An Epidemic Protocol for Managing Routing Tables in very large Peer-to-Peer Networks

Spyros Voulgaris, Maarten van Steen

Vrije Universiteit

Abstract

Last years' internet growth raised the demand for highly scalable, self-organizing, non-centralized systems. In particular, building self-maintained overlay networks for message routing has recently attracted significant research interest [7–11]. All suggested solutions have a common goal: To build and maintain structures (routing tables) that can be used to route messages. Several of the proposed algorithms focus on efficiency of bandwidth usage. However, their behavior is uncertain in the presence of highly dynamic environments, or serious disasters (i.e. half of the nodes crashing). In this paper we present an alternative approach to managing routing tables for peer-to-peer routing overlay networks, based on the Newscast epidemic protocol [1]. We substantiate our claims by presenting experimental results. We, therefore, demonstrate the potential of the Newscast epidemic protocol to create highly robust, self-administered overlay networks, able to sustain and adapt fast to severe network changes.

1 Introduction

The Internet has dramatically expanded over the past few years, proving the traditional client-server model of communication inadequate for a number of services in the large scale. The network research community has realized that using centralized servers is not the way to go with respect to managing and administering very large scale distributed systems, as well as for certain applications for such systems. As a result, considerable effort has been made in designing *peer-to-peer* (P2P) overlay networks. These networks are highly (or totally) decentralized distributed systems, where nodes are equal peers cooperating to provide a service all together. The major advantage of such systems is that they do not involve any central point of administration or control.

A significant part of the recent research in P2P systems has been in designing overlay networks for routing. These networks operate in the application layer, on top of an existing physically interconnected set of nodes (such as the Internet). They assign each participating node an ID, and route messages to a node based on that, rather than based on its IP address. Performance (in terms of routing hops) is usually inferior compared to traditional IP routing. However, they offer a number of other, attractive advantages, such as higher fault tolerance, flexibility of deployment, adaptivity, as well as lack of central control. A number of such P2P systems has been proposed, such as CAN [7], Chord [8], Pastry [9], and Tapestry [10]. Their common property is that they all try to form and maintain some sort of structure across the large number of participating nodes, that is then used to route packets among them.

Other P2P algorithms (such as Newscast [1]) fall in the category of *epidemic* (or *gossip*) protocols. They aim at exploiting randomness to disseminate information across a large set of nodes to keep that set of nodes highly connected even in the event of major disasters, without keeping any static structures or requiring any sort of administration. Connection between nodes in such systems is highly dynamic. These systems are more adaptive to major network changes, and appear to have a self-healing behavior with respect to major network disasters. Their lack of structure, however, restricts them from carrying out some types of services (i.e. routing) in an efficient way.

In this paper we combine the advantages of routing overlay networks with those of highly fault tolerant, self-healing epidemic networks. In particular, we investigate how to bootstrap and maintain structures used for peer-to-peer routing based on the highly dynamic emergent behavior of Newscast. Moreover, this paper demonstrates the power of an epidemic protocol as simple as the Newscast protocol, in managing structures

across very large-scale distributed systems, in a totally distributed and scalable way, with no need for external administration, and with very high fault tolerance.

Section 2 provides a brief description of Newscast, concentrating more on its epidemic protocol. Section 3 describes structures that can be used for peer-to-peer routing. The architecture proposed for management of peer-to-peer routing tables is presented in section 4. Section 5 describes the experiments we conducted, and section 6 discusses the results obtained. Finally, section 7 presents conclusions and directions for future research.

2 The Newscast Protocol

Newscast (introduced in [1]) is a model for information dissemination and membership management in large-scale, agent-based distributed systems. It deploys a simple, peer-to-peer data exchange protocol. The *Newscast protocol* forms an overlay network and keeps it connected by means of an epidemic algorithm. The Newscast protocol is extremely simple: each agent knows only a (continuously changing) small set of peers, and periodically picks randomly one of them to exchange information with.

In this section we present a brief overview of the Newscast protocol's operation, and subsequently explore some properties of its emergent behavior.

2.1 Principal Operation

In Newscast information is exchanged by means of *news items*. A news item is a 4-field structure containing (a) the ID of the agent where it originated, (b) the network address of that agent, (c) a timestamp of the moment it was generated, and (d) some application-specific data. Each agent maintains a *fixed-sized* cache of c news items (with typical value 20 to 40). The basic idea is that each agent periodically picks a random peer from its cache and subsequently both agents replace their cache entries with the c freshest news items of the union of their original caches.

