Pastel
Bridging the Gap Between Structured and Large-State Overlays

Nuno Moura e Silva Cruces

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri
Presidente: Prof. Luís Eduardo Teixeira Rodrigues
Orientador: Prof. Rodrigo Seromenho Miragaia Rodrigues
Vogais: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Prof. Paulo Jorge Pires Ferreira

Novembro 2007
Acknowledgments

First and foremost, I would like to thank my thesis supervisor, Prof. Rodrigo Rodrigues, without whose ideas this work would never have been possible. I particularly appreciated his trusting me with his own original ideas, and allowing me the indulgence to then dictate my own pace, and set my own course. This gave me freedom to breath and explore, which I value immensely.

I would also like express my gratitude to each member of my thesis committee (Prof. Luís Rodrigues, Prof. Teresa Vazão, Prof. Paulo Ferreira and Prof. Rodrigo Rodrigues), for their valuable input to the final version of this document.

Finally, I would like to thank all my colleagues, friends and family who contributed to the work described here, or just supported me during these years. Without their help I would not have reached this moment.
Resumo

Os overlays peer-to-peer pretendem oferecer um substrato único utilizável por múltiplas aplicações de requisitos diversos (simultaneamente, inclusive). Contudo, os sistemas atuais focam-se exclusivamente em oferecer caminhos curtos e rápidos ideais para contactar directamente o responsável por determinado conteúdo, ou caminhos longos, mais lentos, para aplicações capazes de explorar a topologia do overlay (e.g., multicast).

Neste trabalho, é apresentado Pastel, uma extensão do Pastry que tenta fazer a ponte entre estes dois tipos de overlays. Pastel mantém tanto as tabelas de encaminhamento do Pastry, como uma tabela de informação completa, sendo evidenciadas sinergias que podem ser exploradas para manutenção das duas em paralelo. É também proposta uma nova API mais rica que a actualmente oferecida pelos overlays existentes, que permite dar à aplicação controlo sobre o tipo de lookups que estas efectuam (encaminhamento multi-hop estruturado, ou contacto directo).

Implementámos Pastel de forma assíncrona e event-driven, e avaliámos a sua performance no testbed PlanetLab. Os resultados obtidos mostram que os lookups do Pastel são mais eficientes que os do Pastry. Mais, a largura de banda consumida pelo Pastel é modesta, mesmo para sistemas de tamanho moderado, o que demonstra a importância das técnicas inovadoras a que recorremos para manter a largura de banda utilizada por sistemas de informação completa sob controlo.

Palavras Chave

Peer-to-peer, overlay, contacto directo, encaminhamento, estruturado, não estruturado, informação completa
Abstract

Peer-to-peer overlays envision a single overlay substrate that can be used (possibly simultaneously) by many applications, but current overlays either target fast, few-hop lookups for directly contacting nodes responsible for content, or slower multi-hop lookups that can be used by applications that exploit the overlay topology (like multicast).

In this work, we present Pastel, an extension to Pastry that bridges the gap between the these two types of overlays. Pastel maintains both Pastry routing tables and a full information table, and we show how we can exploit synergies between the maintenance of the two. We also propose a novel API that is richer than the one offered by existing overlays, to give applications control over the type of lookups they perform (structured, multi-hop routing, or direct contact).

We implemented Pastel in an asynchronous, event-driven fashion, and evaluated its performance on the PlanetLab testbed. Our results show that Pastel’s lookups more efficient than Pastry’s. Furthermore, the bandwidth required by Pastel is modest, even for moderate sized systems, demonstrating the value in the novel techniques we use to keep bandwidth used by full information systems under control.

Keywords

Peer-to-peer, overlay, direct-contact, routing, structured, non-structured, full information
# Contents

1 Introduction .................................................. 1

2 Related Work .................................................. 5
  2.1 Background ................................................. 5
  2.2 Pastry ..................................................... 8
  2.3 Broadcast ............................................... 11

3 System Overview ............................................... 13

4 System Design ................................................ 15
  4.1 Interface .................................................. 15
  4.2 Node State ............................................... 16
  4.3 Routing ................................................... 18
    4.3.1 Direct Routing .................................... 18
    4.3.2 Structured Routing ................................. 20
    4.3.3 Broadcast .......................................... 21
  4.4 Node Arrival ............................................. 22
  4.5 Node Departure .......................................... 24

5 Implementation ............................................. 25

6 Evaluation .................................................. 27
  6.1 Experimental Setup .................................... 27
  6.2 Bandwidth ............................................... 28
  6.3 Lookups .................................................. 30
  6.4 Unreachability .......................................... 31
  6.5 Replication ............................................. 33

7 Future Work ................................................ 35

8 Conclusions ................................................ 37
List of Figures

2.1 A Symphony network with 1000 nodes (adapted from [21]) . . . 6
2.2 A Chord network with 1000 nodes (extracted from [21]). . . . . 6
2.3 Item responsibility in Pastry. . . . . . . . . . . . . . . . . . . . . 9
2.4 Leaf sets in Pastry. . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.5 Prefix routing in Pastry. . . . . . . . . . . . . . . . . . . . . . . . . 10
3.1 Replica sets in Pastel. . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.1 Structured state of a Pastel node (inspired by [2]). . . . . . . . . 17
6.1 Bandwidth of Pastel. . . . . . . . . . . . . . . . . . . . . . . . . . 29
6.2 Bandwidth of Pastry. . . . . . . . . . . . . . . . . . . . . . . . . . 29
6.3 Bandwidth of Pastel with varying $b$. . . . . . . . . . . . . . . . . 30
6.4 Bandwidth of Pastel with varying $t$. . . . . . . . . . . . . . . . . 32
6.5 Bandwidth of Pastel with varying $k$. . . . . . . . . . . . . . . . . 33
## List of Tables

6.1 Lookups in Pastel with 750 nodes. . . . . . . . . . . . . . . . . . 31  
6.2 Lookups in Pastry with 750 nodes. . . . . . . . . . . . . . . . . . 31  
6.3 Lookups in Pastel with 1500 nodes. . . . . . . . . . . . . . . . . 32  
6.4 Lookups in Pastel with $t = 5$ min. . . . . . . . . . . . . . . . . 33  
6.5 Lookups in Pastel with $k = 5$. . . . . . . . . . . . . . . . . . 34
List of Algorithms

1 Direct routing – send. ................................................. 19
2 Structured routing – route. ........................................... 20
3 Dissemination – broadcast. ......................................... 22
Chapter 1

Introduction

Peer-to-peer systems have become notorious through the popular large-scale, distributed file-sharing applications, that account for much of today’s Internet traffic. This architecture is based on the concept of a network of equals (called peers), where by design each participant has equivalent responsibilities and status, and participants are able to collaborate towards a common goal, without the need for central coordination. These systems are typically implemented through an application-layer overlay network, built on top of the existing Internet. Despite its roots in the commercial world, the subject has quickly garnered interest within the scientific community.

