coverage" among subscriptions, or by aggregating them, like in [5]. immediate impact on the content of this considered further.

Per dispatcher information:

- list of neighboring dispatchers
- subscription table, stored as a pair (*dispatcher, pattern*)

Process a subscription for a pattern p received from a neighboring dispatcher d

subscriptionReceived(d, SUB(p))

if d is not subscribed to pattern p **then**
  subscribe d to p in the local subscription table
  if this is the first subscription received for p **then**
    send SUB(p) to all neighbors except d
  else if this is the second subscription received for p **then**
    send SUB(p) to first subscriber
  end if
end if

Process an unsubscription to a pattern p received from a neighboring dispatcher d

unsubscriptionReceived(d, UNSUB(p))

if d is subscribed to p **then**
  remove the entry (d, p) from the local subscription table
  forwardUnsubFrom(d, p)
end if

Figure 2.1: Actions for subscription and event processing using a subscription forwarding scheme (I).

analogously to subscriptions, although at each hop entries in the subscription table are removed rather than inserted.

An informal description of the subscription forwarding strategy we use is provided in Figures 2.1 and 2.2, including actions for subscription management (subscriptionReceived, unsubscriptionReceived) and the action for event

Process event from a neighboring dispatcher or a client

eventReceived(d, EVENT)

send EVENT to all the dispatchers (except d) found subscribed to a matching pattern in the subscription table

Auxiliary macro for forwarding to the neighboring dispatchers an unsubscription for pattern p coming from node d

forwardUnsubFrom(d, p) $\equiv$

if there are no more dispatchers subscribed to p **then**

send UNSUB(p) to all neighboring dispatchers except d

else if there is only one dispatcher d subscribed to p **then**

send UNSUB(p) to d

end if

Figure 2.2: Actions for subscription and event processing using a subscription forwarding scheme (II).

forwarding (eventReceived). Action execution is assumed to be atomic, and message passing among components is assumed to be asynchronous and bounded. Here and in the rest of the paper, we ignore the presence of clients and focus on dispatchers, not only to simplify the treatment, but also because in the domains we target there is often no distinction between the two.

Figure 2.3 shows a dispatching graph where a dispatcher (the dark one) is subscribed[4] to a certain event pattern. The arrows represent the routes laid down according to this subscription, and reflect the content of the subscription tables of each dispatcher. Note how the graphs corresponding to routes

[4]More precisely, only clients can be subscribers. With some stretch of terminology, here and in the following we will say that a dispatcher is a subscriber if it has at least one client that is a subscriber.

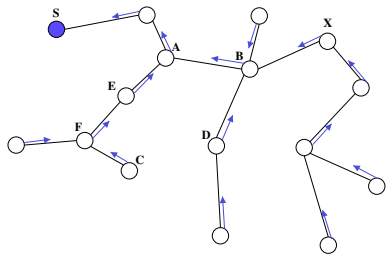

Figure 2.3: A undirected acyclic graph with subscriptions laid down according to a subscription forwarding scheme.

for the two separate subscriptions all insist on the same graph constituting the dispatching network. To avoid cluttering the figure, subscriptions are shown only for a single event pattern.

Chapter 3

Epidemic Algorithms

Epidemic algorithms are a new kind of distributed algorithms inspired by theory of epidemics. They have recently become popular as a solution to address scalable and reliable multicast dissemination(e.g., [1, 23, 15]). In these algorithms communication is achieved by trying to "infect" as many nodes as possible, with a measure of success defined only in probabilistic terms.

Although epidemic algorithms were originally developed to deal efficiently within the consistency management of replicated databases [13], they have been applied to date to a number of relevant problems, including dissemination of news through NNTP and multicast in ad hoc mobile networks [7, 24].

When applied to information dissemination, epidemics techniques are sometimes known as gossip techniques and this for obvious reasons; in this thesis the two terms will be treated as synonyms[1].

[1]We prefer gossip when the topic is relevant to computer science field and revert to epidemic terminology when the issue addressed can be better explained with reference to

In essence, gossip algorithms trade the strong reliability guarantees, typical of traditional deterministic approaches, for better scalability, achieved at the price of weaker guarantees defined only in probabilistic terms.

3.1 Basic Concepts

By and large, the idea underlying this family of algorithms is for each process to communicate periodically its knowledge about the system "state" to a random subset of other processes. The state of the system can differ from system to system: from data base replica to the history of messages appeared in the system so far. The state tend to global consistency through exchanges called "gossip rounds", during which processes update their local states.

A single gossip round consists of the following steps:

1. Process A chooses randomly another process to communicate with, say B.

2. A sends to B information that allows to determine the presence of inconsistencies (e.g., the identifiers of the messages, checkpoint hashes etc.).

3. A and B reconcile their state by exchanging information.

The *identity* of the processes that are contacted at each round is typically determined in probabilistic terms [2].

the epidemiological literature

[2]As a consequence, these algorithms are sometimes referred to also as *probabilistic* algorithms.

An aspect in which gossips algorithms differ is how many nodes are contacted at each round. This is usually a fixed parameter called fanout: a high fanout means contacting a bigger fraction of the processes, and hence increased reliability, but at the expense of increased overhead, and hence reduced scalability. In a sense, this parameter represents the knob that can be adjusted to obtain the desired tradeoff between reliability and scalability.

A peculiar case is obtained when a process contacts every node it knows of. This is actually a technique known as *flooding*, and can be considered as a degenerate form of gossip. Gossiping is especially preferable to flooding when in the presence of a clique or a well connected graph in which gossip has basically the same reliability as flooding with a fraction of the overhead. Some works in fact exploit gossip as an optimization of the flooding technique[21, 23].

3.2 Properties

The distinctive features of gossip algorithms are that their operations are very decentralized and driven by probabilistic decisions. Decentralization derives from the fact that the burden of information transmission is not placed in a single point of the network, be it the origin or some other well-known process, rather it is shared among multiple processes, and hence decentralized.

Moreover not only this class of algorithms is distributed but is also very fault tolerant at both global and local level:

- the system on the whole can withstand a very high number of faulty

nodes and network losses and still exhibit a correct behavior since the protocol is redundant in itself.

- each a node does not rely on the correct operation of any other specific node since the protocol behavior is probabilistic.

The probabilistic and decentralized nature of these algorithms gives them other desirable properties: gossip algorithms for example are very resilient to changes in the system configuration (e.g., topological changes) since they do not rely on the existence of one or more known processes. Moreover, these properties are preserved as the size of the system increases, thus leading to good scalability. Finally, these algorithms are very simple to implement and rather inexpensive to run.

From this short discussion, it is evident how gossip algorithms are a good match for the scenarios defined by mobile computing. Indeed, these scenarios are characterized by very little determinism themselves, both at application and system level. And already this approach has proven useful in a mobile ad-hoc networking scenario [7, 24].

3.2.1 Epidemics Flavors

Epidemic literature usually makes a distinction between two fundamental kind of models defining a disease spread. The first one is sometimes defined as "simple epidemics" while the second one is called "complex".

Simple epidemics subjects are allowed only two states: *susceptible* and *infective*. All individuals start in the former until one of them becomes

infective. As soon as a subject is "promoted" to the second state, it starts to spread the disease on its own and stays in this state for its whole life.

Complex epidemics, on the contrary, allows a third state to be reached after the *infective* one: this state is usually called *removed* (or *recovered*) and models that take in to account this possibility are also called SIR models.

If a *susceptible* or *infective* individual comes in contact with a *removed* one the interaction is fruitless in the sense that neither of the two individuals taking part in the exchange change their state.

In epidemiology this is useful for modeling situations where, after an amount of time since infection, the organism recovers from the disease, stops infecting others and gains immunity to the pathogen.

As the ultimate goal of epidemic techniques as applied to information dissemination is to spread the information to every node as quickly as possible, *removed* nodes at first, could seem not to be useful at all.

The point here is that *removed* nodes are not a desirable situation but an unavoidable one. In many systems each node stores the state of the whole system (e.g. replicated databases) or, at least, the state of the system for which each node is the "authority" (e.g. resource location). In this favorable scenario *simple epidemics* can be used: an infection is always available for further spread. In other systems unfortunately the total system state grows with time: this is the case for transmission systems and since buffers have finite length after some time inevitably messages must be discarded. *Removed* sites could also be used to model other causes for not being infective: for example in this thesis we used *removed* nodes to model a typical content-

based situation where nodes are not interested in "seeing" an event.

3.2.2 Epidemic guarantees

One of the fundamental results of *simple epidemics* shows that if one of the nodes becomes infected eventually all the population does.

This is very important property and can be translated into our domain stating that when a *simple epidemics* can be used that is sufficient to guarantee **Eventual Delivery**.

If *simple-epidemics* cannot be used there could be the possibility that a certain information (a pathogen in the epidemiological view) could be discarded from the system before every node sees it: this requires special care to avoid discarding it too early.

Even in scenarios where eventual delivery cannot be guaranteed one important property, called **Bimodal delivery**, can still be achieved: in such a situation, there is a high probability for a message to reach almost all processes in a group, a low probability to reach only a very small set of processes in the group, and a negligible probability to reach some intermediate number of processes in the group. In essence, the traditional "all or nothing" guarantee provided by deterministic protocols is weakened by gossip algorithms into "almost all or almost none" [1].

To understand intuitively why this happens we can start from the consideration that, in an infinite network, epidemic growth is exponential. The behavior in a finite network differs from the infinite one after a certain number of nodes are infected since it's more likely to contact already infected

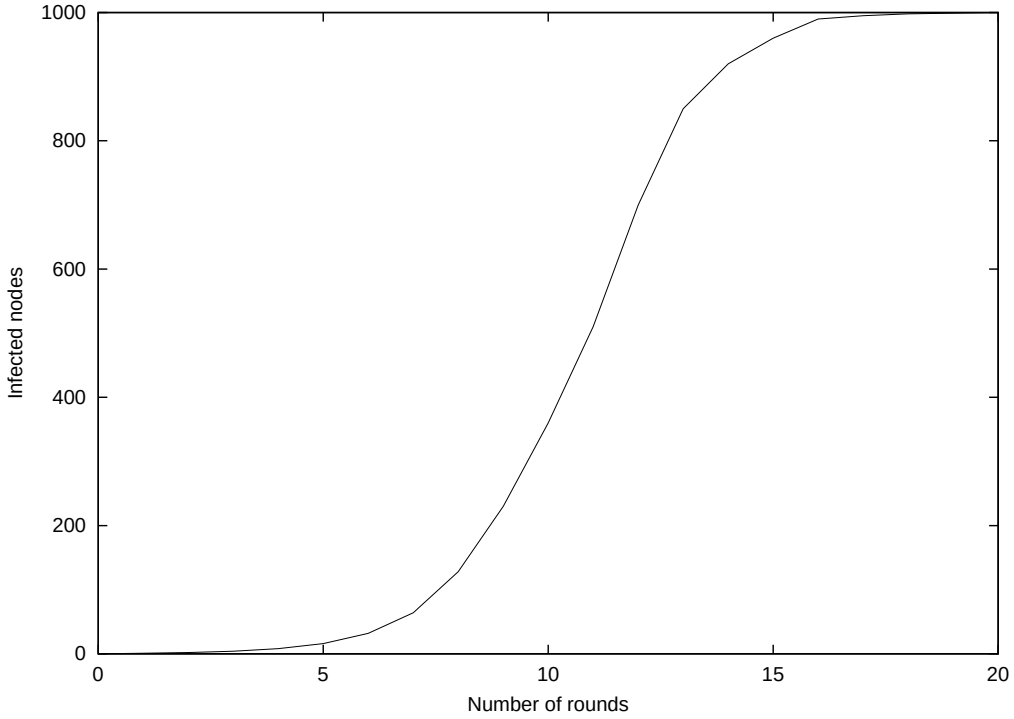

Figure 3.1: An example of the diffusion of an infection in a finite population.

nodes. A typical infection spread in a finite population is plotted in figure 3.1 with respect to the numbers of rounds.

Now consider what would happen if the spread stops at an arbitrary instant in time: if we pick this instant with a uniform distribution, since the slope of disease spread is rather steep, it is very unlikely that we chose one of the few instants that should provide a non-bimodal delivery.

3.3 Classification

In the following, we present those aspects which, we believe, are the most meaningful for a classification of epidemic algorithms.

3.3.1 Message Exchange (Push versus Pull)

The direction of information exchange is one of the most notable features, defining a gossip algorithm. The mode of message exchange whereby gossip senders (aka *gossipers*) are updated by gossip receivers with messages missing in the digest gossiped is commonly referred as *gossip-pull*. On the contrary, the term *gossip-push* indicates the dual mode of message exchange whereby every process in the system periodically gossips a digest of its received messages; gossip receivers can solicit such messages from the sender if they have not previously received them. Finally, the term *push/pull* refers to a mixed variant whereby two process symmetrically update each other.

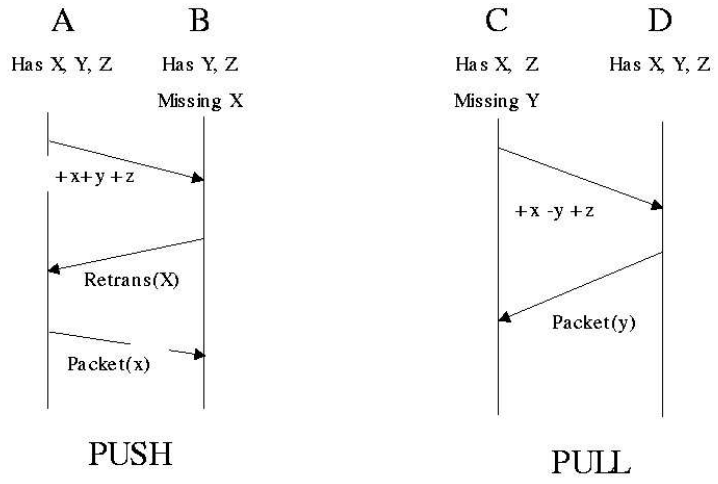

Figure 3.2: Gossip recovery examples:in (PUSH) gossiper $\mathcal{A}$ pushes out a missing packet $\mathcal{X}$ to $\mathcal{B}$. In (PULL) gossiper $\mathcal{C}$ pulls in a missing packet $\mathcal{Y}$ from $\mathcal{D}$

Figure 3.2 illustrates the two forms of gossip initiated retransmission.