More formally, but omitting specific details described in [1], each agent executes the following four steps once every ΔT time units (ΔT is referred to as the *refresh interval*):

1. Add a fresh (agent-specific) news item to the cache.
2. Randomly select a peer agent by considering the network addresses of other agents as found in the cache.
3. Send all cache entries to the selected peer agent, and, in turn, receive all that peer's cache entries.
4. Out of the (up to) $2c$ cache entries, keep the c newest ones, and discard the rest.

The selected peer from step 2 executes the last two steps as well, so that after the exchange both agents have the same cache. Note, however, that as soon as any of these two agents executes the protocol again, their respective caches will most likely be different again.

This algorithm resembles the traditional push-pull epidemic protocol [4]. A critical difference, however, is that no correspondent knows the complete member list, but only a small, random fraction of it.

The protocol does not require that the clocks of the agents are synchronized, but only that the timestamps of news items in a single cache are mutually consistent. We assume that the communication time between two agents is negligible compared to ΔT (which is generally in the order of minutes). When an agent A passes its cache to B, it also sends along its current local time, T_A. When B receives the cache entries, it subsequently adjusts the timestamp of each entry with a value $T_A - T_B$, effectively normalizing the time of each new entry to those already cached.

2.2 Properties of Newscasting

As it turns out, this simple model of communication has desirable statistical properties. To understand the behavior of newscasting, we consider the undirected communication graphs G_t at different time instants t. Each such graph is constructed as follows. The vertex set V_t of G_t contains the agents that are alive at time t. A link between agents $a, b \in V_t$ exists if and only if either a is in the cache of b or b is in the cache of a

at that time. The cache-exchange algorithm leads to a series of graphs G_t, given an initial graph G_0. Graph G_t expresses the possibility of cache exchanges, and in essence information flow, at time t.

Now consider the series of graphs $G_0, G_{\Delta T}, G_{2\Delta T}, \ldots$. Note that during a time interval ΔT each agent initiates the cache-exchange algorithm. In other words, after ΔT time units, all agents will have added a fresh news item to their caches, and will have exchanged and merged caches with at least one of their neighbors (and possibly more). We say that a *cycle* of the Newscast protocol has completed.

We have conducted simulations with up to 50,000 agents [1] assuming an idealized communication infrastructure with no communication delays and packet losses, and emulations by deploying up to 128,000 actual Newscast agents on a real wide-area network [6]. Both our simulations and emulations show that even for small cache sizes (say, $c = 20$), each graph $G_{k\Delta T}$ stays connected. Moreover, it turns out that the *average path length* (average length of shortest paths between any two nodes) converges to a very low value in just a few cycles, and which is only slightly longer than the average path length in random graphs. For real experiments with 128,000 nodes, and cache size of c=20, 30, and 40 entries, the average path length converges to 6, 5, and 4, respectively within the first 30 cycles.

A more significant property of Newscast is, however, its strong connectivity. Let G_t' be a subgraph of G_t, where a number of random nodes (and their links) have been removed. Our simulations and emulations show that G_t' remains connected even when more than half of the nodes are removed. This means that when even half of the agents of a Newscast network are removed, the rest of the nodes remain connected in a single cluster. In fact, Newscast's connectivity property is so strong that one needs to remove over 75% of the nodes to start breaking up the remaining network into disjoint clusters. The nodes surviving such a major disaster, quickly converge to an independent strongly connected Newscast network, capable of sustaining further major disasters of similar severity.

Our experiments also show that we need only an extremely simple way of handling membership, which is an important improvement in comparison to other epidemic models, such as [5]. Consider the worst solution to handling membership that could possibly disrupt the emergent behavior of our protocol: an agent contacts the agent running on a well-known central server and simply initiates the cache-exchange protocol with it. This approach systematically biases the content of caches, which now all depend on what is stored at the central server.

We conducted a simulation experiment in which we admitted 50 new agents at every communication cycle until 5,000 agents had joined the network, after which no new agents were allowed to join. By measuring the average path length again, we saw that shortly (i.e. approximately 15 cycles) after the last agents had been added, the average path length quickly converged to the one we would expect in a stable graph. We can conclude that even this worst-imaginable membership protocol does not affect the general properties of newscasting. In effect, when a node wants to join, it needs to know only the address of a single other node and can simply start executing the newscast protocol. Leaving is done by simply stopping communication.