Peer-to-peer overlays like Chord [1], Pastry [2], Tapestry [3] or CAN [4], form a decentralized, self-organizing substrate that can be used by a myriad of different applications with distinct requirements. In many cases the designers of such overlays have expressed the vision of deploying a single peer-to-peer overlay, on which many disparate applications are deployed (e.g., [5, 6]).

We can categorize the applications that have been proposed to run on these peer-to-peer overlays in two broad groups.

The first group consists of direct-contact applications, that typically use a narrow get/put interface offered by a thin layer running over the routing overlay, implementing a distributed hash table interface (DHT) [7, 8]. These applications (or the DHTs that underlie them) use the routing overlay simply to locate the node (or set of nodes) responsible for certain information. The nodes are then contacted directly to store or retrieve the data. Examples of such applications include distributed file systems [9], or citation databases [10].
The second group consists of routing applications, which are applications that actively use the topology formed by the routing overlay. For example, a multicast application (one that disseminates information across a set of nodes taking advantage of common links between them) can form multicast trees by taking the union of the lookup paths to a common node [11].

In previous work, authors have presented proposals for reducing the lookup latency of peer-to-peer overlays by increasing the amount of routing state maintained by each node [12, 13, 14, 15]. Such overlays achieve faster lookups because increasing the knowledge of each node about other members of the network will lead to shorter lookup paths, or ultimately preclude routing (i.e., given full information, a one-hop lookup). However, such overlays cannot be effectively used by routing applications, where structured multi-hop paths are required.

This divide between overlays that support structured multi-hop lookups and overlays that keep a large routing state conflicts with the vision of a single overlay that can support multiple disparate applications (either simultaneously or otherwise). This is so, because routing applications cannot be deployed in overlays with large routing state (since they lack both the interface to contact nodes along a lookup path, and the topology formed by a structured overlay), and direct-contact applications pay the penalty of slow lookups if they are run on a small state overlay.

For a system to support adequately both direct-contact and routing applications, it must provide a richer interface than the one currently supported by existing peer-to-peer overlays [16]. That interface should enable the use of short lookup paths (or, even better, direct lookups) for direct-contact applications, and long, proximity-based lookup paths for routing applications.
This work presents Pastel, an extension of Pastry [2], designed to bridge the gap between the two types of overlays discussed above. The routing tables in a Pastel node can be functionally divided in two parts:

- a structured part that resembles a Pastry routing table, used by routing applications;
- and, an unstructured part that maintains a full membership information table with a large number of entries, to support efficient lookups for storage applications.

This extended routing state allows us to maintain the Pastry functionality that is used by routing applications, and to extend it with highly efficient (i.e., low latency) direct-contact primitives for those remaining applications.

Reduced latency is not the only advantage of Pastel, however. As we will show, there are synergies between the two parts of the routing state that improve maintenance protocols.

The design of Pastel also raises interesting issues, like how to control the extra bandwidth required to maintain full membership information at each node. We introduce a distinction between strong links (that are aggressively kept current), and weak links (that have a delayed response to unreachability) for the routing state in Pastel. We will further show how applications can successfully deal with the presence of weak links, without having their correctness or performance negatively affected.

After preliminary simulations on a discrete-event packet level simulator called p2psim [17], we implemented a state-of-the-art version of the Pastry protocol [18], to which the mentioned Pastel extensions were subsequently added. This allowed us to perfect our design, and to measure the real-world efficiency of direct-contact lookups and maintenance overhead introduced by Pastel, as well as to compare both to those of its ancestor Pastry.

Our results show that Pastel can achieve lookups that perform better than Pastry’s for direct-contact applications: in the wide majority of cases, multi-hop routing is not required by these applications. Our results also demonstrate that the bandwidth required to maintain the extra state is modest, even for a system with thousands of nodes. Furthermore, we achieve this without sacrificing the multi-hop routing interface required by routing applications.

The remainder of this document is organized as follows. Chapter 2 will try to put Pastel in context, with a broad look at previous work. Then in Chapter
3, we will give a brief overview of our system, with Chapter 4 further detailing our design. Chapter 5 reports on how we have approached the implementation of Pastry and Pastel, and Chapter 6 shows an experimental evaluation of that implementation. Chapter 7 introduces possible future work on Pastel, after which we conclude in Chapter 8.
Chapter 2

Related Work

Since we set to design a new peer-to-peer overlay, it is important to start by exploring existing solutions to the same problem. Thus, the following section presents a broad survey of the current state of the art, after which we take a closer look at the work Pastel builds upon, in the latter sections of the chapter.

2.1 Background

For the purpose of this work, there are two important properties of peer-to-peer systems that we need to analyze: the overlay structure, or network topology; and the amount of membership information gathered at each node. These properties are not entirely independent, but they do not necessarily overlap either.

An overlay can be denoted by a graph, where vertexes represent system nodes, and edges logical links between nodes. Being commonly implemented over the Internet, these networks have each node be identified by a public \((IP, port)\) pair. To a link between nodes, corresponds the concept that two nodes know each others identifiers, and can thus communicate between themselves.

Given this definition, unstructured overlays separate themselves from structured overlays, by virtue of the graphs that represent them showing no apparent structure, \(i.e.\) the set of edges departing from a given vertex follows no apparent pattern – see Figure 2.1. The classic example of this kind of overlay is Gnutella [19], but other systems, like Accordion [15] or Symphony [20], fit this
definition. Namely, overlays on which each node maintains global membership information (e.g., [14]), can be thought of as unstructured, since neighbors to a particular node follow no special pattern – they are ideally all nodes in the system.

In structured overlays, on the other hand, the corresponding graph does present an interesting topology – see Figure 2.2. In these systems each node follows rather strict rules when picking its neighbors. Examples of this type of overlay are Chord [1] and Pastry [2].