Demers et al. in [13] showed that a pull approach converges faster than

push, provided that a majority of the participants have the requested message. This is easily explained by their simple deterministic model. Let p_i be the probability of a process to remain susceptible for an event received after the i^{th} gossip round. With a pull strategy, a process remains susceptible after $i+1$ rounds if it was susceptible after round i and it contacted a susceptible process during round $i+1$ (whose probability to remain such was again p_i). Thus, we obtain the recurrence

$$p_{i+1} = (p_i)^2 \tag{3.1}$$

which converges very rapidly to 0 when p_i is small. For push, a process remains susceptible to a given event after round $i+1$ round if it was susceptible after round i and no infective process chose to contact it in round $i+1$. Thus, the analogous recurrence relation for the push alternative is

$$p_{i+1} = p_i \left(1 - \frac{1}{n}\right)^{n(1-p_i)} \tag{3.2}$$

where n is the size of the group. (3.2) also converges to 0 but much less rapidly (see figure3.3 , since for every small p_i (and large n) it is approximately

$$p_{i+1} = p_i e^{-1} \tag{3.3}$$

The difference in the speed of convergence can be understood intuitively by considering a scenario where a multicast reaches all the receivers but one. In this case, the pull strategy allows a receiver to immediately recover the missing message instead of waiting to be pushed, thus improving packet delivery latency.

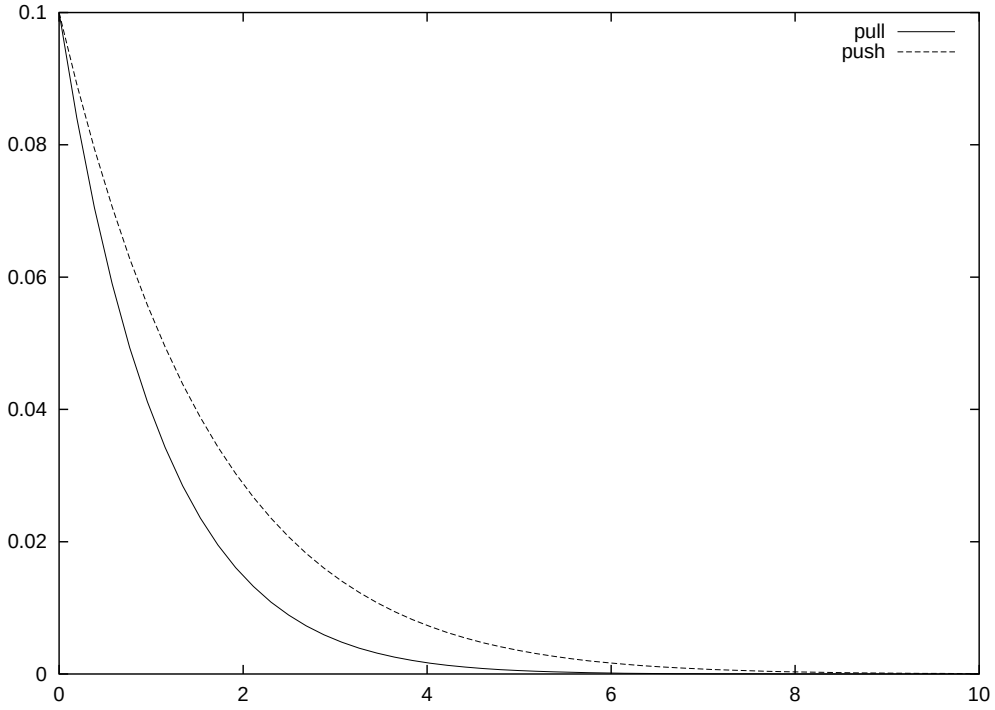

Figure 3.3: The convergence of *push* and *pull* approach

3.3.2 Positive versus Negative Digest

Another dimension along which gossip algorithms differ is the scheme used for disseminating the information about the process state, which can exploit either *positive* or *negative* gossip digests. In the former scheme, each gossip message sent by a process contains the state of communication as it perceives it, e.g., the content of the process' event history. Hence, the gossip message lists all the events that the process has *received* lately. In a negative gossip scheme, instead, the gossip message contains the events that the process has *missed*. Note that listing missing messages will usually be smaller than describing the entire buffer, especially under high send rates.

While in principle the style of communication, push vs. pull, and the information dissemination scheme, positive vs. negative, are orthogonal, in practice pull/negative and push/positive are the most meaningful and those typically exploited. In presence of a pull strategy, a negative scheme can be naturally used to react to a missing message by "pulling" it from other processes, while the positive scheme is best employed to proactively push a process' state to the rest of the system.

3.3.3 Proactivity versus Reactivity

This last remark highlights a key difference between the two solutions we are considering, and one that impacts their performance and applicability. The pull/negative strategy is intrinsically reactive, in that it is triggered only when a process realizes it has lost a message event; otherwise, no action is necessary. Instead, push/positive must be implemented according to a proactive scheme, where gossip takes place periodically. Typically, the high degree of reactivity provided by the first approach is preferable, in that it reduces the overall latency experienced by a message. Nevertheless, in some scenarios, e.g., those characterized by a low rate of communication, a process may not realize it has missed a message until the next one is received. In these scenarios, the other approach may be preferable.

3.3.4 Membership View

A key element in designing a gossip algorithm is the amount of information on the system at disposal of the peer since it defines the set of available peers

to gossip with. We can distinguish in:

- Global
- Local
- Hierarchical
- Partial Randomized
- Anonymous

Global

Each peer has an (almost) complete knowledge of all other nodes. This is the simplest mechanism because it allows the peer to gossip with the whole system. However this could represent a great obstacle to the scalability of the system, because the memory requirement on each peer can grow indiscriminately. Nevertheless in some specific application, such as failure detection[33], it is inherent in the goal of the application to have a global view.

Local

Each peer has knowledge only of its neighbors. This strategy allows to save memory but may limit the speed of the infection because a peer can not infect far nodes.

In some scenarios, however, this can be the only view available: in a ad hoc network it is too expensive to keep trace of the whole system.

Hierarchical

Each peer has a precise view of its immediate neighbors, while its knowledge becomes less exhaustive at increasing distance. This approach can be very useful, for instance, in Resource Location[42] whereby a tree-based structure is used to represent the system: the leaves contain directly measured information (*precise view*) whereas internal nodes are generated by aggregating information of children.

Partial randomized

The view of every individual member consists in a uniform random process list.

The advantage of partial view, introduced for the first time by Eugester et al. in [15], is to improve the scalability of the system. In fact, as with global view, a peer can contact both neighbors and far nodes (and this improves the speed of spread) but the memory requirements are significantly lower. To ensure a uniform distribution of membership knowledge, every gossip message piggybacks a set of peer identifiers to update views.

Anonymous

Peers do not have to know *a priori* the members they are gossiping with. Though a peer has neither global nor local view, it can gossip with other members because its gossip messages are routed by the underlying structure, e.g. a multicast tree[7].

This view well-adapts to those scenarios, such as low density rapid-

changing ad hoc networks, where changes in system topology occur very frequently making it very expensive to keep an up to date view of the system.

3.3.5 Member Selection

Member selection defines the policy in the selection of a member to gossip with. It is worth to underlining that member view bears as a completely orthogonal dimension, although some couples are more meaningful and hence more used than others.

We can differentiate between:

- Uniform
- Distance-based
- Hierarchical
- Connection-based

Uniform

Peers are selected from the view according to a uniform distribution. This is the simplest way of selection and nevertheless it is often used because ensures good performance in that a peer can infect uniformly all the peers in its view.

However, as a drawback, this strategy could overload routers because it does not take into account any topological information.

Distance based

Near peers are chosen with higher probability than distant ones. This reduces traffic based on proximity, however gossiping with a distant peer is extremely important because it improves the spread of the epidemy.

Furthermore, in a multicast scenario, where gossip algorithms are employed as recovery mechanism, it is of paramount importance to reach distant nodes because message loss could affect an entire locality. For instance in [7], the messages are steered on the tree with higher probability towards nearest nodes and each member has a certain probability to respond to the message thus stopping its propagation.

Hierarchical

Most gossip exchanges are in the local domain and the gossip exchange rate decreases with the distance between any peers in the tree. Although this strategy seems to well-adapt to a hierarchical view, often it is used in conjunction with other views. In [33] all of peers share a global view but they adopt a hierarchical member selection in order not to overload the bridges between physical networks.

Connection-based

A peer selects with higher probability less connected peers. Note that, in contrast to Hierarchical selection, this technique is adopted not to reduce traffic but to improve reliability in a bad-connected graph. This allows nodes scarcely connected to the rest of the system to be contacted as well as the

other nodes.

In [23], the connection based selection is achieved by the introduction of the notion of *weight*. *Weight* represents the minimum number of edges that must be removed for the peer to become disconnected from its neighbors. A peer floods to neighbors with small weights and gossips with the others.

3.4 Applications Areas

In this section we provide an overview of the many contributions produced by epidemic algorithms application in different IT contexts.

3.4.1 Replicated DataBase Maintenance

"Epidemic Algorithms for replicated Database Maintenance" [13] by Demers et al. is the first works applying epidemics theory to the computer science field. The goal is maintaining consistency between several copies of a database replicated at several sites.

The paper at first presents two method for maintaining data base consistency already employed by the Grapevine project [2]: the first, simple, deterministic method used to propagate updates is "direct mail" as exemplified in figure 3.4: every node experiencing a change in its local database state tries to notify it to every other site. As "direct mail" could fail to contact every site a second mechanism called "anti-entropy"(fig. 3.5) is used to maintain consistency between copies. It is run during offpeak hours and

Symbol legend:

- $s \in S$ a site hosting a replica of the database.
- s.ValueOf $\in (v \in V, t \in T)$ a time varying partial function specifying the timestamp and the value of the key (without loss of generality we suppose a single key in the database)

Invoked at a site s when an update occurs.

DirectMail ()

for all $s' \in S$ **do**

PostMail[to : s',msg : ("Update", s.ValueOf)]

end for

Invoked at a site upon receipt of an update message.

HandleMail ("Update",(v, t))

if s.ValueOf.$t < t$ **then**

s.ValueOf $\leftarrow (v, t)$

end if

Figure 3.4: Direct mail algorithm.

Symbol legend:

- $s \in S$ a site hosting a replica of the database.
- s.ValueOf $\in (v \in V, t \in T)$ a time varying partial function specifying the timestamp and the value of the key (without loss of generality we suppose a single key in the database)

Invoked at a site at periodic intervals.

AntiEntropy ()

for some $s' \in S$ **do**

ResolveDifference[s,s']

end for

ResolveDifference in a pull variant.

ResolveDifference (s, s')

if s.ValueOf.$t < s'$.ValueOf.t **then**

s.ValueOf $\leftarrow s'$.ValueOf

end if

ResolveDifference in a push variant.

ResolveDifference (s, s')

if s.ValueOf.$t > s'$.ValueOf.t **then**

s'.ValueOf $\leftarrow s$.ValueOf

end if

ResolveDifference in a pull-push variant.

ResolveDifference (s, s')

if s.ValueOf.$t < s'$.ValueOf.t **then**

s.ValueOf $\leftarrow s'$.ValueOf

else if s.ValueOf.$t > s'$.ValueOf.t **then**

s'.ValueOf $\leftarrow s$.ValueOf

end if

Figure 3.5: Anti-entropy algorithm.

```
Per dispatcher information:
  - s ∈ S a site hosting a replica of the database.
  - s.ValueOf ∈ (v ∈ V, t ∈ T) a time varying partial function specifying the timestamp
    and the value of the key (without loss of generality we suppose a single key in the database)
  - T_int interval between RumorMongering rounds
  - RedundanceFactor a constant specifying the tradeoff between network traffic and reliability

Invoked at a site s when an update occurs. (push)
RumorMongering ()
  k ← RedundanceFactor
  while not stopCondition do
    choose s ∈ S
    alreadyKnown = SendMessage[to : s',msg : ("Update", s.ValueOf)]
    if blind OR (feedback AND alreadyKnown) then
      if counter then
        k ← k − 1
        if k = 0 then
          stopCondition ← TRUE
        end if
      else
        with probability 1/k: stopCondition ← TRUE
      end if
    end if
    sleep(T_int)
  end while
```

Figure 3.6: Rumor Mongering algorithm.

basically consists of each site choosing a number of sites to synchronize to, then comparing its database with theirs to resolve differences.

In [13] it is noted that "anti-entropy" is indeed a *simple epidemics*: updates are never discarded from the system as the local copy of the database contains the whole state of the system as perceived by the local site. Thus anti-entropy guarantees that every update eventually becomes known at all sites.

Then a second epidemic algorithm is presented as an alternative to trivial direct-mail. It consists of a *complex epidemic* technique called "Rumor mongering": whenever a site observes a change in a value of the database, either because modified locally or because notified, it treats it as an hot rumor thus shared during the following rumor mongering rounds, until a stop condition is met. In the algorithms four possible stop conditions are explored: they can be viewed as the possible combinations of two values on two orthogonal aspects: either blind or feedback and either counter vs coin. If a feedback condition is adopted the site makes a decision to spread the update further or not whenever the other site has already seen it, while with a blind condition, it makes a decision every time it spreads the update. If a counter condition is used a site stops the spread after a constant number of "decisions", if a coin condition is used it does so on the basis of a fixed probability.