Further investigations revealed that the induced communication graphs have many properties in common with what are known as *small worlds* [2]. An important property of these types of networks is that they show a relatively high *clustering coefficient*, which, for a given node, is the ratio of the number of *existing* edges between the node's neighbors and the number of *all* possible edges between them. For example, in a complete graph all nodes have a clustering coefficient of 1 while in random graphs this coefficient is typically small (if our graphs were random, the clustering coefficient would be expected to be c/n).

Below we shall show how we can use these properties of the Newscast protocol to maintain structure for routing in large-scale peer-to-peer networks.

3 Peer-to-Peer Routing

One of the key issues in designing large-scale peer-to-peer overlay networks is to provide an efficient way to do routing. Several architectures have been proposed as peer-to-peer routing substrates, such as CAN [7], Chord [8], Pastry [9], and Tapestry [10]. Such distributed systems that map "keys" onto "values" in a way similar to hash tables, are referred to as *distributed hash table* (*DHT*) based networks [11]. Two of the most popular of them, Pastry and Tapestry, employ routing based on the same concept: incrementally matching the destination's ID, digit by digit. In this section we present the structure and operation of the principal structures used for routing, the *routing tables*.

3.1 Basic concept

Each node is assigned a unique numeric identifier, its *nodeId*, or simply *ID*. When presented with a message and a numeric *key*, a node routes the message towards the node whose ID is equal to the given key. NodeIds and keys are N-bit integers, forming a nodeId space that spans from 0 to $2^N - 1$. *N* has a typical value of at least 64 to provide a sufficiently large nodeId space to accommodate possibly billions of nodes. Nodes pick their nodeIds randomly with uniform probability from the set of N-bit strings. It is, therefore, assumed that nodeIds are uniformly distributed across all geographic regions, multiple jurisdictions, and various networks.

3.2 Internal structure of the Routing Tables

For the purpose of routing, nodeIds and keys can be thought of as a sequence of digits in base 2^b (*b*-bit long digits), where *b* is a configuration parameter with typical value 4 (which implies *hexadecimal* digits). Routing a message to its destination is achieved gradually, by matching one additional digit of the message's key at a time, say, from left to right. That is, in each step the message is normally forwarded to a node whose ID shares with the key a prefix at least one digit (*b* bits) longer than the prefix the key shares with the present node's ID, if such a node is known. If such a node is not known, routing of that message fails. [1]

To implement the logic described above in message routing, each node maintains its *routing table*. The routing table of a node consists of N/b rows of 2^b entries each. An entry contains the ID of a node, and its corresponding IP address. A given row of the routing table contains 2^b entries, and represents a matching prefix in the nodeId up to a digit position. Entries in the *r*th row ($r \in \{1, \ldots, N/b\}$) contain nodes whose IDs share the same $(r-1)$-digit prefix with the present node. The *c*th entry of the *r*th row contains such a node, with the additional constraint that its ID's *r*th digit is equal to *c*. For instance, assuming b=4 (hexadecimal digits for the nodeId), the 2nd entry of the 3rd row of the routing table for node 437BF52...($N/4$ hex digits in total) is some node whose ID starts with 43**2**, while the 8th entry of its 5th row has a node whose ID starts with 437B**8**.

3.3 Routing

Upon receiving a message, a node compares the message's key to its nodeId. If they share a common prefix of *i* digits, it should forward it to a node whose ID shares a prefix of $i+1$ digits with the key. To accomplish that, the present node looks up the $(i+1)$th row of its routing table, which contains nodes sharing with the key the same *i* first digits. Out of that row, it picks the *k*th entry, where *k* is the value of the key's $(i+1)$th digit, and forwards the message to that node. That node not only shares with the key the same first *i* digits, but also the $(i+1)$th one. This process continues either until the node whose ID matches all digits of the message's key is reached, or, else, until the message cannot be forwarded any further.

4 P2P Routing based on Newscast

An important issue in DHT-based peer-to-peer systems is managing the routing tables. These tables are kept up-to-date by having nodes that join or leave the system contact other nodes explicitly. To handle failures, heartbeat algorithms are used to probe nodes and to take measures when a failure is detected.