Structured systems were initially conceived as a way to fight inefficiencies that were thought to be inherent to unstructured overlays. Lack of structure seemed to imply lookups must use inefficient techniques, like flooding.
or random walks. Systems like Accordion and Symphony, disprove this theory though. Despite being based on randomized neighbor selection, or actually because of their probabilistic neighbor selection, they achieve better routing performance than many structured overlays. Full information systems that preclude routing entirely, permitting direct contact, also combine lack of organized structure with excellent lookup performance.

More importantly however, is that the topology formed by a structured overlay, can be an advantage in and of itself. As we have previously discussed, routing applications can often take advantage of the topology of the underlying overlay. Structured, proximity-based overlays like Pastry, have been shown as great tools in the implementation of efficient application level multicast [11], and broadcast [22]. This is so, because Pastry’s structure allows the application to reuse network links to great effect, when disseminating information.

On the other hand, direct-contact applications based on DHTs (e.g., [7, 8]), benefit first and foremost from reduced latencies. For this type of application, systems like Accordion, that have the flexibility to keep more routing state, and specially systems that keep full membership information, are far more desirable. In [12] we find a detailed study that concludes that, for the scale of present-day systems, and for direct-contact applications, one-hop overlays will typically deliver the best performance.

However, and as we can gather from [14], any sophisticated implementation of a full information system ends up introducing some level of structure. This stems from the fact that membership changes must be disseminated throughout the entire network, if you are to keep comprehensive membership information. Note how we have previously remarked that structured overlays excel at disseminating information.

As such, the idea of combining a proximity-based, structured system with a full-information system, naturally arises. Not only should such a system better support complex applications with disparate requirements, but certain synergies in the maintenance protocols themselves may very well be explored.

For a system like this to conveniently support both direct contact and structured, proximity based routing, its API needs to extend the one presently offered by most peer-to-peer systems to date [16], with primitives that allow a distinction between both forms of routing.
In designing an overlay to meet these requirements, we chose to adapt Pastry, taking advantage of its structured overlay, and combining it with a streamlined full-information system. This choice was due to two major factors, mostly:

- Pastry’s locality properties, which are responsible for its notable lookup performance [23];
- and, its known adequacy as a substrate for typical routing applications, like those dependent on multicast or broadcast.

We will briefly describe this system in the following section.

The design of the full information system with which we extend Pastry, is relatively simple. Based on a broadcast primitive we will describe in the final section of this chapter, we are able to disseminate membership changes throughout the network, which is the only requirement to implement such a system.

Nonetheless, maintaining full membership information is an expensive task. So, and as a way to obviate this burden, we have decided not to be aggressive when maintaining this information. With a delayed response to unreachability, it is possible to avoid disseminating information about transient departures from the network. According to a recent study on the life-time of nodes in typical peer-to-peer networks [24], this may well be an important optimization, as it is fairly common for a node to abandon the system only to return a little later.

### 2.2 Pastry

In this section we summarize the design of Pastry [2], the peer-to-peer overlay we used as starting point for the design of Pastel.

A Pastry system is a self-organizing overlay of nodes, each of which is assigned a 128-bit identifier. These identifiers are ordered in a circular space modulo $2^{128}$, and are assumed to be uniformly distributed (e.g., they can be computed as a secure hash of the node’s network address).

Pastry applications use items to partition their workload. Items are abstract entities, which are assigned identifiers in a similar way as nodes. Each item is ascribed a responsible node in the overlay, which is the numerically closest live node in the ring of identifiers (see Figure 2.3).
Pastry offers a route primitive, that given a message and an item, routes the message to the responsible node. This routing algorithm depends on keeping fundamentally two sets of state.

The leaf set consists of a list of $l$ nodes (where $l$ is a system parameter), with the $l/2$ numerically closest identifiers to the local node, in both directions of the identifier space (see Figure 2.4). The correct maintenance of this information is essential to the guarantee that messages are delivered to the responsible node.

The routing table keeps the information of approximately

$$n = (2^b - 1) \left\lceil \log_2 N \right\rceil$$

nodes (where $b$ is also a system parameter, and $N$ is the number of nodes in the network). The routing algorithm interprets identifiers as a sequence of $b$-bit digits (i.e., in base $2^b$), and routes messages digit by digit, through nodes that share increasingly larger digit prefixes with the item (see Figure 2.5). A message is routed in less than

$$s = \lceil \log_2 N \rceil$$

steps under normal operation with this procedure.
Figure 2.4: Leaf sets in Pastry. The leaf set for node 32003210 consists of the 2 nodes immediately to the left, and the 2 to the right in the identifier ring (with $l = 4$).

Figure 2.5: Prefix routing in Pastry. Node 10233102 routes a message to item 32010123. This identifier is resolved digit by digit (3..., 32...), until the leaf set is reached at node 32001230 ($b = 2$). From there the message is routed directly to node 32003210, the responsible for the item.
Node joins and departures are handled as follows. When a node $x$ joins the overlay, it initializes its state by contacting an existent node $a$, asking it to route a special join message to the identifier of node $x$, resulting in some responsible node $z$. Then, $x$ obtains the leaf set from node $z$, and each $n$th row of the routing table from the $n$th node encountered along the path from $a$ to $z$.

To handle node departures, nodes periodically exchange keep-alive messages. Leaf sets are symmetrical, and it suffices for a node to periodically send a message to all its neighbors. Routing tables on the other hand are not, so a node must periodically request all members of its routing table to send it a message. Leaf sets are repaired by including information about a few leaf set members in the message periodically sent to all leaf set members. Routing table entries are repaired more lazily, by periodically requesting random routing table entries for their routing tables, and selecting the best nodes from both tables for each entry (which also serves the purpose of optimizing query latency).

### 2.3 Broadcast

This section summarizes the work on which the broadcast primitive of Pastel was based upon.

A full information system requires an efficient broadcast primitive to disseminate membership information across the network. Given a broadcast primitive, a joining node can announce its arrival to the entire network, and departures can be disseminated by the neighbors of those departing nodes.

The broadcast algorithm chosen for Pastel was inspired by the one described in [22, 25], which follows quite elegantly from Pastry’s design. Each non-leaf node forwards at most

$$m = (2^b - 1) \lfloor \log_2 N \rfloor$$

messages on average with this procedure. The chosen method permits delivery of the message to live nodes of a given prefix, even if the node responsible for forwarding the message to such nodes is unaware of such nodes.