Among the other parameters explored in this work there is for example the direction of the information flow either pull, push, or both.

Another issue in such a synchronization method is how to handle deletion of keys from the database as a deletion will not spread to other sites, the

opposite would happen instead: updates form other sites will recreate the deleted key.

The issue is resolved through the use of "death certificates" which contain a key and a timestamp. While they are sent over the network they remove every copy of the same key with an older timestamp. Then the question of how to delete death certificates arises, a simple solution could be to keep this certificates for a fixed amount of time (e.g. 30 days) to be sure that every site has seen the certificate. To further increase this threshold a death certificate could be retained at some sites in a "dormant" state. In a way analogous to an immune reaction if a dormant death certificates comes in contact with an update older than its timestamp it is "awakened" ad spread again in the network.

3.4.2 Failure Detection

In [33] van Renesse et al. gossip is employed to develop a Failure Detection Service that does scale well and provides timely detection.

In their protocol every member gossips to figure out who else is still gossiping. Each member maintains a list in which a row is made up of:

- the symbolic name of each member
- its IP address
- an *heartbeat counter* to be used for failure detection
- the last time that its corresponding heartbeat counter has increased

Furthermore every member has an its own heartbeat counter. Every T_{gossip} seconds, each member increments its own heartbeat counter, chooses randomly a member from the list and sends its own list of non-failed members. When a member receives such a *gossip message*, it merges the list received with its own list, and keeps the maximum heartbeat counter for each member. Occasionally a member broadcasts its list in order to be located initially and also to recover from network partitions.

If the heartbeat counter has not increased for more than T_{fail} seconds, then the member is considered failed. T_{fail} has to be set appropriately so that the probability of mistake is very low.

After a member detects a faulty member, it cannot immediately remove the relative row. This is in consequence of the fact that not all members will realize the crash at the same time and thus a member $\mathcal{A}$ may still receive an older heartbeat about a member $\mathcal{B}$ that it has considered faulty. If $\mathcal{A}$ should have already removed $\mathcal{B}$ row, it would consider $\mathcal{B}$ as a new member, joining the group.

Therefore, the failure detector does not remove a member from its membership list until after $T_{cleanup}$ seconds ($T_{cleanup} > T_{fail}$). $T_{cleanup}$ should be tuned so that row removals of a failed member start only after all members agree on its failure.

Note that this protocol only detects hosts that become entirely unreachable. It does not detect link failures between hosts.

To allow the protocol to scale well over the Internet, members can detect the bound of Internet domains and subnets, and thus avoid overloading the

bridges between physical networks. With this goal in mind the algorithms adopt a hierarchical member selection: gossips are mostly done within subnets, with few gossips going between subnets, and even fewer between domains. Within subnets, the above protocol is employed with no changes, whereas for gossips that cross subnets and domain, a modified version is needed. In particular, this modified protocol varies the probability of gossip so that for every round, on average one member per subnet will gossip to another subnet, and one member per domain will gossip to another domain. So the cross subnet bandwidth in a given domain will depend only on the number of subnets in that domain.

To achieve that, every member tosses a weighted coin every time it gossips. One out of n times, where n is the size of the subnet, it picks at random another subnet in its domain, and within that subnet, a random peer to gossip with. The member then tosses another weighted coin with probability $\frac{1}{n \times m}$ where m is the number of subnets in its domain, it picks a random domain, then a random subnet within that domain and finally a random peer within that subnet to gossip with.

This mechanism allows a significant reduction in the amount of gossip messages flowing over the Internet routers.

3.4.3 Resource Location

In [42], the author presents *Captain Cook*[3], a gossip-based resource location protocol for the Internet. Such a protocol has a tree-based hierarchical view of all the collected information. The leaves in the tree contain directly measured resource information, while internal nodes are generated using *condensation functions* that aggregate information in child nodes.

Differently from traditional resource location service, *Captain Cook* does not require the presence of centralized servers. Instead, every machine on the network contributes in maintaining the entire service.

Monitoring information is stored in one or more trees. Each leaf node represents a particular machine, whereas an interior node is generated by a so-called *condensation function* which produces a Management Information Base (MIB) for a collection of child nodes, representing a *domain* of participating machines. For each child domain, there is a row containing such information as average load on the machines in the domain, or the presence of a particular resource, as shown in figure 3.7.

The tree is distributed over the participating machine with MIB automatically replicated on several machines. Gossip messages are employed to guarantee that updates eventually propagate to the whole tree.

Each row is timestamped with the clock time of its last update. Periodically, each machines chooses a random row in its local tables and using the *Contacts* information stored in that row sends a gossip message. The gossip

[3]The name reminds the famed explorer who mapped the world before being clubbed to death.

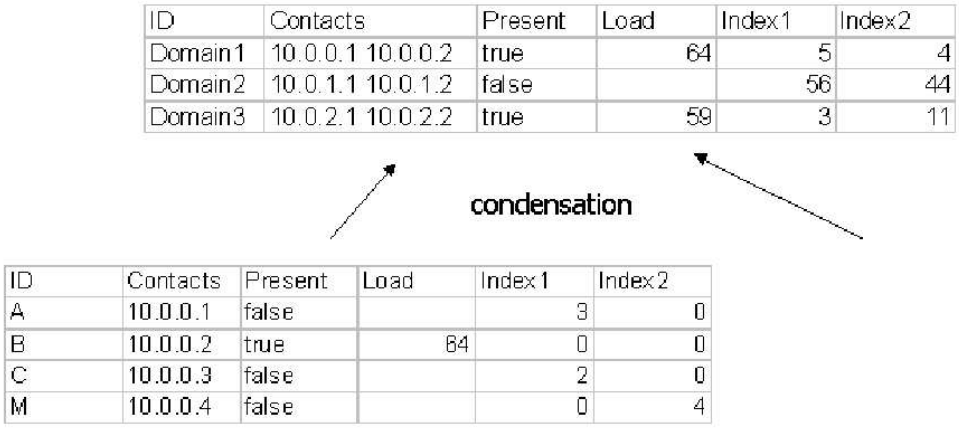

Figure 3.7: This figure shows part of two level tree used by *Captain Cook*

message contains a list of (MIB ID, timestamp) pair for each row stored at the gossiper.

The receiving machine compares this list to the one it has stored locally and using the timestamp[4] it can decide which of the MIBs it has are more up-to-date than the gossiper's. It then returns an update to the gossiper, by means of a *push-pull* message exchange.

If the machines happens to pick itself as a contact, it repeats the process for the parent domain and so on. This strategy ensures that most gossip exchanges occur in the local domains, and that gossip exchange rate decreases exponentially with the distance between any two participating machines in the tree.

To address the frequent changes in membership, typical in large distributed application where machines join and leave at very high rate, a mem-

[4]The author supposes that machines have loosely synchronized clock.

bership mechanism is proposed.

First of all, it is worth underlining that in *Captain Cook*, keeping track of membership is easier than in failure detection, because the granularity of information degrades with the distance in the tree. That is, participants know the membership in their local domain, but typically only how many members there are in the other domain.

The mechanism of failure detection in *Captain Cook* is rather simple. A "clock" value is added to each row in the MIB table. Machines update this value in their local table each time they send a gossip message. For internal nodes, the "Clock" value is computed by taking the median time in the input table. When a participant notices that a clock value is slow by a predetermined value T_{fail}, it removes the row from the table.

As final remark, the present work is one of the few exploiting epidemic algorithms to address security issues. Actually, gossip is interesting from the perspective of security. On one hand, it is hard for an adversary to stop the flow of updates to provoke denial-of-service. On the other, it is easy to introduce and spread invalid update of existing MIBs or generate bogus MIBs. Note that the goal of the authors is to ensure the integrity of data, not to guarantee confidentiality.

The basic idea is to replace each MIB ID by a public key certificate. The private key is given to all machines storing that MIB and they use it to sign all the updates of the MIB. Upon receipt, every machine checks the integrity of the update by decrypting it via the public key; if the operation is not successful it discards the update.

In addition to using cryptographic techniques, condensation functions are designed so that they are invulnerable to a small percentage of incorrect input. For example, rather than the average load, the median load is preferable, in that it eliminates outliers potentially generated by malicious participants.

3.4.4 Multicast

Multicast dissemination has been the most fertile area where epidemic algorithms have found application. They have been employed to recover lost message[1], to disseminate messages[23] or as both dissemination and recovery mechanism[15].

Here we present the main works exploiting epidemics technique to address reliable dissemination.

Bimodal Multicast

With *Bimodal Multicast* (*pbcast*[1]), Birman et al. have renewed the interest in gossip-based algorithms. *Pbcast* relies on two phases. A "classical" best-effort multicast protocol (e.g. IP Multicast) is used for a first rough dissemination of messages. A second phase ensures reliability with a certain probability by using a gossip-based retransmission: every participant in the system periodically, according to *push* scheme, gossips a digest of its buffered messages, and gossip receivers can solicit such messages from the sender it they have not received them previously.

Every participant sends a gossip message per round and on average it receives a gossip messages per round. Every participant has a global view of

the system and selects a node with an uniform distribution. Actually, in [1] details on membership maintenance are not dealt with but the author refers to *Captain Cook*[42] as a possible way to address membership consistency.

The gossip approach allows *pbcast* to reach an high degree of scalability, keeping a stable throughput.

As most gossip algorithms, however, *pbcast* can guarantee only a *bimodal delivery* (see 3.2.2). In order to offer a complete guarantee of delivery, *Reliable Probabilistic Multicast* (rpbcast[37]) adds a deterministic third phase to the *pbcast* algorithm, in which centralized loggers are contacted if the second gossip-based phase fails.

Lightweight Probabilistic Broadcast

Lightweight probabilistic broadcast (lpbcast[15]) is a probabilistic broadcast algorithm developed with the twofold goal of delivering new messages and recovering lost ones.

Lpbcast adds an inherent notion of memory consumption scalability to the notion of network consumption scalability primarily targeted by gossip-based algorithms. In contrast to deterministic approach such as hierarchical or global view, *lpbcast* introduces a new probabilistic approach to membership; each participant has a *random partial* view of the system. This allows to consume little resources in terms of memory and requires no dedicated messages for membership management. Its scalability is enforced because a peer only knows a fixed number of processes, and fault-tolerance is achieved since each process is known by several processes.

Gossips are used to disseminate the payload (i.e. events) and to propagate digests of received events, but also to propagate membership information.

Every gossip message contains:

Notifications A message piggybacks notification received for the first time since the last gossip round such that every notification is gossiped at most once by each dispatcher. Older notifications are stored in a buffer to satisfy retransmission requests.

Notifications identifiers Each message also carries a digest of the notifications buffered at the gossiper. The gossip receiver checks whether it is missing some of those notifications and in case, it sends a retransmission request to the gossiper.

Unsubscriptions A gossip message also piggybacks a subset of unsubscriptions. This type of information enables the graduate removal of members which have unsubscribed from local views.

Subscriptions A set of subscriptions is attached to the message to update local views with new members.

Upon receipt of a gossip message, a peer executes the following operations:

1. The unsubscriptions are handled, by removing corresponding nodes from the local view.

2. The subscriptions, not yet received, are added to the local view

3. Notifications received for the first time are delivered to the application and become eligible for being forwarded with the next gossip message. Missing notifications are requested to the gossiper, by comparing notification IDs in the gossip message and local list of received notifications.

As last remark,the proposed membership approach is not to limited to the use with *lpbcast* algorithm, but can be put to work easily with other algorithms as is diffusely explained in [18].

3.4.5 Routing in Ad Hoc Networks

Gossip, for its distributed nature and for its high resilience to faults, well-adapts to highly dynamic environments such as ad hoc networks. In particular gossip has been proposed to address two different kinds of issues. The first paper[19] exploits gossip to reduce the overhead of many *ad hoc* routing protocols. The second[41] focuses instead on using gossip to enable message delivery when a network partition exists at the time the message is originated.

Gossip-Based Ad Hoc Routing

In ad hoc networks, the power supply of individual nodes is limited, wireless bandwidth is limited, and the channel conditions can greatly vary. Furthermore since nodes are mobile, route changes are very frequent. Many ad hoc routing protocols have been developed, but most of them are based on flooding and, despite the optimization, many messages are propagated unnecessarily.

Actually this paper focuses on "Ad-hoc on-demand distance vector routing" (AODV[29]) but the conclusion holds for other flooding-based algorithms as well. The authors show that adding their gossip protocol to AODV allows to significantly reduce the amount of route request and improves network performance in terms of end-to-end latency and throughput.

The approach adopted to employ gossip algorithms is different from the traditional ones. Usually, gossiping proceeds by choosing some set of nodes at random to gossip with. In this context, instead there is no availability of stable paths to the nodes and hence such an approach is not suitable at all.

In ad hoc network, if a message is transmitted by a node, due to the nature of radio communications, the message is received by all the nodes one hop away from the sender. The authors of the paper take advantage of this physical-layer broadcasting, by controlling the probability with which this physical-layer broadcast is sent. In other words they use probability not to choose which peers to gossip with but to choose whether to gossip or not.

A source sends a route request with probability 1. When a node first receives a route request, with probability p it broadcasts the request to its neighbors and with probability $1 - p$ it discards the request. If the node receives the same route request again, it is discarded. The authors, through simulations, show that in such a way it is possible to reduce by about 35% messages the load of AODV, obtaining the same quality of route discovery.

Epidemic Routing for Partially-Connected Ad Hoc Networks

Existing ad hoc routing protocols, while robust to rapidly changing network topology, assume the presence of a connected path from source to destination. In [41] techniques are developed to deliver messages in case there is never a connected path from source to destination.

The basic idea is to distribute application messages to hosts, called *carriers*, within connected portions of ad hoc networking. Epidemic Routing is based upon these carriers coming in contact with other carriers of another connected portion of networks through node mobility. Through such a dissemination of information, there is an high probability of eventually reaching the destination. Figure 3.8 depicts the above behavior.