We propose a different approach, namely to separate routing from table management, similar to the separation deployed in Internet routing protocols such as OSPF or RIP. We believe such a separation often leads to a cleaner and simpler design, although sometimes at the cost of performance.

Newscast can typically be used as a distributed background process by which nodes are kept up-to-date in a lazy fashion. For DHT-based peer-to-peer systems, we propose to deploy Newscast for maintaining routing tables. Our method is completely decentralized, highly robust, and quickly adjusts itself to major changes in the network. These advantages come at the price of continuous bandwidth consumption.

---

[1] If such a node is not known, existing systems consult auxiliary routing structures that they maintain (such as the *leaf set* in Pastry), which are out of the scope of this paper, as we are interested only in the efficiency of Newscast to maintain the main routing tables.

4.1 The Principal Idea

Newscast's epidemic protocol has a number of important properties, as described in section 2. It maintains a strongly connected graph, it sustains disasters, it adapts very fast to (possibly major) network changes, and it is highly scalable. The idea is to combine the adaptivity strength of the Newscast epidemic protocol with the efficiency of the routing scheme presented in section 3, to create a robust, highly fault resilient, peer-to-peer overlay network for efficient routing.

Knowledge of peer nodes provided by Newscast can be used to populate the routing tables. In each iteration of the Newscast protocol every node receives references to c other nodes, randomly chosen among *all* the participating nodes. Each node has to gather enough information to build and maintain all its N/b rows.

Let us concentrate first on building the first row of a node's routing table. This requires references to nodes whose IDs differ from the present node's ID in the first digit, which makes a total of $2^b - 1$ nodes. Seen differently, considering 2^b *classes* of nodes split according to their ID's first digit, we require a reference to an arbitrary representative from each class (excluding the present node's class). Assuming evenly distributed node IDs, each class contains roughly $1/2^b$ of the nodes. Therefore, with very high probability, a node will have learned about at least one representative from each of the $2^b - 1$ classes when 2^b (or a few more) random nodes become known to it. Assuming $2^b = 16$ (for $b = 4$) and cache size $c = 20$, this might happen even when a node executes the Newscast cache-exchange protocol only once.

For the second row of a node's routing table, we require references to $2^b - 1$ nodes of IDs with the same first digit, but different second digit than the present node's ID. Apparently, we are seeking for representatives of much narrower node ID ranges, each containing roughly $(1/2^b)^2$ of the total nodes. In general, filling up the kth row requires representatives of $2^b - 1$ classes, each containing just $(1/2^b)^k$ of the total nodes. The narrower a node ID range gets, the more difficult it becomes to find a representative from it by taking random samples across all the nodes. Obviously it would be inefficient to rely on Newscast over the whole set of nodes to find representatives of these narrow node ID ranges. A more focused approach is required, described in the following subsection.

4.2 Multi-layer Newscast scheme

As we mentioned, executing the Newscast protocol can be seen as running a distributed background process by which nodes are kept up-to-date. To efficiently maintain routing tables, we run multiple instances of the Newscast protocol, each node running several Newscast agents in parallel. In fact, each node runs exactly N/b Newscast agents, each one being responsible for maintaining one of the rows of the node's routing table. The agent responsible for row $r \in \{1, \ldots, N/b\}$ of node X will be referred to as *agent #r of node X*.

Note that a node's agent #i deals only with nodes whose IDs share a common prefix of length $i - 1$ with the present node's ID, that is, it does not accept any nodes with a different prefix in its cache. Moreover, a node's agent #i interacts, in terms of the Newscast protocol, only with agent #i of peer nodes, as shown in figure 1. Apparently, agent #i of a peer node contacted by agent #i of the present node, contains items whose node IDs share the same $i - 1$ long prefix too. What we are thus seeing, is that agent #i of node X maintains a small-world network with *some* other nodes whose IDs are the same in the first $i - 1$ digits as the ID of X. Agent #1 of all nodes maintain a single connected small world of the whole set of nodes.

To collect *all* nodes with a particular prefix in a *single* connected small world, we apply the following strategy. Peers that become known to a node's agent #i are *also* reported to the same node's agent #($i+1$). That agent, in turn, inserts the peers that match its prefix requirement in its cache (by replacing the oldest cache items), and further reports them to agent #($i+2$) of the same node, if any. In other words, any peer that becomes known to a node's agent #i, is also made known to all the agents #j ($j > i$) of the same node that are potentially interested.