A simpler version of the algorithm that does not make this guarantee (but which is the basis for all versions of the algorithm), can be generically described as follows. The initiating node $x$, broadcasts a message by sending it to all nodes $x'$ in its routing table. Each message is marked with a parameter $n$, for
the $n$th row of $x$’s routing table where node $x'$ was found. When a node $x'$ receives a message for broadcast marked with $n$, it forwards the message to all nodes in its routing table from rows $n' > n$. The process terminates once there are no further nodes to which to forward the message to.
Chapter 3

System Overview

In this chapter we present a very brief overview of how we plan to extend Pastry to meet the goals we set for ourselves with Pastel.

We consider the Pastry system organization where nodes are assigned a random 128-bit identifier, and the identifiers are ordered in a circular identifier space modulo $2^{128}$.

As in Pastry, we assume applications use items to partition their workload, and assign to each item a responsible node in the overlay, which is the numerically closest live node. However, we also need to take into account a variant of this when the application uses replication, in which case items have a set of responsible replicas, which we will consider to be the set of $k$ nodes ($k \leq l$ is another system parameter) with the $k/2$ numerically closest identifiers to the item, in both directions of the identifier space (see Figure 3.1).

To maintain the routing capabilities of the overlay, and enhance the performance of direct-contact applications, we extend Pastry to maintain, side by side, two sets of routing state: the Pastry leaf set and routing table; and, a full information table. This additional routing information will enable an extended API that satisfies both direct-contact applications, and applications that rely on structured, proximity-based routing.

Central to our design is the concept of a delayed response to unreachability. Bandwidth in full information systems is a concern, because churn can lead to significant maintenance traffic if tables are to be kept current. Traffic that is linear in the number of nodes in the system is unavoidable, but linearity in the number of membership changes can be avoided given typical node typical lifetime distributions. Our system introduces the concept of weak links, mem-
Figure 3.1: Replica sets in Pastel. The responsible replicas for item 32010123 are nodes 32003210 and 32013232 (with $k = 4$). This set always includes the responsible node (here node 32003210).

bership information that is barely maintained in an attempt to reduce such bandwidth costs. This stale information is nevertheless still useful, as through the use of application-level replication, it can offer interesting performance improvements all the same.

We envision that applications with mixed requirements are those that would benefit the most from having these two styles of routing available in the same overlay. Unicast, anycast, multicast and broadcast are all well supported, and are usage patterns shared by many applications. For example, a file sharing system might use broadcast to perform complex queries on shared files, unicast to gauge the responsible node for a file that has already been identified, and multicast groups to manage groups of nodes sharing and downloading the same files.
Chapter 4

System Design

In this chapter we will present the design for the Pastel system in further detail. We start by describing the interface applications use to interact with the routing substrate. After that, we detail what local state is kept by each node, how that state is used for routing different requests, and finally, how that state is maintained in the presence of churn.

4.1 Interface

With the two previously mentioned classes of applications in mind, we devised the following application programming interface, which we briefly outline.

An application interacts with the underlying Pastel substrate in an event-driven and asynchronous fashion: provided primitives are asynchronous in that they immediately return control to the caller, and system events are handled through the use of callbacks that applications must implement.

Note that the presented interface is slightly simplified for clarity. The extension to Pastry (and to the API proposed in [16]) consists mostly of the send and broadcast primitives.

Primitives

\texttt{join(node=\textit{nil})} – allows the local node to either join an existing Pastel overlay network, by referencing an existing node, or to bootstrap its own, initializing all relevant state.
part() – abandons the network in an orderly fashion, performing all necessary local cleanups.

send(msg, dest, rep=false) – sends the message directly (i.e., using full membership information) to the node responsible for item dest, contacting any of k replicas if rep is true.

route(msg, dest, rep=false) – routes the message through the structured overlay to the node responsible for item dest, stopping at the first encountered of k replicas if rep is true.

broadcast(msg, prefix=ε, depth=∞) – broadcasts the message throughout the structured overlay; only nodes with the given prefix, and only up to $2^{b \times \text{depth}}$ nodes uniformly spread over that prefix space are contacted.

Callbacks

joined() – callback invoked when the node has completed the join procedure, and is ready to start processing messages.

leafs(set) – callback invoked when membership changes in the node’s leaf set are registered.

deliver(msg, dest) – callback invoked when a message is received and the local node is the recipient for that message.

forward(msg, next, dest) – callback invoked when the local node is about to forward the message to node next, en route to item dest; the node is permitted to stop further routing of the message.

4.2 Node State

Each Pastel node maintains a leaf set, a routing table, and a full information table.

The leaf set and the routing table are identical to the ones implemented in Pastry (as introduced in Section 2.2), and the system tries to populate these with reachable nodes, thus they must be kept current rather aggressively, especially the leaf set.
Figure 4.1: Structured state of a Pastel node (inspired by [2]). Leaf set and routing table for node 10233102 ($b = 2$ and $l = 4$).

The leaf set tracks the $l/2$ numerically closest identifiers to the local node, in both directions of the identifier space. The routing table has $2^b$ columns, and $128/b$ rows. The $2^b$ entries of row $n$, are filled by nodes that share an $n$-digit prefix with the local node, and which $(n+1)$th digits correspond to the column they occupy (see Figure 4.1).

The full information table does not need to be as aggressively maintained. In fact, we deliberately allow unreachable nodes to remain in this table for some time period (we call these entries weak links). This is because the correctness and liveness of Pastel, and the applications that use it, do not depend on the freshness of the information present in this table. The problem of performing lookups using these weak links is address in the next section. This table keeps data about all system nodes. In particular, for each node, we maintain its identifier, network address and freshness (time when last contacted by the node).

The storage cost of this data structure is acceptable, especially if secondary storage is considered. For instance, even 1 million IPv4 nodes with 128-bit identifiers, require no more than 25 MiB of storage. Given this size, though, this table may need to be implemented using a disk-friendly data structure such as a B-tree.
Bandwidth costs might be an issue, since nodes can have short sessions [24, 26], and this leads to a large number of notifications about routing information being sent to everyone in the system. However, weak links do not generate much maintenance traffic. This is because we employ a strategy of delayed response to unreachability: we wait for a certain amount of time $t$ before we remove unreachable nodes from the full information table. Therefore, we do not trigger events due to transient failures, or even a large percentage of temporary disconnections. The importance of this strategy is substantiated by the above experimental studies, that have shown that despite short sessions, nodes in peer-to-peer overlays tend to have much longer membership lifetimes, or in other words when they disconnect from the system, they tend to reconnect not much later on.