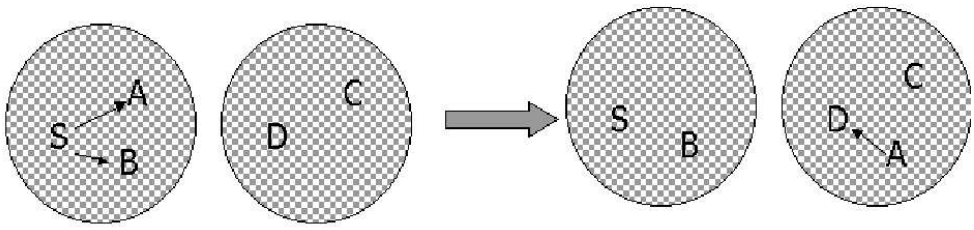

Figure 3.8: An example of message delivery in presence of network partition

In Figure 3.8, a source, $\mathcal{S}$, wishes to send a message to a destination, $\mathcal{D}$, but no connected path is available from $\mathcal{S}$ to $\mathcal{D}$. $\mathcal{S}$ transmits its messages to its two neighbors, $\mathcal{A}$ and $\mathcal{B}$, within direct communication range. At some later time, $\mathcal{A}$ comes into direct communication range with $\mathcal{D}$ and finally sends the message to its destination.

Gossiping is used to minimize the number of carriers, needed to deliver

the message. When two hosts come into communication range of one another, the host with the smaller identifier starts a *push-pull* session to update each other. The use of a "Bloom filter"[3] is proposed to reduce the overhead associated to gossip message.

Optimizations are introduced to limit the spread of the message to the whole network by limiting the number of hops made by messages during the dissemination phase.

Results show that Epidemic Routing is able to deliver nearly all messages in scenarios where existing ad hoc routing protocols fail to deliver any message, assuming enough per-node buffering to store between 10-25% of the messages originated in the scenario. Of course, there is an inherent tradeoff between aggregate resource consumption and message delivery rate/latency.

Chapter 4

Gossip and Content-Based

The application of epidemic algorithms to the case of content-based publish-subscribe system is not straightforward. Content-based systems pose peculiar challenges that have not been tackled thus far by the research community, which at best has concentrated on the simpler subject-based publish-subscribe systems. Still, the synergy between the two approaches is worth investigating, since a content-based approach enhances the underlying publish-subscribe middleware with unprecedented levels of flexibility, which in turn simplify significantly the programmer's task.

In this chapter, we analyze in more detail the problems that make the exploitation of gossip algorithms more challenging when content-based systems are at stake.

4.1 Challenges

As already observed in 3.3.5, a key issue in designing a gossip algorithm is how to determine the processes to gossip with. When gossip is applied to multicast protocols or subject-based publish-subscribe systems this issue has a trivial solution based on the notion of group (or subject) these systems define explicitly. In the case of a push approach a different gossip message can be produced periodically for each group and sent to the processes of the same group. Similarly, in the case of a pull approach the most reasonable solution to retrieve a lost message m is to contact the processes that are part of the group m belongs to.

Unfortunately, this approach cannot be applied in the case of content-based publish-subscribe systems, since they do not rely on any *explicit* notion of group to route messages to their recipients. In these systems, routing is entirely based on the message content: hence, a single message can match different patterns, and consequently reach different subscribers. Moreover, in general the entire message content must be matched against a pattern, while in subject-based schemes only the subject is considered.

As a consequence of these observations, we can identify three key issues that must be dealt with when applying gossip algorithms to content-based publish-subscribe:

How to route the gossip messages. As mentioned above, determining how to route gossip messages, i.e., choosing which other nodes to gossip with is critical in content-based publish-subscribe. In particular, when

a push strategy is adopted gossip messages include only a (usually positive) digest of messages, and hence the standard matching mechanism employed for routing cannot be applied to them to reach potentially interested nodes. The pull case is different and even more complex since in this case the gossiper is interested in retrieving lost messages and, by definition, their content is not available for matching.

How to determine that some event has been lost. To adopt a reactive approach it is crucial to determine if some message has been lost. This apparently simple task has not a trivial solution in content-based publish-subscribe. In fact, differently from subject-based systems, when content-based routing is applied, a node N_1 will not necessarily receive every message generated by another node N_2 under a given subject, but only those matching N_1's patterns. As a consequence, N_1 cannot employ the easy solution of logging the sequence number of the last message received for each message source and subject to determine if some message has been lost, since it has no way to distinguish whether a message coming from N_2 and whose sequence number has been skipped must be considered as lost or simply not relevant to N_1's patterns.

How to determine which events have been lost. If a negative scheme must be adopted it is fundamental to determine not only if some message has been lost, but also exactly which ones have. Again, in the case of a content-based system this problem does not allow the easy solution based on message sequence numbers, that is instead adopted

by multicast or subject-based routing.

4.2 A simple (but Inefficient) Solution

In an effort to keep things as simple as possible, it could seem reasonable to solve the first of the aforementioned issues by using a "blind" approach, where the set of nodes to gossip with is drawn randomly from the overall set of nodes. Unfortunately, this solution exhibits poor performance, both in its push and pull variants. With a pull approach, the convergence rate, given by equation (3.1), must be modified to consider the probability r of contacting a node that is not subscribed to any pattern matching the messages that must be retrieved:

$$\begin{aligned} p_{i+1} &= \mathbb{P}(\text{susceptible at } i+1) \\ &= \mathbb{P}(\text{contacted node is not infected}) * \mathbb{P}(\text{susceptible at } i) \\ &= (p_i + r) * p_i \end{aligned}$$

In case of gossip-push, the convergence rate is obtained by modifying the push equation (3.2) as follows:

$$\begin{aligned} p_{i+1} &= \mathbb{P}(\text{susceptible at } i+1) \\ &= \mathbb{P}(\text{susceptible at } i) * \mathbb{P}(\text{not chosen by any infective node}) \\ &= p_i * \left(1 - \frac{1}{N}\right)^{N - Np_i - Nr} \end{aligned}$$

As N grows it becomes:

$$p_{i+1} = p_i * e^{p_i + r - 1} = p_i * e^{p_i - 1} * e^{r}$$

These formulas show that a large number of nodes not subscribed to any pattern matching the messages that have to be retrieved (i.e., a large r) slow down the convergence considerably thus badly hurting the recovery of messages with a low number of recipients. Observe that this situation, i.e., the presence of a low number of recipients for each message, is common in publish-subscribe systems since it justifies the overhead of more complex routing schemes instead of the trivial flooding approach [8, 27].

Hence, more sophisticated solutions to the problem of how to route gossip messages are needed. These are presented in the next chapter, together with our solutions to the other aforementioned challenges.

Chapter 5

Gossip Algorithms for Reliability

In the previous chapter we analyzed the specific challenges content-based routing poses on gossip algorithms. In this section, we focus on deriving a solution to such challenges. In Section 5.1, we show an algorithm that uses proactive gossip push with positive digests. Then, in Section 5.2 and 5.3 we show two algortihms using reactive pull with negative digests.

In all the solutions we present in this section, every dispatcher periodically initiates a new round of gossip by performing the operations described by an action startGossipRound, whose behavior is shown in the description of the algorithm. In such description, however, we omitted the operations concerning the setting and triggering of the *gossip interval* T separating two gossip rounds, as their semantics is trivial.

```
Per dispatcher information:
  - buffer Cache holding a copy of the last events received

Invoked periodically, e.g., after timeout expiration.
Triggers the start of a new gossip round for a pattern in the subscription table.
startGossipRound ()
  choose a pattern p from the subscription table
  create digest = ∅
  for all event e ∈ Cache do
    if matches(e, p) then
      insert e.id in digest
    end if
  end for
  create gossipMsg = (self, p, digest)
  send gossipMsg towards one or more subscribers for p
```

Figure 5.1: Push algorithm(I).

5.1 Push

To provide an answer to the questions identified in Chapter 4 in the case of proactive push with positive digests we observe that, with this strategy, a gossip message sent by a dispatcher should include information about the set of events it cached. Moreover, this gossip message should be sent only to dispatchers subscribed to such events. As we already discussed, in content-based publish-subscribe systems this set of subscribers cannot be computed once for all. Nevertheless, we can leverage off of the fact that every dispatcher that received and cached an event e knows, from its subscription table, all the patterns matching e. This means that each dispatcher is able to construct a gossip message which includes a digest of all the cached events matching a

```
Per dispatcher information:
  - buffer Cache holding a copy of the last events received

Invoked on a dispatcher upon receipt of a gossip message.
handleGossipMsg (gossipMsg)
  if self is subscribed to gossipMsg.pattern then
    create a new reqMsg = ∅
    for all id ∈ gossipMsg.digest do
      if ¬isReceived(id) then
        insert id in reqMsg
      end if
    end for
    if reqMsg ≠ ∅ then
      send reqMsg to gossipMsg.initiator
    end if
  end if
  with probability P_forward send gossipMsg towards one or more subscribers for gossipMsg.pattern

Invoked on the gossip initiator when a request for a missing event is received.
handleReqMsg (reqMsg)
  for all id ∈ reqMsg do
    if ∃e ∈ Cache | e.id = id then
      send e to the sender of reqMsg
    end if
  end for
```

Figure 5.2: Push algorithm(II).

given pattern p. This gossip message can then be labelled with p and routed similarly to events matching p.

When startGossipRound is invoked by the dispatcher acting as the gossip initiator (or *gossiper*), a pattern p is chosen according to some strategy (e.g., randomly) from the dispatcher's subscription table and a digest is con-

structed which includes the (globally unique) identifiers[1] of all the cached events matching p. The gossip message *gossipMsg*, is then labelled with the pattern p and propagated along the dispatching graph. Routing of *gossipMsg* message outside the gossiper and along the way towards a subscriber is determined in the usual way, by looking at the subscription table to find neighbors interested in the pattern p. Nevertheless, it is worth noting that in the case of gossip, differently from the normal operation of a publish-subscribe system, the gossip message is not necessarily *duplicated* on all the outgoing routes towards subscribers, like in the case of events. Instead, it is sent only to a subset of the potential recipients, e.g., using a random subset of the routes available. The extent of propagation is determined by the probability $P_{forward}$, in order to limit overhead.

Also, note how in a traditional push-based approach every node gossips only with nodes sharing the same interests. A similar behavior can be obtained in a content-based publish-subscribe system by limiting the choice of p to the patterns in the subscription table which correspond only to subscriptions sent by the clients attached to dispatcher. Nevertheless, this would be detrimental, in that it would limit the number of nodes to gossip with and it would disregard the fact that in such systems a single event may match several patterns and thus reach different sets of subscribers. For this reason, in our solution p is drawn by considering the whole subscription table, i.e., from all the patterns known to the dispatcher. This increases the chances to

[1] A straightforward implementation of this identifier can be the source identifier and a monotonically increasing sequence number associated to the source.

eventually find all the dispatchers interested in the cached events and, with respect to the trivial solution sketched in Chapter 4.2, speeds up convergence and exhibits a lower overhead.

Upon receipt of *gossipMsg*, a dispatcher is expected to perform the operations represented by the action handleGossipMsg. These consist of checking whether the dispatcher is subscribed to the pattern p labelling *gossipMsg* and, if so, of verifying whether all the identifiers contained in the digest correspond to events that have already been received. In our solution, the details of how this test is performed are glossed over, and encapsulated in a function isReceived (id), which returns *true* if the dispatcher received an event with the given id.

The identifiers of all the missed events, if any, are then included in a request message *reqMsg* which is sent back to the gossiper. Upon receipt of this message, the gossiper selects the events with the corresponding identifier from the cache, and sends them back to the requester. This third and last phase concludes the interaction taking place in our push approach.

5.2 Pull

In some situations, as discussed in 3.3.1, a proactive push approach may converge slowly or results in unnecessary traffic. In these cases, a pull approach may be preferable.

When a reactive pull with negative digests is used, things become slightly more complicated, as we discussed in Chapter 4. In fact, in a content-based

Per dispatcher information:

- buffer *LostBuffer* holding a triple *(source, pattern, sequence number)* for each lost event
- buffer *Cache* holding a copy of the last events received

Invoked periodically, e.g., after timeout expiration.

Triggers the start of a new gossip round for a pattern in the subscription table.

startGossipRound ()

if $LostBuffer \neq \varnothing$ **then**
  choose a pattern p from the subscription table (considering only those coming from clients)
  create $digest = \varnothing$
  for all $(s, p, c) \in LostBuffer$ **do**
    insert (s, c) in $digest$
  end for
  create $gossipMsg = (self, p, digest)$
  send $gossipMsg$ towards one or more subscribers for p
end if

Invoked on a dispatcher upon receipt of a gossip message.

handleGossipMsg ($gossipMsg$)

for all $(s, c) \in gossipMsg.digest$ **do**
  if $\exists e \in Cache \mid e$ satisfies the triple $(s, gossipMsg.pattern, c)$ **then**
    send e to $gossipMsg.initiator$
  end if
end for
if *self* is not subscribed to $gossipMsg.pattern$ **then**
  send $gossipMsg$ to one or more subscribers for $gossipMsg.pattern$
else
  with probability $P_{forward}$ send $gossipMsg$ to one or more subscribers for $gossipMsg.pattern$
end if

Figure 5.3: Pull algorithm.

system dispatchers are not supposed to receive all events, rather only those matching the subscribed patterns. Hence, holes in sequence numbers are not enough to determine whether a given event message has been lost or it was

instead simply not relevant to the dispatcher's subscriptions.