An important observation is that once agents #i of *all* nodes that share the same first $i - 1$ ID digits have formed a single small world, agents #($i+1$) of the nodes among them that also share an arbitrary same ith digit form a single connected small world very fast. Each agent #i learns about c random peers with the same first $i - 1$ digits every ΔT time units. Assuming evenly distributed node IDs, we expect that on average $c/2^b$ of the peers that become known every ΔT time units share the ith ID digit too with the present node, in addition to the first $i - 1$ digits. Given typical parameter values of $c = 20$ and $b = 2$, one or more
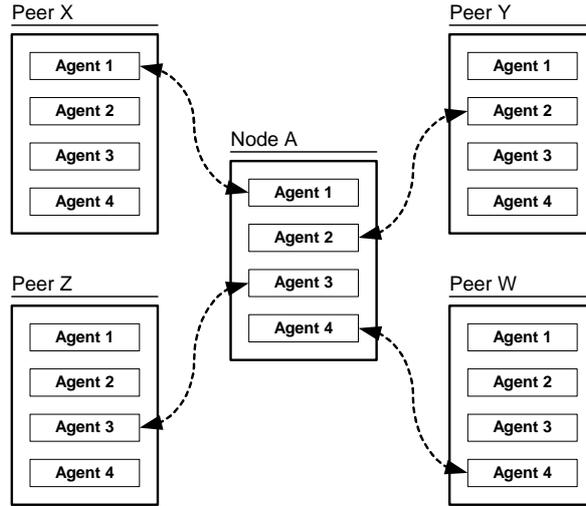
Fig. 1. Communication of node *A* during one communication cycle.

peers sharing i digits become known every ΔT time units on average. This partly explains why all agents of every node form small worlds quickly, as we shall see later.

Notice that, initially, every node's agent #$(i+1)$ forms its own (trivial) small world, disjoint from all the rest. Such a small world generally expands on each cycle of agent #i, since a random peer satisfying the prefix requirement of agent #$(i+1)$ is introduced. Moreover, two small worlds of n and m nodes unite if any of the n nodes of the first happens to learn about the existence of any of the m nodes of the other, respectively. Therefore, the larger disjoint small worlds become, the more likely they will unite. What we are seeing, is an increasingly accelerating behavior in the process of merging among disjoint small worlds. It is therefore reasonable to state that agent #$(i+1)$ of a set of nodes sharing the same first #i digits form a small world in just a few cycles, provided agent #i of nodes sharing the same first #$(i-1)$ digits form a small world too.

The set of all nodes' agent #1 run a pure Newscast instance that guarantees a (quickly formed) single connected small world of *all* existing nodes. By induction, and based on the claims of the previous paragraph, we expect all instances of Newscast executed by all agents of all nodes, to quickly form the small worlds they are designed for.

Our claims are further strengthened by the presentation of emulation analysis in section 6.

5   Experimental Setting

We implemented the architecture described in section 4.2 in Java and deployed it on the DAS-2 [3], a 400-processor cluster geographically distributed over a wide-area network across the Netherlands. We carried out experiments with a set of 65,536 nodes, a number of them running on each DAS-2 processor simultaneously.

Regarding the parameters related to peer-to-peer routing, we considered node IDs of length $N = 16$ bits, and digits of length $b = 4$ bits (hexadecimal digits). This setting resulted in $N/b = 4$ rows and $2^b = 16$ columns per routing table.

As far as the Newscast parameters are concerned, each node was running 4 Newscast agents, one for each of its 4 routing table rows. A cache size of $c = 20$ was used for each Newscast agent. We ran our experiments with the same refresh interval of $\Delta T = 10sec$ for all agents. That is, every 10 seconds *each* of the 4 Newscast agents of each node initiated a cache exchange. We recorded and analyzed the behavior of our architecture at intervals of 60 seconds, that is, we logged the whole network's state every 6 communication cycles.

Another facet of our experiments that is worth noting is the *bootstrapping* mechanism. By bootstrapping we refer to the procedure of providing agents with the information required to jump-start the overlay network's formation. In principle, a new agent joins by contacting *any* existing agent and exchanging caches. When the whole network starts from scratch, a systematic way has to be present to provide one or more initial communication points to each agent. In our experiments, all nodes' agents #1 were provided with the address of one single-selected node's agent #1. Providing agents with a choice of (possibly random) agents to connect to initially, enhances the randomness of the network from the early cycles. However, a bootstrapping mechanism as simple and centralized as the one we chose further endorses our claims of our architecture's fast convergent behavior, as discussed in the following section.