Keeping full maintenance information can be advantageous to the structured routing protocol itself, as it may lead to better choices when replacing entries in the routing table, as well as permit a speedier recovery from massive failures. In the next sections we will show how the maintenance of the full information table benefits significantly from their structured counterparts.

4.3 Routing

Pastel routes different message types in different ways, supporting the requirements of diverse applications. This section discusses the multiple routing methods offered by Pastel, that correspond to the `send`, `route` and `broadcast` primitives we have introduced. The first two allow both unicast, and anycast to a replica set; the third implements broadcast. Group multicast can be implemented by an application like [11].

4.3.1 Direct Routing

This routing method is the simplest of all three. It takes advantage of the full information table to send a message directly to the node responsible by the specified item. In case the responsible node is not reached directly, we revert to structured routing. Algorithm 1 implements this procedure, which we detail in the next paragraph.

Given a message, a destination item, and whether or not we which to use replication (line 1), we first check if the local node is itself responsible for the
**Algorithm 1** Direct routing – *send*.

```python
def send(msg, dest, rep=False):
    if responsible_for(dest, rep):
        deliver(msg, dest)
        return

    if local_invocation():
        for next in full_info_table.next(dest):
            next = forward(msg, next, dest)
            if next:
                rpc(next, send, msg, dest, rep)
            else:
                route(msg, dest, rep)
```

item (line 2). As we can see this procedure takes account replication, considering any of the  \( k \) replicas responsible for the item. If that is indeed the case, and we are responsible for the specified item, we deliver the message by invoking the corresponding callback (line 3), after which we are done (line 4). In the common case where we are not responsible for the item, we check if this is a local, or remote invocation (line 6). If it is a local invocation we will forward the message two the \( k \) closest nodes known to us, using the full information table (line 7); that is, after invoking the `forward` callback (line 8), and making sure the application is not signaling us not to forward the message (line 9), we make a remote procedure invocation of the same `send` method on the specified node (line 10). Going back to the case where this has already been a remote invocation (line 11), we fall back to using the `route` procedure (line 12).

Note how this method forwards the message in parallel to, not one but a set of nodes, regardless of replication. This heuristically assumes that at least one of those nodes shall be alive, and is either responsible for the item (case in which the lookup is satisfied in one hop), or knows directly the responsible item through its leaf set (case in which the lookup is satisfied in two hops). These cases cover the vast majority of lookups. Obviously in such a situation, both the initiating node and the destination nodes, must correctly support message duplication in requests as well as in responses. In the particular case where replication is used, the node that made the initial request may then decide to either accept the first response (which further improves latencies),
Algorithm 2 Structured routing – route.

```python
def route(msg, dest, rep=False):
    if responsible_for(dest, rep):
        deliver(msg, dest)
        return

    if leaf_set.contains(dest):
        next = leaf_set.next(dest)
    else:
        next = (routing_table.next(dest) or
                routing_table.nearer(dest) or
                leaf_set.nearer(dest))

    if next:
        next = forward(msg, next, dest)
    if next:
        rpc(next, route, msg, dest, rep)
    else:
        deliver(msg, dest)
```

or take a majority vote (which can improve reliability to the application). All these decisions are left for applications to deal with.

### 4.3.2 Structured Routing

Pastel’s structured routing algorithm, follows that of Pastry’s precisely, using exclusively the fresh information kept in leaf sets and routing tables. Routing progresses by digit prefixes, which means under normal circumstances only

\[
s = \lceil \log_2 N \rceil
\]

steps are required to route a given message. Under a significant number node failures, the algorithm is better than linear in the number of nodes in the system (a detailed analysis can be found in [2]). The description of Algorithm 2 is presented in the next paragraph.

Given a message, a destination item, and whether or not we which to use replication (line 1), we first check if the local node is itself responsible for the item (line 2). If that is indeed the case, and we are responsible for the specified item, we deliver the message by invoking the corresponding callback (line 3),
after which we are done (line 4). In the common case where we are not responsible for the item, we will select a node to which to forward the message to. If the item is covered by our leaf set (line 6), that node can be obtained using the leaf set (line 7), which gives us the nearest node to the item. Otherwise (line 8), we retrieve the node from the routing table by prefixes (line 9), or if such a node is unavailable, we get any node closer to the item from either the routing table (line 10) or the leaf set (line 11). Such a node would always exist unless we were the responsible node. If we are forwarding the message (line 13), we then invoke the appropriate callback (line 14), giving the application the opportunity to stop routing locally (line 13), before making a remote procedure call of the same route method on the found node (line 16). If routing was stopped (line 16), we deliver the message to the local node, invoking its callback (line 18).

4.3.3 Broadcast

This broadcast algorithm was derived from the one introduced in Section 2.3. It is a recursive version of the one there described, but it is essentially the same. We allow the broadcast tree to be pruned at a given depth, limiting the number of contacted nodes to $2^b \times \text{depth}$, case in which at most $(2^b - 1) \times \text{depth}$ messages are sent by each node. If the depth is unlimited, and when $2^b \times \text{depth} > N$, each node will forward

$$m = \left(2^b - 1\right) \left\lceil \log_2 N \right\rceil$$

messages on average. Algorithm 3 is detailed in the next paragraph.

Given a message to broadcast, the prefix to which it should be broadcasted, and the depth at which to prune the broadcast (line 1), the algorithm proceeds in the following way. If the node’s own identifier is not prefixed by the given digit prefix (line 2), the node to which to forward the message is selected in a manner similar to route (lines 3–8). A prefix for these routing purposes, corresponds to an item placed in the middle of that prefix’s identifier space. After having selected such a node, the application is notified of the routing substrate’s intention to forward the message through the appropriate callback (line 10), and if it confirms that intention (line 11), the message is forwarded by performing a remote procedure call on that node with the appropriate parameters (line 12). If on the other hand, the prefix does prefix the local node, there are two alternatives. Either routing is finished because the prefix already
Algorithm 3 Dissemination - broadcast.

```python
def broadcast(msg, prefix=empty, depth=infinity):
    if not prefixes(prefix, local):
        if leaf_set.contains(prefix):
            next = leaf_set.next(prefix)
        else:
            next = (routing_table.next(prefix) or
                    routing_table.nearer(prefix) or
                    leaf_set.nearer(prefix))
        next = forward(msg, next, prefix)
        if next:
            rpc(next, broadcast, msg, prefix, depth)
    elif prefix == local or depth == 0:
        deliver(msg, prefix)
    else:
        for digit in range(0, 2^b):
            broadcast(msg, prefix + digit, depth - 1)
```

matches the local node's identifier in full length, or the depth is already zero (line 14), cases which the deliver callback is invoked; or further digits can be added to the prefix, and the procedure be recursively applied (lines 16–18).