To solve this issue, we tag events with identifiers that, besides containing the information about the event source, contain the patterns[2] matched by the event, each associated with a sequence number incremented at the source each time an event is published for that pattern. For instance, let us consider an event source named JOHN, which already published four events matching a hypotethical and simplified pattern GREEN, and other three matching RED. When this event source publishes a new event that matches both patterns, the identifier associated to the event message is JOHN:GREEN-5:RED-4. Patterns are associated to an event at its source: this is made possible by the fact that a subscription forwarding strategy is chosen, and hence subscriptions are known to all dispatchers.

This scheme enables the design of a reactive pull approach, based on the sequence numbers stored in the event identifier. Whenever a dispatcher receives an event matching a pattern p, but for which the sequence number associated to p in the event identifier is greater than the one expected for that pattern and source, it can detect the loss of an event and trigger the appropriate actions.

In the solution shown in Figure 5.3, as soon as a lost event is detected it is immediately inserted in the buffer *LostBuffer*, by an action that is not shown explicitly in the figure, in order to keep the algorithm description concise. The elements of *LostBuffer* are the triples identifying an event in our encoding, i.e., source, pattern, and sequence number associated to the pattern.

[2] A hash signature of the pattern is actually sufficient.

The action startGossipRound, that is invoked at regular intervals like in the push solution, first checks whether there are lost events. If so, a gossip round is effectively triggered[3]. In this case, the choice of the pattern p characterizing this gossip round is limited to the subset of pattern in the subscription table that have been sent by clients attached to the dispatcher. Note how, unlike push, we are not using the whole subscription table, since this time the focus is on retrieving only events that are relevant to the gossiper, and not on disseminating events to as many dispatchers as possible. The pattern p is used to select the corresponding lost events from *LostBuffer* and insert them in the digest attached to the gossip message *gossipMsg*, which is then routed in a way analogous to the push solution.

When a dispatcher receives a *gossipMsg*, it checks its event cache to see whether it holds some of the events requested by the gossiper. It does not matter whether that dispatcher is a subscriber for the pattern p requested by the gossiper. For instance, following up on our earlier example, let us suppose that the gossiper is missing the event JOHN:GREEN-5, and that this information is included in a *gossipMsg*. Of course, there is no way for the gossiper to know that this event has been delivered also to dispatchers subscribed to RED. Instead, a dispatcher that is subscribed to RED events and has cached the event can easily determine that JOHN:RED-4 and JOHN:GREEN-5 are indeed the same event by looking at the event identifier (JOHN:GREEN-5:RED-4).

[3]In principle, a gossip round could be triggered immediately upon detection of a lost event. Nevertheless, in scenarios characterized by frequent losses it is convenient to delay the triggering to the next gossip round, so that multiple lost events can potentially be retrieved during a single round.

Hence, the dispatcher can retransmit the missed event to the gossiper.

Although events can be retransmitted by any dispatcher that has "seen" the event, it is very important for a gossip message to be steered towards a subscriber for the same pattern of the lost event (GREEN in our case). In fact, subscribers act as "points of accumulation" for events, in that not only they might have the requested event in the cache, but they also actively try to recover lost events through gossip. Hence, when a *gossipMsg* reaches a subscriber, rather than a middleman dispatcher, the likelihood of recovering the lost events is much higher.

Clearly, the effectiveness of the pull solution depends on the number of subscribers for each pattern. The more the subscribers, the more efficient the recovery process is, since there are more subscribers to exploit, and more alternative routes to follow during the propagation of *gossipMsg* thus increasing the probability to recover events from dispatcher not belonging to the "group" defined by the pattern specified by the gossiper. As a consequence of this observation, the performance of the pull algorithm is likely to be subject to a drastic reduction in scenarios where instead common interests are very rare and alternative routes scarce. This motivates our third approach, illustrated in the next section.

5.3 Source

The last solution we present in this paper is illustrated in Figure 5.4, and employs a source-routing scheme that tries to recover lost events by

Per dispatcher information:

- buffer *LostBuffer* holding a tuple *(source, pattern, sequence number)* for each lost event
- buffer *Cache* holding a copy of the last events received
- buffer *Routes* holding a pair *(source, route)* for each event source

Invoked periodically, e.g., after timeout expiration.

Triggers the start of a new gossip round for a source.

startGossipRound ()

if $LostBuffer \neq \varnothing$ **then**
  choose a publisher s
  create $digest = \varnothing$
  for all $(s, p, c) \in LostBuffer$ **do**
    insert (p, c) in $digest$
  end for
  create $gossipMsg = (self, s, digest, r)$ with $r|(s, r) \in Routes$
  send $gossipMsg$ to the first node in $gossipMsg.route$
end if

Invoked on a dispatcher upon receipt of a gossip message.

handleGossipMsg ($gossipMsg$)

for all $(p, c) \in gossipMsg.digest$ **do**
  if $\exists e \in Cache \mid e$ satisfies the triple $(gossipMsg.source, p, c)$ **then**
    send e to $gossipMsg.initiator$
  end if
end for
if $self = gossipMsg.source$ **then**
  drop $gossipMsg$
else
  send $gossipMsg$ to the next node in $gossipMsg.route$
end if

Figure 5.4: Source-based algorithm.

walking backwards towards the event source. This solution can be regarded as dual and complementary to the pull solution we just described, in that

the gossiping focuses on sources instead of patterns.

We present the source algorithm separately for clarity's sake but it has been developed as a fallback for the pull solution. It allows to overcome the aforementioned drawback of the pull solution ensuring good performance even in scenarios where the number of subscribers for each pattern is very low.

At simulation stage usually source and pull solution are coupled into a single algorithm (where the choice between a pull or a source gossip round is made in probability) but occasionally they are evaluated separately to remark each contribution.

In this solution, we assume that every published event is cached at the source, and possibly at dispatchers located on routes towards the subscribers for that event. Moreover, the address of each dispatcher encountered by the event during its travel towards a subscribers is recorded in the event message, thus constructing a route from the source to the subscriber. Lost events are handled in the same way as in the pull solution and inserted in *LostBuffer*, based on the same identifier scheme. In addition, a new buffer *Routes* is necessary to store the route towards a given source, e.g., based on the route information stored in the most recent event received from that source.

When a new gossip round is triggered, an event source is chosen among those known. The actions startGossipRound and handleGossipMsg essentially behave like their counterparts in the pull algorithm, except for the fact that the information distinctive of the gossip message is now the event source rather than pattern, and that *gossipMsg* is now augmented with the route

information necessary to be routed back to the source. It is interesting to note that there is no guarantee that the route stored in *Routes* is the same originally followed by the missing event. On the other hand, it is likely that the two share at least the first portion or, in the worst case, the source.

One reasonable question to ask is whether this solution suffers from the well-known "acknowledgment implosion" problem, that affects several reliable group communication schemes, and occurs when several nodes missing a message request retransmission simultaneously to the same node. Our push and pull solutions, thanks to their distributed nature, are essentially free from these risk. For the source-based solution, the probability of such a phenomenon is rather low. In fact, it is unlikely that two subscribers holding different subscriptions (e.g., our usual GREEN and RED) realize at the same time that they have missed an event. Following our example, this would happen only if the next event published by JOHN matches both patterns as well. Finally, differently from traditional NACK-based reliable multicast schemes, retransmission requests are handled by the first dispatcher holding the desired event found along the path, thus avoiding to overload the source.

Chapter 6

Simulation Results

In this chapter we first show the behavior of our algorithms in a number of scenarios and then we test them against main parameter values in order to explore their changes in performance.

In the last section we introduce some graphs pertaining to overhead introduced by our algorithms and analyze its impact with respect to performance.

6.1 Simulation Setting

In the absence of reference scenarios for comparing content-based systems, we defined our own, based on what we believe are reasonable assumptions covering a wide spectrum of applications.

In our simulations we considered a default scenario and explored the behavior of our algorithms as one parameter varies. In the following we present these parameters and indicate their default value. These values are summarized in table 6.1

Events, subscriptions, and matching. Events are represented as randomly-generated sequence of numbers, where each number represents a pattern of the system. We settle for a uniform distribution in this sequence. Subscriptions are represented as a single number. An event matches a subscription if it contains the number specified by the subscription. In each simulation the number of different subscriptions available in the system is set to a constant **numPatterns** (default value = 70). Each dispatcher can subscribe to a number of subscriptions (drawn randomly from the available ones) such that for each pattern the number of subscribers is equal to **numSubscribersPerPattern** (default value = 2.8).

Publish rate. The behavior of each dispatcher in terms of publish and (un)subscriptions is governed by a triple of parameters, f_{pub}, f_{sub}, and f_{unsub}, respectively governing the frequency at which publish, subscribe, and unsubscribe operations are invoked by each dispatcher. The most relevant is f_{pub}, which essentially determines the load in the system in terms of event messages that need to be routed. Based on this parameter, we choose a load scenario with a **publishRate** of about 50 publish/s per dispatcher ($f_{pub} = 0.05$).

Tree topology. The results we present here are all obtained with tree configurations of **numEDs** event dispatchers (default value = 100), where each dispatcher is connected with at most four other dispatchers (one parent and three children).

Clients are not modeled explicitly, as their activity ultimately affects only the dispatcher they are attached to. Moreover, in the scenarios we target

(e.g., MANET and peer-to-peer networks) the architecture of the publish-subscribe system is likely to have clients and dispatchers coincide.

Channel Reliability. We assume that each link in the tree connecting two dispatchers behaves as a 10 Mbit/s Ethernet link with a fixed **error-Rate** (default value = 0.1) which, of course, affects both events and gossip messages.

Tree reconfiguration. The selection of the links breaking or appearing is done randomly. Reconfigurations are triggered with a frequency determined by the duration of the interval between two reconfigurations, **recInterval**.

Buffer size. Every dispatcher is equipped with a buffer storing events to satisfy retransmission requests. Its size varies according to the parameter **bufferSize** (default value = 1500). In our simulations we adopt a simple FIFO buffering strategy where each dispatcher caches only events for which it is either the publisher or a subscriber. Other simple strategies, such as buffering also the events simply routed through the dispatcher, have not been reported here as their performance did not improve on selected strategy.

Gossip interval. The frequency of gossiping is controlled by a parameter. We set it to a fixed value of 0.03 seconds.

Source probability When a combination of pull and source algorithms is used, a parameter called **sourceP** sets the probability of using the source strategy against the pull one. In our simulation we chose a value of 0.3.

Simulation tool. In our simulations, we are not concerned with modeling the behavior of the underlying networking stack. Instead, we are essentially comparing the algorithms only at the application level. For this reason, we decided to develop our simulations using OMNET++ [43], a free, open source discrete event simulation tool.

Parameter	**Default Value**
numPatterns	70
numSubscribersPerPattern	2.8
publishRate	50
numEDs	100
errorRate	0.1
bufferSize	1500
sourceP	0.3
gossipInterval	0.03

Table 6.1: Default values used in simulations

6.2 Event Delivery

As mentioned in Chapter 1, all the algorithms introduced so far have been designed for and find their natural application in situations where events get lost because of the on-going reconfiguration of the event dispatching infrastructure, due to mobility or other causes. To take into account such situations we did not make any hypothesis about the stability of network

routes among dispatchers. At the same time, it is important to note that *we did not make any hypothesis about the cause of event loss*. Hence, our algorithms enjoy general applicability, and can provide improved reliability in any situation determining event loss, e.g., in presence of a stable dispatcher graph with lossy channels.

In order to show that we tested our algorithms in two scenarios which experience different causes of event loss.

6.2.1 Error Rate

We defined two scenarios with different error rates. The first, very critical, exhibits an average delivery equal to 50% obtained by setting **errorRate** to 0.1. The second shows an average delivery of 70% (**errorRate** = 0.05).

In figures 6.1 and 6.2 is shown that pull-source and push algorithms exhibit similar performance, recovering up to 90% in the critical scenario and up to 98% in the other. In the former one the recovery phase is responsible for the delivery of almost half of the total message delivered.

6.2.2 Tree Reconfiguration

To retain some degree of control over when a reconfiguration occurs, we assume that each broken link is replaced by a new link in 0.1 s.

Then we considered two reconfiguration scenarios: **recInterval** = $0.2s$ which yields non-overlapping reconfigurations, and **recInterval** = $0.03s$, which defines a situation where several reconfigurations overlap. The latter can be regarded as an approximation of the case in which a non-leaf dis-

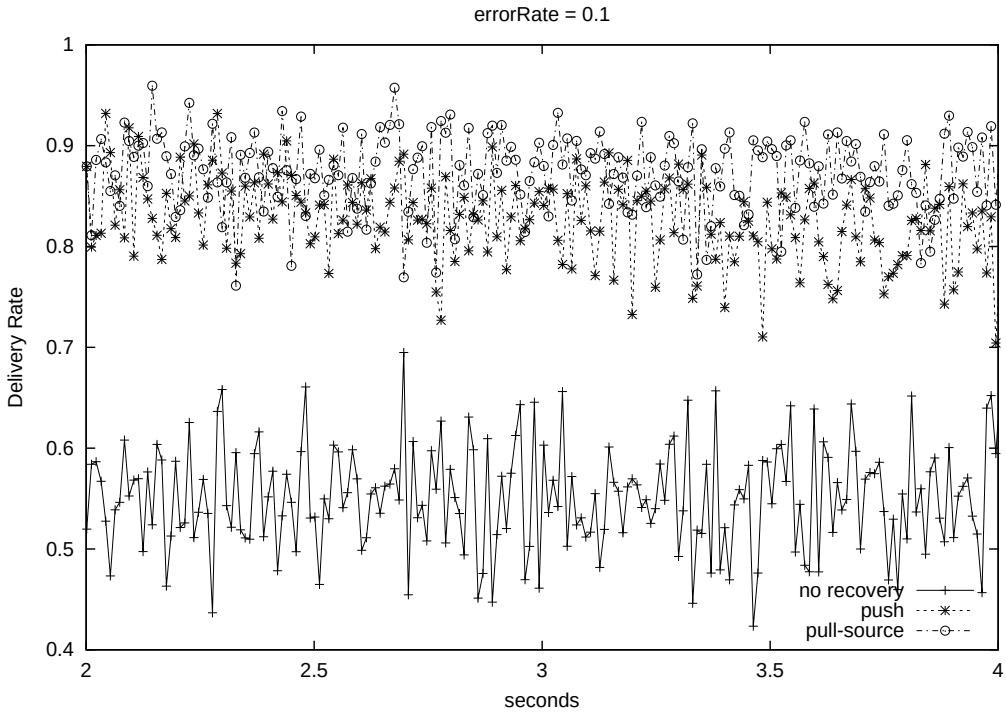

Figure 6.1: Example of delivery in a very critical scenario

patcher is detached from the tree, and hence multiple links are broken at once. In any case, it defines a particularly difficult reconfiguration scenario, and provides a good, extreme test case for our analysis.