Finally, we imposed a fake large-scale failure while the experiment was running, to observe and analyze the behavior of our system in such cases. In particular, we killed 50% of the nodes in the middle of the experiment. Our observations of the experiments and their analysis are presented in the following section.

6 Experimental Results and Analysis

This section presents the output of our experiments with 65,536 agents. We recorded and analyzed two aspects of the system's behavior: dynamic forming of the routing tables when bootstrapping, and following a large-scale failure.

6.1 Bootstrapping

The first part of our experiment aimed at observing the system's behavior while bootstrapping. Figure 2 presents the system's fast convergence to a fully operative routing substrate. It shows the average number of routing table rows that are completely filled per node, as a function of the number of cycles elapsed from the experiment's start. A node's ith routing table row being completely filled means that the node can route *any* message whose key shares $i-1$ digits with the node's ID to a peer node whose ID additionally matches the ith digit of the message's key. Note that the system manages to fill *all* routing table entries in *all* nodes in less than 30 cycles.
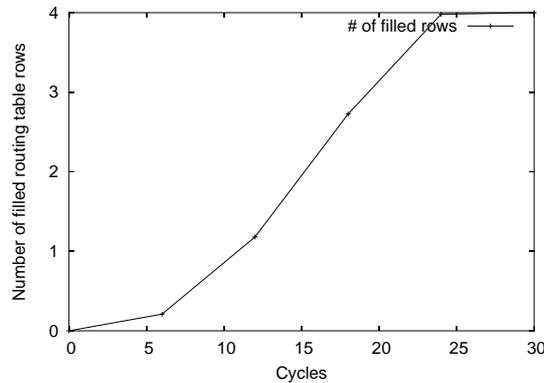

Fig. 2. Average number of filled routing table rows.

Figure 3 demonstrates the efficiency in routing messages to random destinations. From each node we routed a number of messages to random nodes. Figure 3 presents the average values. The left-hand diagram shows how many routing steps messages took on average en route to their destination. Initially, routing tables are empty, so messages cannot take any steps towards their destinations. However, as routing tables are gradually formed, messages are correspondingly routed through more steps. This diagram is similar to the one of figure 2, as the number of routing steps a message takes is directly dependent on the number of filled routing table rows.

The right-hand diagram of figure 3 shows what percentage of the messages manage to actually reach their destinations, as a function of the number of cycles. For the first 10 cycles few or none of the messages reach their destinations. However, as routing tables are filled, more messages are routed all the way through to their destinations. As it turns out, after the first 24 cycles 99.74% of the messages are delivered to their destinations, and after 30 cycles, this fraction increased to 99.998%.
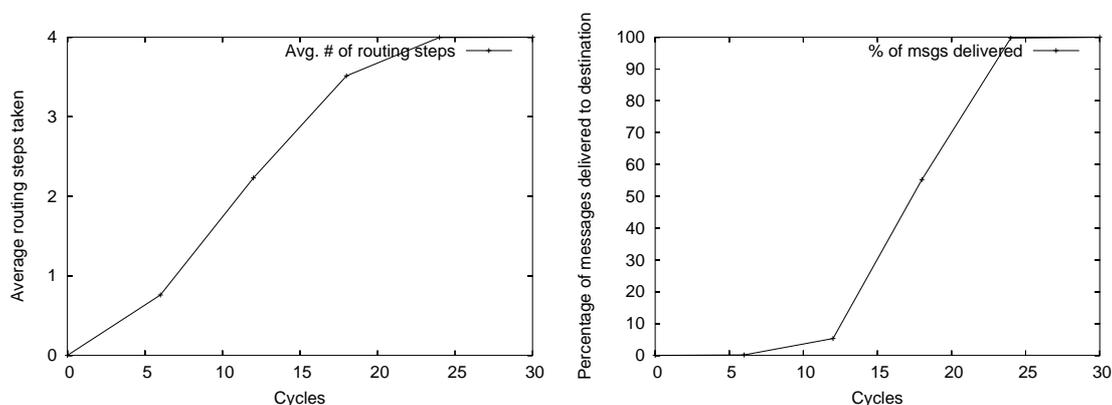

Fig. 3. Left: Average routing steps taken. Right: Percentage of messages delivered.