Note that, despite the $2^b$ recursive calls (lines 17–18), only one of those calls will possibly lead to further recursive calls, as only one of the resulting prefixes will prefix the local node (line 2). This works to make the number of recursive calls linear in both depth and identifier length. The procedure is also bound to terminate, as either prefix size or depth are limited (line 14). Finally, observe that despite the different presentation, this algorithm is functionally equivalent to the one previously introduced: the same nodes are contacted as a result of both strategies.

### 4.4 Node Arrival

When a node arrives, it must initialize all relevant state and let others know it joined.

The first steps of the join protocol are identical to Pastry's (as described in Section 2.2): the incoming node asks a know overlay member to route a special
join message to its identifier, and the contents of the structured tables (routing
table and leaf set) of all the intermediate nodes contacted along the path to the
node responsible for that identifier, are used to initialize the structured tables
of the incoming node.

At this point, the full information table is mostly empty, as the node knows
of the existence of very few system nodes; in fact, only about
\[ n = l + (2^b - 1) \lceil \log_2 N \rceil \]

nodes (where \( N \) is the number of nodes in the system) are known to the joining
node. Also, most system nodes know nothing of the existence of that incoming
node.

Although this does not affect the system’s correctness (all messages are still
delivered to their correct responsible nodes), it affects its performance, in par-
ticular the latency of direct queries originating from, or targeting the joining
node. To address this, we need to disseminate the join information to the re-
mainng system nodes, and the joining node must gather the full membership
information.

The protocol for disseminating the information about the node join closely
mimics the broadcast primitive (discussed in Section 2.3). Routing table en-
tries are contacted and made responsible for informing nodes that share their
prefixes about the join. A node should however locally terminate the routing
of such messages if it determines the joining node was recently present in the
overlay, case in which it will still be in most node’s full information tables.
Also, when a node finally delivers the join messages to nodes in its leaf set, or
when it decides to terminate further routing, it should report back to the join-
ing node. This report contains information about all nodes of the appropriate
prefix, and allows the joining node to gather complete membership informa-
tion incrementally; it also aggregates this information in bite sizes of a few
nodes, which somewhat reduces transport-layer overhead.

Note that, since correction is ultimately guaranteed as in Pastry, by the
leaf sets and normal routing mechanisms, all these messages are low prior-
ity, and might be piggybacked with other maintenance traffic to further reduce
transport-layer overheads.
4.5 Node Departure

When a node leaves the system, this is quickly noted by members of its leaf set, who will evict this node from that set as in Pastry (see Section 2.2). These nodes then start a timer with time $t$ plus some small random value. If the node shows no activity before the first of these timers expires, the node whose timer expired first will broadcast a special part message on behalf of the departed node.

Upon receiving a part message for a given node, all information about this node should be deleted, including information present in the full information table and pending timers related to that node.

Also, to prevent information about dead nodes from building up in the full information tables of long-running processes (e.g., in the event that messages got lost), nodes should periodically perform a local vacuum operation on the table, pruning nodes they have not heard of in a very long time (i.e., obviously more than the wide majority of node’s life-times). Furthermore, if a live node detects it is being announced as dead, it should wait for some random time before again announcing its arrival. These conditions combine to guarantee that in the event of race conditions, and eventually, most live nodes of the system will be wildly know as live, and all dead nodes will be evicted from full information tables.

Again, we stress that this scheme leads to a significant fraction of unreachable nodes in the full information table of any give node, but we leave the solution to this problem to the higher-level layers, for instance using replication and waiting for a reply from any replica.
Chapter 5

Implementation

We will now describe in some detail our realization of the previously introduced design.

We started out by implementing our design in a discrete-event packet-level simulator, specifically tailored for the testing of peer-to-peer applications and overlays, called p2psim [17]. This was chosen because it showed to be the best option to test system performance parameters like: latencies, number of query hops, bandwidth, etc. Other options evaluated included PlanetSim [21] (that deserves a mention for the interesting topology visualization options it includes, see Section 2.1), and FreePastry [27] (that also seconds as a real-world implementation, but which at the time, showed to be better suited at implementing Pastry applications, rather than at customizing the Pastry engine itself). This initial implementation allowed us to iron out some issues with our early design. It also allowed us to get a grip on how exactly the overlay would behave under larger, albeit simulated overlays.

After this early experimentation, we decided on implementing a usable, real-word version of the protocol, that could be tested and deployed on real IP networks, the Internet in particular. Armed with our initial experience, we decided to base our implementation in the more asynchronous and stateless version of Pastry presented in [18]. This was implemented in the Python programing language [28], using the Twisted event-driven networking engine [29], which we found to be perfect for our purposes.

We produced implementations of both Pastel and Pastry (Pastel’s building upon Pastry’s), so that we could compare performance between them on even ground. Both implementations lack some more complex optimizations that
are left for future work. The implementation is modular, and was kept under a pretty manageable size (in the order of 1000–1500 SLOC, depending on whether you count comments and/or empty lines, and how much test code is counted), meaning that it can be easily modified and extended.

Messages in our implementation are forwarded mostly using UDP with no acknowledgments. TCP is sparingly used, only to send messages larger than (a conservative estimation of) the MTU, or those that are likely to cause contention on the receiver’s side (to avoid having to implement custom congestion control algorithms on top of UDP). This strategy is efficient, but messages can be lost both to the network, or more worryingly to a faulty node. To guarantee that a high percentage of messages make through our network unhampered, we need to keep routing table and leaf set entries relatively fresh, using the periodic probes and keep-alive messages we have described in Section 2.2. The remaining message losses are left for the application to deal with, using appropriate application-specific timeouts and retransmissions.

Using UDP results in better latencies, as well as less transport-layer overheads. This is coupled with efficient, binary representations of data, to produce the good results we will present ahead. For instance, we take advantage of the fact that node identifiers depend only on a node’s \( (IP, port) \) pair, to at all times transmit only that information instead of, or in addition to, the node’s identifier.