Reconfigurations are allowed only in the interval between 4 and 6 seconds, and results are charted in figures 6.3 and 6.4.

We can see that even in situation where routes are under heavy repair where delivery drops as low as 50% our recovery algorithms do not drop below 90% indicating a good robustness to variation in delivery of the underlying system.

In the rest of the chapter, we focus on scenarios characterized by lossy channels since these represent the most general case. In particular, we set

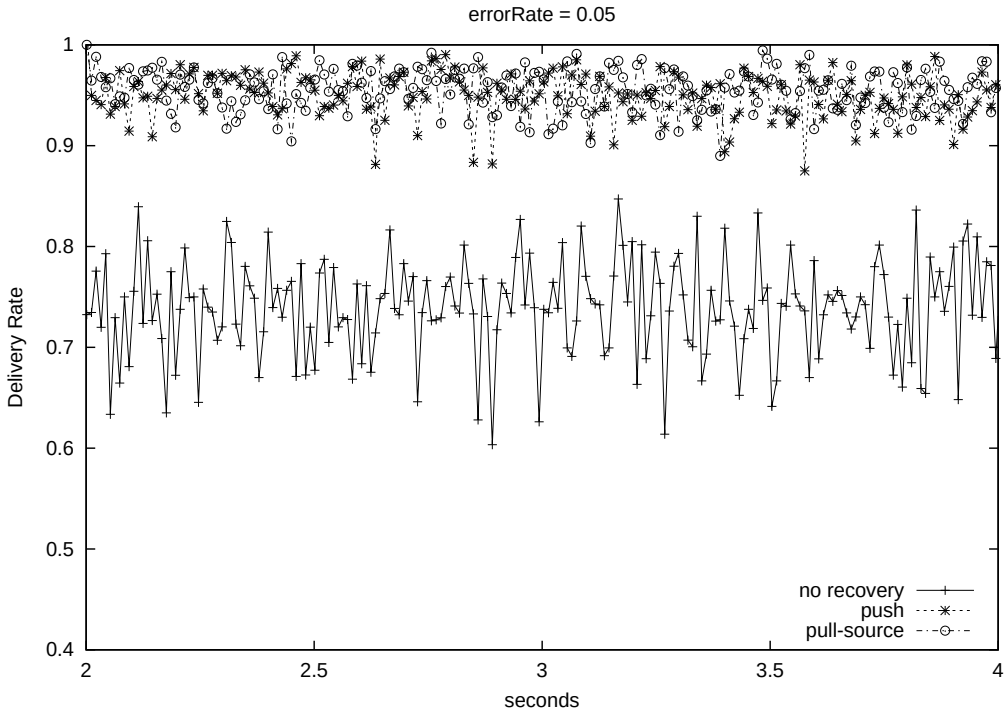

Figure 6.2: Example of delivery in a lossy scenario

errorRate to the critical value of 0.1 to better appreciate the variations in performance with respect to the parameter we vary.

6.2.3 Buffer size

One of the most critical parameters to set in a recovery algorithm is the length of buffers. In fact as remarked by figure 6.5 with the growth of **bufferSize** algorithm delivery rates converge to 1, apart from pull for the reason set forth in 5.2. On the other hand with very small buffers pull performance is better than in the case of other algorithms.

We consider a range of values from 500 to 3500 buffered events, that in our scenario correspond to a time length varying approximately from 1.3

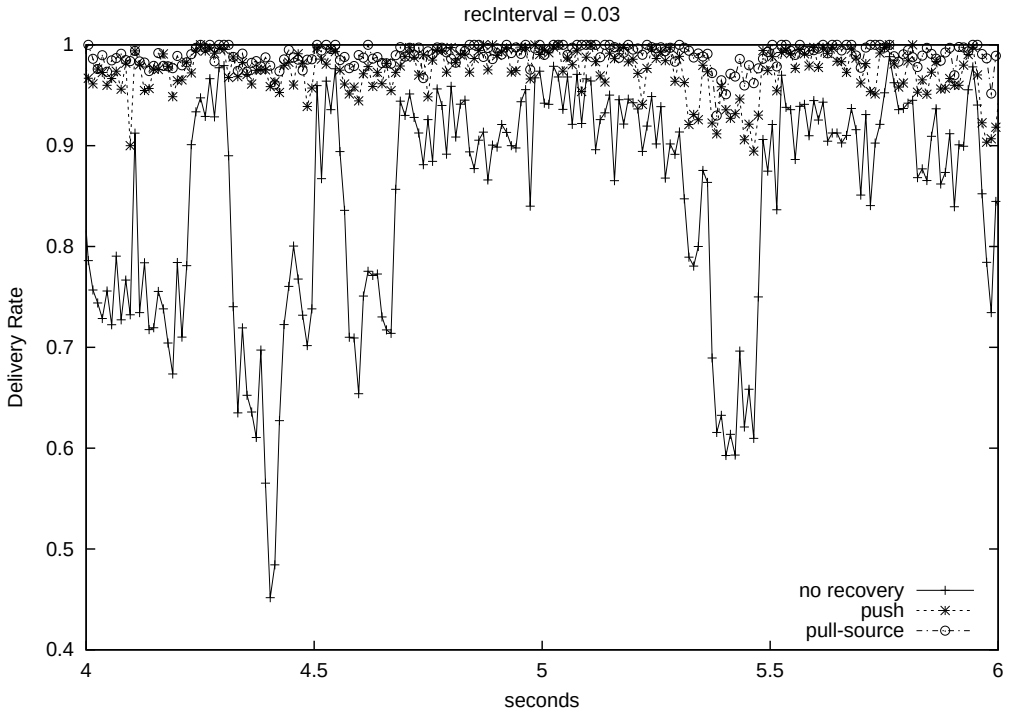

Figure 6.3: Example of delivery in a scenario with overlapping reconfigurations

seconds to 9.2 seconds.

Since buffer length is shown to be responsible for a high percentage of algorithms performance, we were especially careful in setting its value so as to avoid impacting the results of the following simulations.

Reasoning on how to set buffer length to allow algorithms to operate correctly an assumption that seems adequate is to keep messages for a fixed amount of time before discarding them. This of course would result in a constant length as measured in time, and an absolute size that grows linearly with traffic rate.

Consequently, when a modification in a parameter value leads to an in-

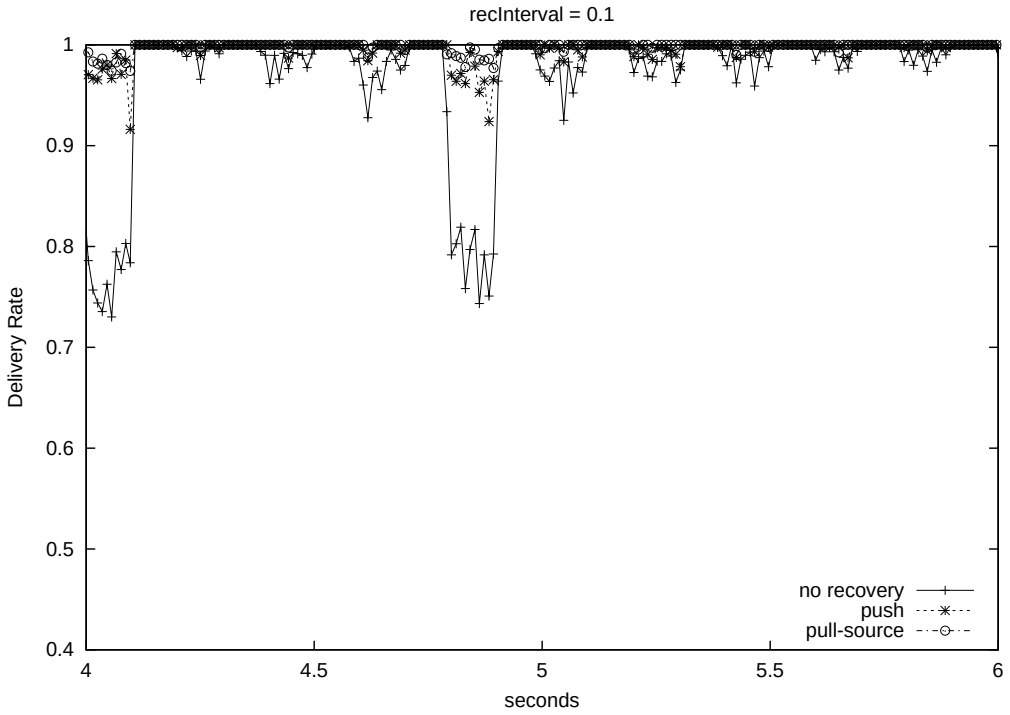

Figure 6.4: Example of delivery in a scenario with non-overlapping reconfigurations

creased traffic on dispatchers, we adjusted **bufferSize** according to the above rule.

This choice is in fact rather conservative considering that a known result from epidemic literature is that the buffering requirement on each dispatcher grows $O(\rho \log n)$[1], where n is equal the system size and ρ is the total message rate (in our case $\rho = n f_{pub}$).

[1]In [28] some improvements have been proposed and their applicability to content-based system will be a subject for future works

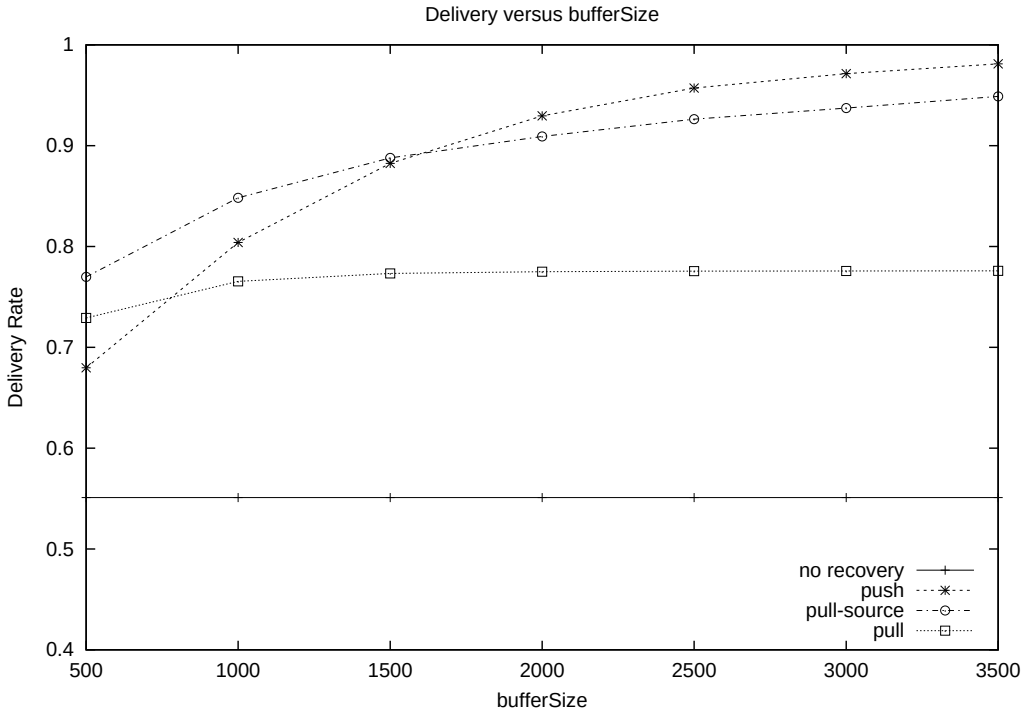

Figure 6.5: Effect of bufferSize on delivery rate

6.2.4 Pattern

Since pattern distribution is a key factor in content-based systems, we explored several situations to evaluate how performance changes. The main parameters we used to characterize the content-based scenario are the total number of patterns present in the system (**numPatterns**) and the number of dispatcher subscribed to each pattern (**numSubscribersPerPattern**).

In the first graph (shown in figure 6.6) we test our algorithms with respect to the former parameter, keeping the latter constant.