6.2 Robustness to large-scale failures

To test the system's behavior in the face of large-scale failures, we intentionally killed *half* of the agents after we knew that all nodes' routing tables had been completely filled, at cycle d.[2] More specifically, we killed all nodes with an odd ID. As we shall see, the network remains connected after such a major disaster, and adapts very quickly to the set of nodes that remain alive.
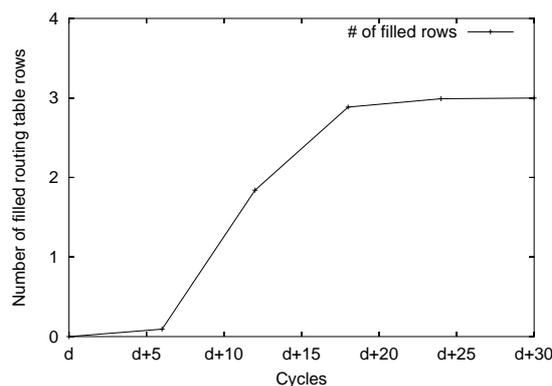

Fig. 4. Average number of filled routing table rows when recovering from a 50% node crash that happened at cycle d.

Figures 4 and 5 are analogous to the previous figures, 2 and 3. Figure 4 shows the average number of routing table rows that are completely filled (with valid entries), per node. Note that outdated entries of crashed nodes (the ones with odd IDs) are not considered valid, and therefore are not counted. Immediately

---

[2] This corresponded to approximately 10 minutes after the experiment's start

after the crash none of the nodes' rows are filled, which implies that all nodes' routing rows also had some entries with odd node IDs. However, as can be seen in the diagram, routing tables are filled very quickly. Within 30 cycles from the crash all nodes' first 3 routing table rows have been filled. Note that this is the maximum number of rows that can be filled per node. Routing tables' 4th rows cannot be filled, as they would require nodes that match all possible cases for the last digit of their IDs. Since nodes with odd IDs do not exist any more, it is not possible to fill up these rows. This, however, does not affect routing, as routing paths to all existing nodes (i.e. nodes with an even ID) do exist and are complete.
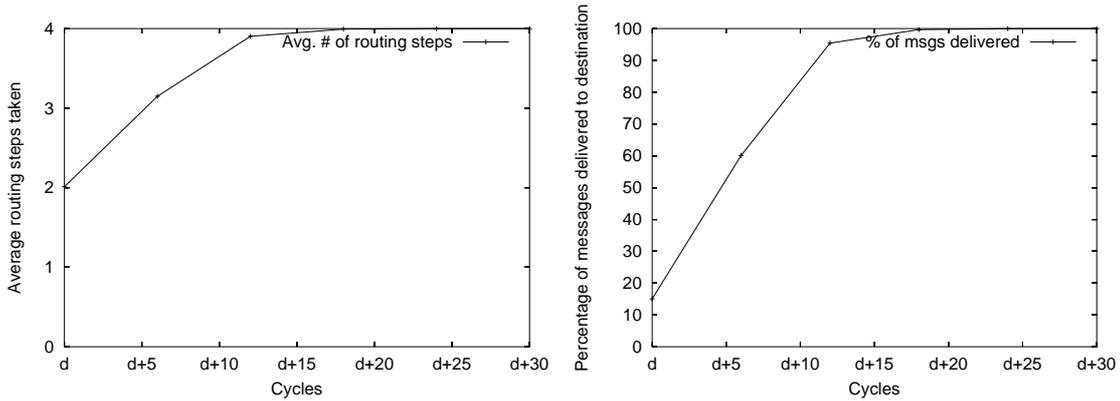

Fig. 5. Message routing while recovering from a 50% node crash that happened at cycle d. Left: Average routing steps taken. Right: Percentage of messages delivered.

The system's capability to route messages can be seen in figure 5. The left-hand diagram shows the average number of steps a message is routed through. Initially, since half of the nodes have been removed, messages are routed on average half-way through to their destination. As routing tables adjust to the change imposed by half the nodes crashing, messages are routed through more steps to their destinations. The right-hand diagram of figure 5 shows the percentage of messages that are successfully routed all the way through to their destination. Just like in the bootstrapping case, routing tables are formed very fast. It takes less than 20 cycles from the moment of the crash to form routing tables that can route any message from *any* source to *any* destination.