Example applications using Pastel have not yet been implemented, and are left as future work, but given Pastry’s event-driven API, Pastry applications like Past [8] or Scribe [11] and others (see [27]), should be quite easy to port over.
Chapter 6

Evaluation

This section attempts to demonstrate the benefits of using Pastel, through real-world test results. It shows that the maintenance bandwidth required by Pastel is modest, and how we can improve on Pastry’s lookup performance.

6.1 Experimental Setup

This evaluation uses the implementations of Pastry and Pastel we described in Chapter 5. We wrote a test application that can be run on top of either overlay with a single configuration change, demonstrating the benefits of the common API we devised. It serves as a nice example of the ease with which new applications can be built.

The test application was deployed on several hundred nodes of the PlanetLab research network [30, 31]. Each PlanetLab node runs one or more independent instances of the test application that form the overlay. These Pastry or Pastel instances log system events to a central log server, that then persists the data to an SQL database for later analysis. The instances communicate between themselves only through the peer-to-peer overlay, with the central log server being used only for information gathering.

The collected data includes not only system events such as joins and parts, but also a log of all messages sent, routed, broadcasted, forwarded and delivered at each node. Each node further logs bandwidth statistics by message type for all traffic, including maintenance traffic. All messages sent and routed through the overlay are replied to, which allows the originating node to log
their round-trip time. All this data allows us to mine for important information such as bandwidth usage, message delivery ratios, average and median round-trip times and hops per message. It also helps us to monitor the system for correction on larger scale tests.

Our tests span for 8 hours with results collected in the later half of those 8 hours. Node lifetimes were modeled through a long-tailed Pareto distribution of shape parameter \( k = 1 \), with median lifetimes of 1 hour [15]. When logged, a node uniformly sends messages of all types to random destinations; when absent, a node models an eventual return through an exponential distribution with an expected value of 1 hour. The default system parameters used in these tests were \( b = 2, l = 8, k = 3 \) and \( t = 1 \) hour, with around 750 nodes. Results presented here were averaged over multiple test runs.

### 6.2 Bandwidth

Our first experiment compares Pastel with Pastry in terms of bandwidth. We compared the average bandwidth consumption from nodes in both systems, during the interval when measurements were taken. Figures 6.1 and 6.2 show that Pastel adds some significant traffic to manage membership changes, traffic which is negligible in Pastry.

Still, even on a system with several hundreds of nodes, bandwidth consumption can be quite modest. As our results show, a 1500-node system needs only 10 B/s per node on average to manage node arrivals and departures.

Despite the fact that this traffic grows linearly with the number of nodes in the system (which is to be expected from a full-information system), this low bandwidth consumption means that our system is likely to scale well from systems with a few thousand nodes, to say a hundred thousand nodes (where linear extrapolation would indicate a consumption of 1 kB/s per node dedicated for membership management).

One of the aspects that contributes significantly towards these good results is the adequate choice of system parameters. In particular, recommended values for Pastry parameters \( b \) and \( l \) are usually \( b = 4 \) and \( l = 16 \), whereas we use \( b = 2 \) and \( l = 8 \). The reason is two-fold.

First of all, as Figure 6.3 shows, reducing \( b \) leads to reduced membership maintenance traffic in Pastel. Since, as we will show in the next section, Pastel
Figure 6.1: Bandwidth of Pastel.

Figure 6.2: Bandwidth of Pastry.
optimizes lookups significantly, increased lookup latency due to reducing \( b \) is no longer a concern, which makes this a net-gain for Pastel.

Secondly, and though this choice of \( b \) to favor Pastel does put Pastry at a disadvantage when compared to Pastel in the next section, unfortunately the tested networks are too small to make the comparisons meaningful for larger values of \( b \).

The \( l \) parameter is usually \( l = 2^b \) or \( l = 2^{b+1} \), which is why we chose \( l = 8 \) here.

### 6.3 Lookups

The next set of experiments examine the efficiency and reliability of lookups in both Pastel and Pastry. Here \texttt{send} corresponds to an anycast to any one replica, and \texttt{route} corresponds to structured routing to the responsible node.

As we can see in Tables 6.1 and 6.2, Pastel isn’t quite able to keep the reliability of Pastry using only 3 replicas. Still, for a system where routing information is allowed to be kept out of date for over 1 hour, a 92% successful delivery ratio for direct lookups seems acceptable. As we will show, increasing the replication factor \( k \) actually helps Pastel expand on Pastry’s reliability.

Also clear from our experiments, are the improvements offered by Pastel in terms of the ratio of direct lookups. With 97% of lookups satisfied in one hop, the average number of hops per lookup approaches 1 – instead of about 3 – for Pastel’s \texttt{send} primitive. Replication also helps Pastry, as evidenced by the
lower average number of hops with send vs. route in Table 6.2, but it is not nearly as much an improvement as the one offered by Pastel.

Finally, latencies are also significantly reduced in Pastel. Tables 6.1 and 6.2 show that Pastel can cut round-trip times in half. Note that these are round-trip times, as that is the only reliable way to measure latencies in a distributed environment. Once a message is delivered to the responsible node, that node replies back to the sender directly. Thanks to the number of hops per lookup, this reply should be responsible for about half of Pastel’s send round-trip times, but maybe down to a fourth for the remaining values, which makes Pastel’s times look even more impressive. On the other hand, the fact that we have not yet implemented all of Pastry’s locality optimizations, means that it is possible there are some improvements to be made there.

Table 6.3 shows that Pastel scales rather with an increasing number of nodes. While the median latency and average number of hops of structured lookups sees a slight increase, both the reliability and the number of direct lookups are largely unaffected by this network size increase.

### 6.4 Unreachability

To understand the impact of having a delayed response to unreachability (i.e., waiting before we declare an unreachable node to be removed from the system)
on both bandwidth and lookup performance, we tried reducing the time \( t \) that we wait before announcing the departure of a node to just \( t = 5 \) min. As Figure 6.4 shows, varying \( t \) greatly increases the amount of bandwidth spent on processing membership changes. In particular, messages related to node departures now account for nearly half of all traffic.

The effect on lookup performance is perhaps less clear. As we can see in Table 6.4, send delivery ratios do increase substantially, but at the same time the number of messages delivered in one hop apparently decreases. There is also a noticeable decrease on latencies of direct lookups.