In the second (figure 6.7) we analyze the dual case. In our simulation we increase **numSubscribersPerPattern** by varying the number of subscrip-

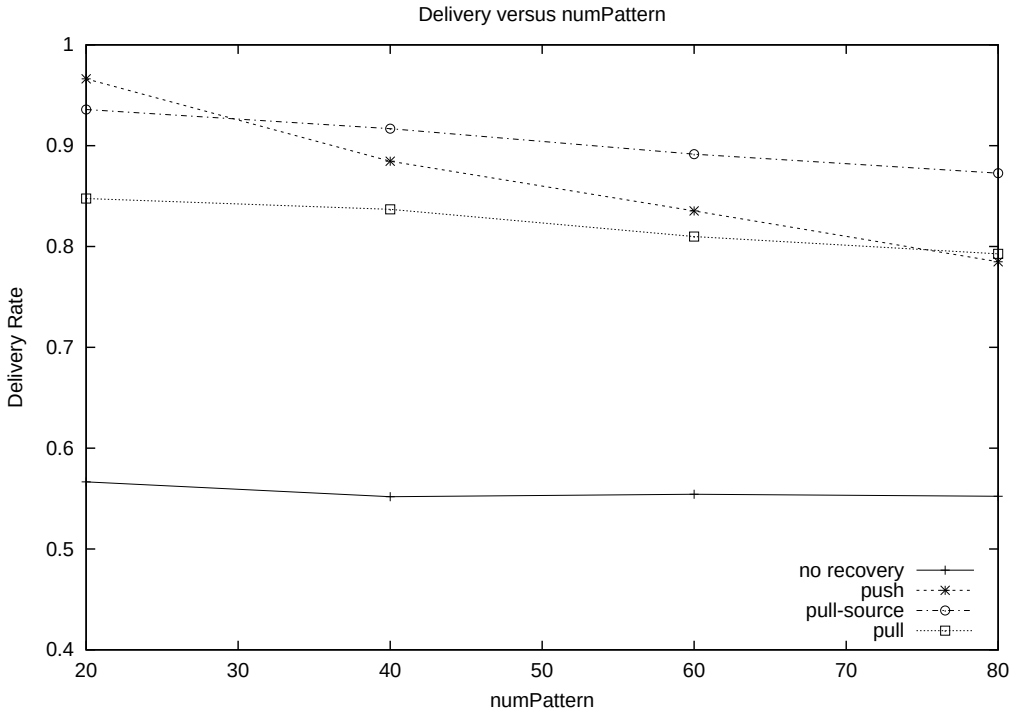

Figure 6.6: Performance in different content-based scenarios (I)

tions held by each dispatcher[2].

Figure 6.6 confirms that pull-based algorithms are not influenced by **numPatterns**. On the other hand push approach performance decreases as patterns grow. In chapter 7 we elaborate on this showing how this effect is lessened in real scenarios.

On the other side pull-based algorithms improve significantly their performance as **numSubscribersPerPattern** increases. Push is somehow independent from changes of this value.

The aforementioned results confirm our previous expectations since pull

[2]As this, of course generate more traffic for each dispatcher, to avoid correlation with the traffic graphs we compensate by slightly rising **bufferSize**

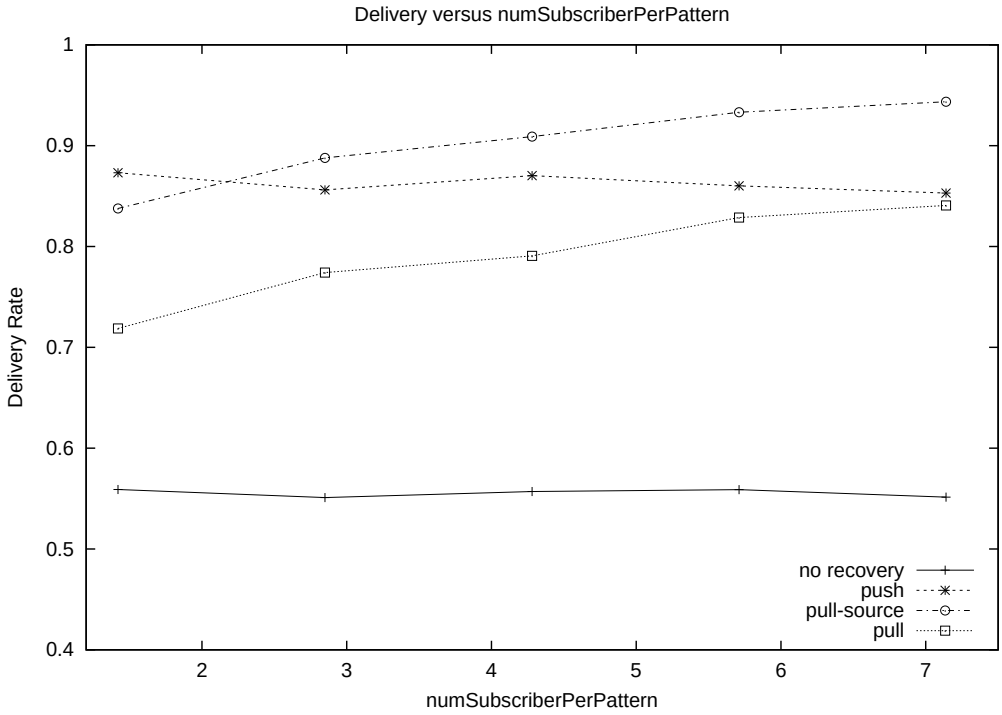

Figure 6.7: Performance in different content-based scenarios (II)

recovery mechanism exploits the presence of subscribers acting as "point of accumulation" for events, whereas push depends on how many different patterns a dispatcher has to pick for gossip messages.

6.2.5 System size

As a conclusion we present a graph relating to the size of the system under simulation. In each run we added a number of dispatchers to the system thus increasing the event load on every dispatcher, to maintain equality among runs we also raised **bufferSize** accordingly to keep buffer time length constant (in this case approximately equal to 4s).

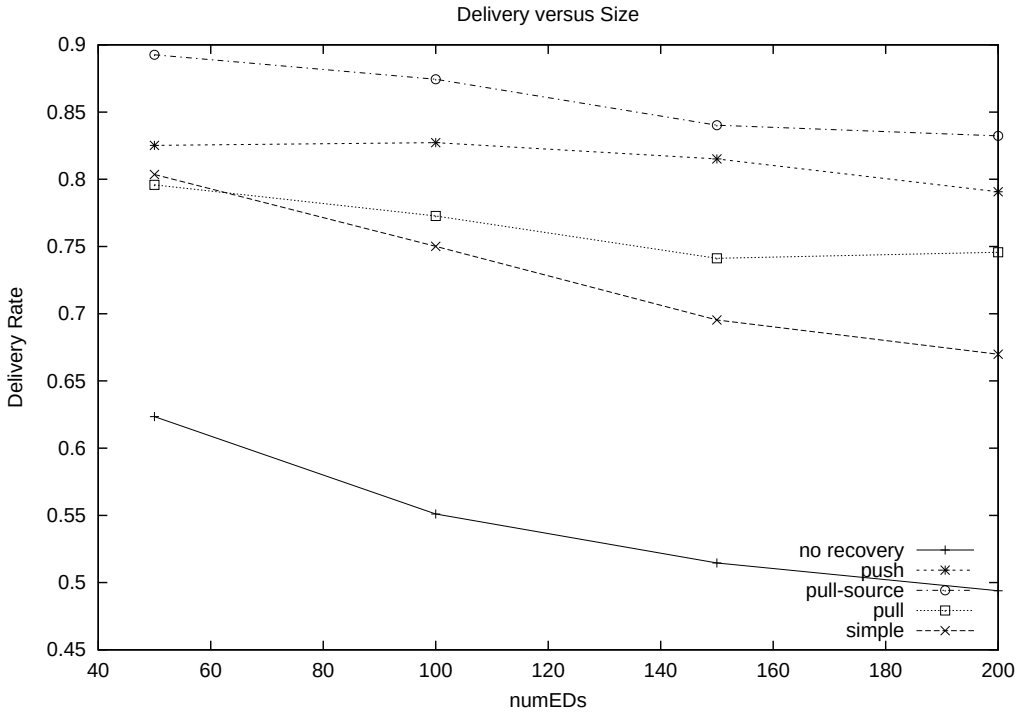

Figure 6.8: Effect of size on delivery rate

The results underline a good scaling behavior relative to the system size for both push and pull variant, as per epidemic algorithms good scaling properties. The "simple solution" series verifies the analysis we presented in 4.2: as the network size grows its efficiency degrades dramatically because the probability of contacting "removed" nodes becomes higher.

6.3 Overhead

In this section we analyze the behavior of our algorithms with respect to system size, focusing on overhead expense rather than delivery rate as we did in figure 6.8 obtaining a constant delivery.

The first measure of overhead we introduce is the amount of gossip messages seen by every dispatcher. As graph 6.9 indicates, when the system scales up, the trend of traffic due to gossip messages is far under linear.

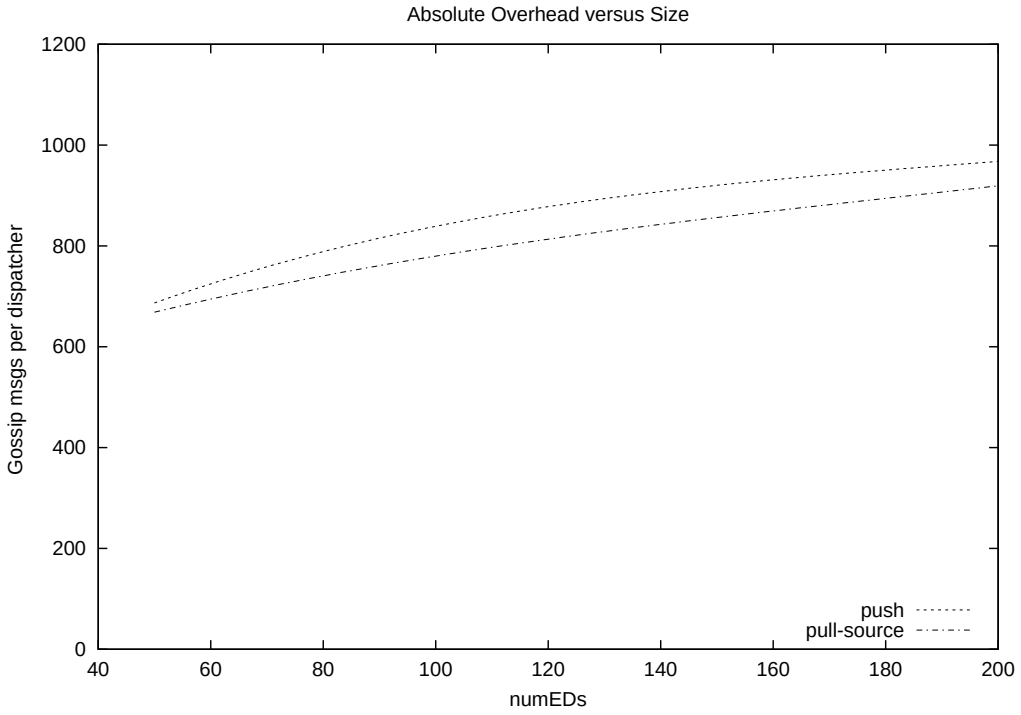

Figure 6.9: Measurement of gossip traffic per dispatcher as system size grows

This result is a direct consequence of the decentralized nature of gossip algorithms: the effort in term of gossip messages published by each dispatcher is independent of system size. Hence, the gossip traffic growth is proportional to the hops traveled by each gossip message which in our case increase logarithmically.

To give an idea of the global traffic rate due to rough event dissemination versus gossip messages we plot the ratio between them.

The real bandwidth ratio should account for the relative size of the two.

We act conservatively by assuming an equal weight to both of them, though it is likely that events are much bigger.

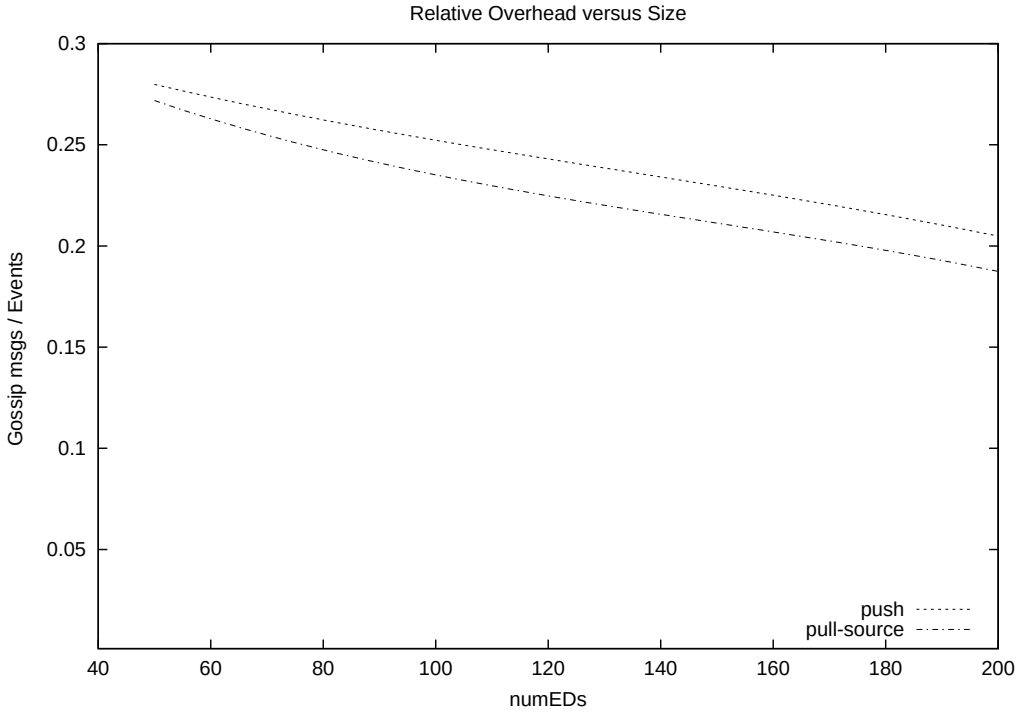

Figure 6.10: Percentage of traffic due to the gossip messages as the system size grows

Graph 6.10 evidences that as network size grows, event traffic rises faster than gossip one. This could be explained by noting that gossip mechanism does not involve any multicast primitive.

Chapter 7

Discussion

The solutions we proposed exhibit very different characteristics, and hence perform best in different scenarios. We analyzed their tradeoffs and applicability through simulation, and results demonstrate indeed that a significant improvement in the event delivery rate is achieved. In the following we provide some additional remarks about the working of our solutions and their potential exploitation.