6.3 Bandwidth considerations

In this section we provide an estimation of the individual (per node) and aggregate bandwidth used in our experiments. Despite the 16-bit node IDs we used in our experiments, we make the estimation assuming node IDs of 64 bits, which would be the ID size in real operation.

A cache entry consists of 16 bytes: 8 bytes for the node's 16-bit ID, 4 bytes for its ip address, 2 bytes for the port, and 2 bytes for the entry's timestamp. One cache has $c = 20$ entries, which account for 320 bytes. A cache exchange involves sending the cache to a peer *and* receiving the peer's cache, therefore causes traffic of 640 bytes. Every ΔT, each node initiates exactly one cache exchange, and also participates on average in one cache exchange initiated elsewhere. Therefore, two cache exchanges cause transfer of 1280 bytes. For running 4 agents, a single node exchanges $4 \times 1280 = 5120$ bytes every $\Delta T = 10 sec$. That is, 512 bytes per second, or 4096bps (4Kbps). This is the price to pay for achieving fully operative routing tables in less than $30 \times 10 = 300$ seconds, which is 5 minutes.

For the aggregate bandwidth we multiply the individual node bandwidth by the number of nodes and divide by two, since the traffic caused by each cache exchange has been counted twice, once for the exchange initiator and once for the peer node. Therefore, we have a total bandwidth of $65,536 \times 4/2 = 131,072$Kbps, which is 128Mbps. Note that even though this bandwidth seems too high, it is in fact distributed across the whole (possibly world-wide) network.

In a real system, with 64-bit node IDs, and a digit length of 4 bits, we would need 16 Newscast agents running per node. This would require the exchange of $16 \times 1280 = 20,480$ bytes every ΔT per node. Note that the refresh interval, ΔT, is a configuration parameter. By setting a longer refresh interval, we can lower the bandwidth used by each node, at the expense of slower completion of the routing tables. For instance, a refresh interval of $\Delta T = 60sec$ would require a bandwidth of $20,480/60 \simeq 341$ bytes per second, or roughly 2.7Kbps. However, in that case routing tables would take longer to be filled, around 30 minutes.

7   Conclusions and Future Directions

This paper aimed at demonstrating the potential of the Newscast protocol in building large-scale, self-managing communities. In particular, we dealt with the application of managing routing tables for DHT-based peer-to-peer networks. We introduced a Newscast-based architecture for this application, and analyzed the system's behavior through experimentation. We showed that the proposed system forms routing tables fast, in a totally decentralized, self-organized manner.

This research is very recent, and currently under development. The results of the experiments suggest that our system can provide highly robust, non-centralized routing table management. However, more research remains to be done to discover potential optimizations for our architecture, such as in the field of bandwidth consumption. Our architecture could possibly use significantly less bandwidth if adaptive refresh intervals were applied. Also, each agent of a node could have an individually optimized set of configuration parameters, such as cache size and refresh interval. Future research aims at optimizing the current approach.

Another goal for future research, in a broader sense, is exploiting Newscast for a multitude of diverse peer-to-peer applications. We envision Newscast as being a basic background process, supporting, organizing, and managing overlay networks in a fully decentralized way.

The contribution of this paper is that it provides and analyzes a complete solution to a specific problem, showing the potential of the Newscast protocol to support such systems.

References

1. M. Jelasity, M. van Steen. Large-scale newscast computing on the Internet. Technical Report IR-503, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Oct. 2002.
2. R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, Jan. 2002.
3. DAS-2. http://www.cs.vu.nl/das2/.
4. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database management. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, Aug. 1987. ACM.
5. A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 2003. To appear.
6. S. Voulgaris, M. Jelasity, M. van Steen. A Robust and Scalable Peer-to-Peer Gossiping Protocol. Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Apr. 2003. *Submitted for publication* (http://acropolis.cs.vu.nl/newscast-experiments.pdf).
7. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, California, USA, August 2001.
8. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, California, USA, August 2001.
9. Antony Rowstron, Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.
10. Ben Y. Zhao, John D. Kubiatowicz, Anthony D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U.C. Berkeley, California, USA, April 2001.
11. Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica Looking up data in P2P systems In *Communications of the ACM*, vol. 46, February 2003, pp. 43-48