Despite the improvements though, these results demonstrate that our approach to unreachability is sound, in that it significantly reduces maintenance traffic related to churn in a full information system, without significantly hampering lookup performance. And as we will show in the next section, there are other, better ways to improve lookup performance, which do not incur in such high bandwidth costs.
<table>
<thead>
<tr>
<th>Pastel</th>
<th>send</th>
<th>route</th>
</tr>
</thead>
<tbody>
<tr>
<td>delivery rates (%)</td>
<td>99.1</td>
<td>94.4</td>
</tr>
<tr>
<td>delivered in one hop (%)</td>
<td>95.0</td>
<td>5.75</td>
</tr>
<tr>
<td>average hops (1)</td>
<td>1.05</td>
<td>3.21</td>
</tr>
<tr>
<td>median round-trips (s)</td>
<td>0.08</td>
<td>0.26</td>
</tr>
</tbody>
</table>

Table 6.4: Lookups in Pastel with \( t = 5 \) min.

Figure 6.5: Bandwidth of Pastel with varying \( k \).

### 6.5 Replication

We tried to analyze the effect of increasing replication factors in the quality of lookups by rerunning our tests with \( k = 5 \). Figure 6.5 shows that increasing replication has virtually no impact on either maintenance or application traffic. This is an all around positive thing, as replication is often necessary as a tool to build reliable applications, and not only as an optimization.

Still, analyzing Table 6.5 we see that the impact on performance is also pretty significant. This version of Pastel comes out on top, or very nearly on top, in almost every tested parameter. Reliability is greatly improved. The ratio of messages delivered in one hop is still very good, which in turn brings the average number of hops down. But most importantly, lookup latency is at its minimum, with only a 60 ms median lookup round-trip – a 4 times improvement over structured routing here.

This shows that increasing replication, and using anycast, is definitely the best way to improve performance on a full information system like Pastel. Ob-
<table>
<thead>
<tr>
<th>Pastel</th>
<th>send</th>
<th>route</th>
</tr>
</thead>
<tbody>
<tr>
<td>delivery rates (%)</td>
<td>99.3</td>
<td>94.8</td>
</tr>
<tr>
<td>delivered in one hop (%)</td>
<td>96.2</td>
<td>5.56</td>
</tr>
<tr>
<td>average hops (1)</td>
<td>1.07</td>
<td>3.14</td>
</tr>
<tr>
<td>median round-trips (s)</td>
<td>0.06</td>
<td>0.26</td>
</tr>
</tbody>
</table>

Table 6.5: Lookups in Pastel with $k = 5$.

Previously, there are diminishing returns, and you can only improve so much by increasing replication. But overall, results so far are definitely encouraging.
Chapter 7

Future Work

In this chapter we discuss as of yet unexplored, future work vectors for Pastel. This includes mostly some more complex optimizations, and peer-to-peer application development and testing.

The implementation of Pastry produced is incomplete, in the sense that it left out a few of the most complex optimizations to this protocol, including the self-adaptation features of [18], and the automatic-discovery algorithm of [23].

The first of those features, should work to improve the reliability of the structured overlay, and at the same time reduce the bandwidth required to maintain that same overlay. This is bound to improve our implementations of both Pastry and Pastel, but should not have a significant impact in the results presented here. That is to say, this feature is likely to benefit both equally, thus maintaining the comparative results between them.

The second feature on the other hand, may reveal itself important to the structured overlay’s locality properties, which should improve that overlay’s performance. This delivers benefits for both overlays (after all, using a structured overlay with good locality properties to build a full information overlay, was one of the premises of this work), but it is likely that it would put structured lookups noticeably closer to direct lookups in terms of latencies, which might be perceived as reducing the advantages of Pastel over Pastry.

On a more positive note, the implementation of Pastel, also lends itself to further optimization. For instance, a node that knowingly exits the system in a transient way, could persist its full information table. Then, on return such a node could negotiate a way of updating its full information table to its and the system’s benefit. A possible implementation would be the use of Merkle trees
as a way of downloading only the change set [32]. Another possible optimization would be for a node that knowingly departs the network in a permanent, and orderly fashion to immediately broadcast its part message, instead of leaving that task to its neighbors. Both these optimizations should work to reduce the constant factor behind the linear maintenance traffic components of Pastel, which could make even larger networks manageable.

Pastel could also perhaps benefit from self-adaptation features similar to the ones mentioned for Pastry, to attempt to reduce message duplication on direct lookups. This could further reduce bandwidth, and put less of a strain on applications.

Also perhaps worthy of investigation is the reason behind node departures consuming more bandwidth than node arrivals in all test cases.

The development and testing of peer-to-peer applications on top of Pastel’s and Pastry’s now unified interface is also important, because it would allow us to measure how much the synthetic performance achievements demonstrated here affect real-world applications. Applications like Past [8] and Scribe [11] are obvious candidates, but complex applications with mixed requirements should benefit from Pastel the most.

Finally, another investigation vector would be the addition of similar full information systems to other structured overlays. In particular, seen as how we have reduced Pastry’s $b$ parameter from its typical value of $b = 4$ to as low as $b = 2$, Chord is a likely candidate, as it is quite similar to a Pastry system with $b = 1$. We feel that Pastry will still have the edge thanks to its locality properties, but it is an interesting possibility nonetheless.
Chapter 8

Conclusions

This work describes Pastel, a truly generic peer-to-peer overlay that can be used (even simultaneously) by many disparate applications. Pastel extends Pastry with full membership information, in order to create a substrate that can support both fast, few-hop lookups, for direct contact applications, and slower multi-hop lookups, for applications that intend to exploit overlay topology.

Pastel’s design demonstrates that synergies exist between the maintenance of full membership information, and the structured routing state of overlays like Pastry. By exploiting them, bandwidth and storage costs can be kept low, even for reasonably sized and dynamic systems.

We implemented Pastel in an asynchronous and event-driven fashion, an implementation that can easily be used to quickly develop and test future peer-to-peer applications. Our own test results show that Pastel has lookups that are more efficient than Pastry’s, in both the number of steps required to contact a given node, and the latency of that operation. Furthermore the bandwidth required by Pastel is modest, even for highly dynamic systems with several hundred nodes. This demonstrates the value of the novel techniques that we employ to reduce bandwidth utilization on full information systems.

We believe Pastel can be further improved and its bandwidth reduced, if a few more complex optimizations were to be implemented. Larger scale tests should be performed, though a suitable testbed is hard to come by. Encouraged by present results, as well as previous simulation and analysis, we still expect Pastel to do reasonably well, even for systems with a few thousand nodes.
Bibliography