7.1 Performance Considerations

We can observe that buffer size plays a key role in determining the performance of our algorithms. Likewise the other parameter responsible of the behavior of our solutions is the interval between gossip rounds. Small values of **gossipInterval** improve the performance but increase the overhead as well, while large values may increase the time needed to retrieve a missed event or even fail to recover it. Of course the two values of **bufferSize** and

gossipInterval are intimately interconnected: they together set the tradeoff between performance and overhead respectively in terms of memory requirements and network traffic.

Relatively to parameters defining the content based scenario, we can observe that the effectiveness of the pull solution strongly depends on the availability of a reasonable number of subscribers for the same pattern used to "pull" messages. Hence, scenarios where the set of subscriptions of each dispatcher are largely disjoint are very critical for pull.

One possibility for overcoming the limitations of a given pull algorithm is to couple it with other algorithms. The most natural solution is actually to use at the same time the pull solution with the source-based one, in that it essentially exploits the dual approach of gossiping towards the event sources, rather than subscribers. The idea is that the combination of pull and source-based improves the chances of recovering an event, since it is going to look not only through the route towards who received the event, but also towards who sent it.

On the contrary the number of subscribers should not affect significantly the performance of push. A challenging scenario for push is represented instead by a high number of patterns. This is alleviated by the fact that what really hampers push dissemination is number of patterns matched by buffered messages; which, given the reasonable assumption of "locality" between patterns, are much fewer than the total number of patterns available globally in the system.

7.2 Overhead Considerations

One great advantage of the pull-based solutions is their reactivity. Reactive approaches perform better, in terms of generated traffic overhead, when there is a large variability in the frequency of event losses, e.g., in situations where at first the system is in a quiescent state with the system losing very few events followed by periods in which the system experiences bursts of errors. This is actually what happens when the topology of the dispatching infrastructure is changed, e.g., because of mobility.

As for push, we may observe that when the system is stable a proactive push approach is likely to result in wasted bandwidth. To remove this potential source of inefficiency, an adaptive approach can be exploited where the gossip interval T is changed dynamically according to the current state of the system. A simple algorithm can be derived along the lines of [9], where a dispatcher d maintains a count of the gossip messages which did not generate an event request *reqMsg*. When this count reaches a given threshold, d increases the value of T by a given amount. The original value of T is reset when d detects that events are lost in the system, i.e., when it receives either a *reqMsg* or a *gossipMsg* concerning events it has missed. This simple scheme allows to consume only negligible bandwidth shortly after an approximation of global consistency is detected.

7.3 Enhancements and Open Issues

Moreover, another issue is that of computational overhead. In this respect, the pull-based solutions require that, when an event e is published by a dispatcher, the latter performs a match of e against *all* the patterns in its subscription table. This is more than normally required, since usually the match processing needed to route a message towards a neighbor stops as soon as the first matching pattern is found. While we are currently investigating optimizations to limit this overhead, we also observe that only the source experiences it: the event routing performed by the other dispatchers in the system follows the normal processing. Clearly, there is also a traffic overhead being paid because the event message is inflated by the subscriptions and sequence numbers attached to it. Nevertheless, since hash values and integers are being carried, we estimate that this overhead should not impact significantly the overall performance of the system.

In large networks with a high volume of traffic, cache management is another critical issue. In particular, the size of the cache and the policy used to discard obsolete elements are key. As for the second aspect, criteria should privilege the persistence of events for which the dispatcher is either a subscriber or a source, with respect to events the dispatcher has received only for routing purposes. Moreover, event persistence should be tuned (e.g., with probability parameters) so that the caches belonging to different dispatchers are somehow differentiated. This would allow to improve the likelihood of retrieving the lost event, and at the same time allow the individual caches to be smaller, leveraging off of the scale factor.

As a final remark, the solutions we presented assume a subscription forwarding scheme. While this scheme is the most common, an open question is to what extent it is applicable to other routing schemes. Interestingly, the push solution is independent of the routing scheme adopted since it is based on the idea of routing the gossip message based on the pattern attached to it, as if it were an event. Conversely, the event identification scheme adopted by the pull and source-based algorithms is based on the global knowledge of subscriptions, which is not necessarily provided by other schemes.

Chapter 8

Related Work

Several publish-subscribe systems offer a reliable service (e.g., all the JMS [38] compliant systems) by adopting a centralized event dispatcher and reliable channels between the dispatcher and its clients. Similarly, some of the existing publish-subscribe systems which adopt a distributed dispatcher provide a reliable service [40, 32, 4, 31, 44, 6], but none of them use a content-based routing scheme.

Researchers working on reliable multicast [34, 26, 20] and group communication [14, 8] proposed several protocols for reliable multicast where routing is group or subject-based.

Traditional implementations of such applications work well in small-scale settings, but show drastic reduction in performance as system size increases. In fact, the ack/nack mechanism employed by such protocols to improve reliability unfortunately tends to compromise their scalability by overflowing the sender's buffer and congesting the nearby network.

More recent protocols (e.g. RMTP[22]) propose a mechanism of *local recovery* to limit the load of the network, but they depend on the stability of the multicast routing tree. In very dynamic environment, such as MANETs or peer-to-peer networks, routes change very rapidly and the methods used are consequently not available to us.

Indeed, other distributed repair based protocols, such as SRM[17], do not require a permanent multicast tree, but have a solicitation and retransmission mechanism that involves multicasts; when a duplicate retransmission occurs, all participants process and transmit extra messages.

FEC-based reliability mechanisms, instead, try to reduce retransmission requests to the sender by encoding redundancy in the data stream. In systems adopting a *content-based* paradigm, even under error-free dissemination, messages sent by a source make up multiple streams that are difficult to identify (see 5.2) and since overlapped are also difficult to add redundancy to. FEC in multicast systems is shown to be very appealing when the group of receivers is large because different nodes could exploit the same overhead to repair different losses or when coupled with ARQ(as in [25]) a single retransmission could repair different losses. Unfortunately large groups are much rarer in content based system.

In recent years, gossip techniques have been employed successfully to deal with scalable reliable multicast in peer-to-peer networks [15] and in highly dynamic environments such as mobile ad hoc networking [7, 24]. Results have been encouraging but, again, their techniques are not directly applicable to our problem since they consider broadcasting information to all participants

in a group.

To the best of our knowledge, the only system that provides reliable content-based routing is *hpcast* [16]. In hpcast nodes are organized in a hierarchy where the leaves represent event subscribers and publishers, and intermediate nodes represent *delegates*, i.e., special nodes which are chosen to represent aggregate interests of their sons. Gossip-push is used to distribute events starting from the root of the hierarchy and moving down each time a delegate retrieves an event that could interest its sons. This idea of using a gossip algorithm not only to reduce the number of lost events but as the only routing mechanism is simple and elegant, but results in several drawbacks. First, in absence of faults it increases the overhead since events are not routed only to interested nodes, but they can reach also non-interested nodes or even be sent more than once to the same node. Second, even in absence of faults it does not guarantee that events are delivered correctly, but provides only a bimodal behavior. Third, it forces the adoption of a gossip-push approach in which gossip messages include the entire event content instead of a simple digest, thus further increasing the network traffic. Finally, the nodes near to the root of the hierarchy are subject to a high traffic, and hence must keep their event caches very large to increase the probability of correctly delivering events.

Chapter 9

Conclusions and Future Work

Mobile computing defines a scenario that is extremely dynamic and fluid. In this scenario, distributed content-based publish-subscribe appears to provide the necessary component decoupling, flexibility, expressiveness, and scalability. On the other hand, in mobile computing communication links are often unreliable, and reconfiguration often brings in another source of event loss. The problem of reliable event delivery has not yet been tackled by researchers, thus hampering the development of middleware for real-world applications.

In this thesis, we presented solutions that exploit epidemics algorithms for improving event delivery in this kind of publish-subscribe systems. Epidemics algorithms [1, 23, 15] provides a lightweight, scalable, and robust means of reliably disseminating information to a group of recipients, and hence are amenable to being used in such highly dynamic and unreliable scenarios.

The presented algorithm were also tested intensively in simulated environments, where performance and overhead are assessed. Future work will

include real implementation in a new generation middleware and further validation through data gathering on the field, together with more exhaustive exploration of distributed buffering techniques.

Bibliography

[1] K. Birman et al. Bimodal multicast. *ACM Trans. on Computer Systems*, 17(2):41–88, 1999.

[2] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proc. of the 8^{th} Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[5] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, 2001.

[6] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure.

IEEE Journal on Selected Areas in communications (J-SAC), 20(8), 2002.

[7] R. Chandra, V. Ramasubramanian, and K. Birman. Anonymous gossip: Improving multicast reliability in mobile ad-hoc networks. In *Proc. 21st Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 275–283, 2001.

[8] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[9] F. Cuenca-Acuna et al. PlanetP: Infrastructure support for P2P information sharing. Technical Report DCS-TR-465, Dept. of Computer Science, Rutgers University, November 2001.

[10] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, September 2001.

[11] G. Cugola, D. Frey, A.L. Murphy, and G.P. Picco. An algorithm for dynamic reconfiguration of publish-subscribe systems. Technical report, Politecnico di Milano, February 2002. Submitted for publication. Available at `www.elet.polimi.it/~picco`.

[12] G. Cugola, G.P. Picco, and A.L. Murphy. Towards distributed publish-subscribe middleware for mobile systems. In *Proc. of the 3rd Int. Workshop on Software Engineering and Middleware (SEM02), co-located with*

the 24th Int. Conf. on Software Engineering (ICSE03), Orlando (FL), USA, May 2002.

[13] A. Demers et al. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, 22(1):8–32, 1988.

[14] C. Diot, W. Dabbous, and J. Crowcroft. Group communication. *IEEE Journal on Selected Areas in Communication. Special Issue on Group Communication*, May 1997.

[15] P. Eugster et al. Lightweight probabilistic broadcast. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 2001)*, pages 443–452, 2001.

[16] P. Eugster and R. Guerraoui. Hierarchical probabilistic multicast. Lausanne, 2001.

[17] S. Floyd et al. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. on Networking*, 5(6):784–803, 1997.

[18] A. J. Ganesh, A.-M Kermarrec, and L. Massouli. Peer-to-peer membership management for gossip based protocols. *IEEE Transactions on Computer*, February 2003.

[19] Zygmunt Haas, Jospeh Y. Halpern, and Li Li. Gossip-based ad hoc routing. In *Proceedings of INFOCOM*, 2002.

[20] B. Levine and J. Garcia-Luna-Aceves. A comparison of known classes of reliable multicast protocols. In *Proc. of the IEEE International Conference on Network Protocols*, October 1996.

[21] L. Li, J. Halpern, and Z. Haas. Gossip-based ad hoc routing.

[22] J.C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, March 1996.

[23] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference*, pages 364–379, 1999.

[24] J. Luo, P. Eugster, and J.-P. Hubaux. Route driven gossip: Probabilistic reliable multicast in ad hoc networking.

[25] Jörg Nonnenmacher, Ernst W. Biersack, and Don Towsley. Parity-based loss recovery for reliable multicast transmission. *IEEE/ACM Transactions on Networking*, 6(4):349–361, 1998.

[26] K. Obraczka. Multicast transport protocols: a survey and taxonomy. *IEEE Communications Magazine*, 36(1):94–102, January 1998.

[27] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In J. Sventek and G. Coulson, editors, *Middleware 2000*, volume 1795 of *LNCS*, pages 185–207. Springer-Verlag, 2000.

[28] Oznur Ozkasap, Robert van Renesse, Kenneth Birman, and Zhen Xiao. Efficient buffering in reliable multicast protocols. In *Proceedings of NGC99*, Pisa, Italy, November 1999.

[29] C. Perkins. Ad hoc on demand distance vector (aodv) routing, 1997. C. Perkins. Ad Hoc On Demand Distance Vector (AODV) Routing IETF, Internet Draft, draft-ietf-manet-aodv-00.txt, November 1997.

[30] G.P. Picco, G. Cugola, and A.L. Murphy. "Efficient Content-Based Event Dispatching in Presence of Topological Reconfigurations". Technical report, Politecnico di Milano, 2002. Submitted for publication. Available at `www.elet.polimi.it/~picco`.

[31] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. of the Workshop on Distributed Event-Based Systems (DEBS), 2002.*, Vienna, Austria, July 2002. IEEE Computer Society.

[32] Real-Time Innovations, Inc. *NDDS: Network Middleware for Distributed Real Time Applications.* `http://www.rti.com`.

[33] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, September 1998.

[34] V. Roca, L. Costa, R. Vida, A. Dracinschi, and S. Fdida. A survey of multicast technologies. Technical report, Laboratoire d'Informatique de Paris 6 (LIP6), September 2000. `http://www-rp.lip6.fr`.

[35] D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6^{th} European Software Engineering Conf. held jointly with the 5^{th} Symp. on the Foundations of Software Engineering (ESEC/FSE97)*, LNCS 1301, Zurich (Switzerland), September 1997. Springer.

[36] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *Int. Symp. on Software Reliability Engineering*, 1998.

[37] Q. Sun and D. C. Sturman. A gossip-based reliable multicast for large-scale high throughput applications. In *Proc. of the Int. Conf. on Dependable Systems and Networks, New York, NY (June 2000)*, pages 347–358, 2000.

[38] Sun Microsystems, Inc. *Java Message Service Specification Version 1.1*, April 2002.

[39] P. Sutton, R. Arkins, and B. Segall. Supporting Disconnectedness—Transparent Information Delivery for Mobile and Invisible Computing. In *Proc. of the IEEE Int. Symp. on Cluster Computing and the Grid*, May 2001.

[40] TIBCO Inc. *TIBCO Rendezvous*. `http://www.rv.tibco.com`.

[41] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical report, Duke University, April 2000.

[42] Robbert van Renesse. Scalable and secure resource location. In *Proceedings of the IEEE Haway International Conference on System Science*, 2000.

[43] A. Varga. OMNeT++ Web page. `www.hit.bme.hu/phd/vargaa/omnetpp.htm`.

[44] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination. In *Proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.

List of Figures