XNET: A Reliable Content-Based Publish/Subscribe System

Raphaël Chand, Pascal Felber
Institut EURECOM, Sophia Antipolis, France
{*raphael.chand, pascal.felber*}*@eurecom.fr*

Abstract

Content-based publish/subscribe systems are usually implemented as a network of brokers that collaboratively route messages from information providers to consumers. A major challenge of such middleware infrastructures is their reliability and their ability to cope with failures in the system. In this paper, we present the architecture of the XNET *XML content network and we detail the mechanisms that we implemented to gracefully handle failures and maintain the system state consistent with the consumer population at all times. In particular, we propose several approaches to fault tolerance so that our system can recover from various types of router and link failures. We analyze the efficiency of our techniques in a large scale experimental deployment on the PlanetLab testbed. We show that* XNET *does not only offer good performance and scalability with large consumer populations under normal operation, but can also quickly recover from system failures.*

1. Introduction

Content-based routing differs significantly from traditional communication, in that messages are routed on the basis of their content rather than the IP address of their destination. This form of addressing is widely used in event notification or publish/subscribe systems [10] to deliver relevant data to the consumers, according to the interests they have expressed. By allowing consumers to define the type of messages they are interested in, producers do not need to keep track of the consumer population and can simply inject messages in the network. In turn, consumers with scarce resources (e.g., mobile devices) can restrict the amount of data that they receive by registering highly-selective subscriptions, and hence limit their incoming network traffic. The complex task of filtering and routing messages is left to the network infrastructure, which typically consists of application-level routers organized in an overlay network.

Our XNET XML content routing network integrates several novel technologies to implement *efficient* and *reliable* distribution of structured XML content to very large populations of consumers. The routing protocol, XROUTE [6], makes extensive use of subscription aggregation to limit the size of routing tables while ensuring perfect routing (i.e., minimizing inter-router traffic). The filtering engine, XTRIE [5], uses a sophisticated algorithm to match incoming XML documents against large populations of tree-structured subscriptions, while the XSEARCH subscription management algorithm [7] enables the system to efficiently manage large and highly dynamic consumer populations.

In this paper, we specifically address the issue of *reliability* in our XML content network. We propose several schemes that are based on different strategies and approaches to fault tolerance. Their common goal is to ensure that the shared state of the system (i.e., all registered subscriptions) is consistent with the actual consumer population at all times. The network can recover from router or link failures by using the most appropriate scheme depending on various factors, such as the expected duration of the outage or application-specific availability requirements. We have performed an extensive performance evaluation of our system by deploying it on the nodes of a real Internet-wide network, with realistic content and subscription workloads. We have evaluated the overall performance of XNET as well as its ability to cope with system failures.

The rest of this paper is organized as follows: We first discuss related work in Section 2, and we give an overview of our XNET system in Section 3. In Section 4, we focus on the mechanisms that we implemented in our system to achieve reliability. Section 5 presents results from experimental evaluation. Finally, Section 6 concludes the paper.

2. Related Work

Several publish/subscribe systems support content-based routing (see [10] for a survey), but few of them guarantee reliable delivery in the presence of link or server failures.

IBM Gryphon [1] uses a set of networked brokers to distribute events from publishers to consumers. It uses a distributed filtering algorithm based on parallel search trees maintained on each of the brokers to efficiently determine where to route the messages. The authors do not discuss how to update the parallel search trees (and thus ensure reliable delivery) in the case of link failures or router crashes.

Siena [2] also uses a network of event servers for content-based event distribution, and relies upon a routing protocol most similar to ours, but with limited support for subscription cancellation. In a recent paper [3], the authors of Siena introduce a novel routing scheme for content-based networking based on a combination of broadcast and selective routing. The system handles subscription cancellations by having routers periodically request the routing table of other routers. However, it does not guarantee perfect routing in the sense that consumers may receive messages that they are not interested in. Also, the authors do not explicitly address the issue of fault tolerance in the system.

In [16], the authors propose an approach for content-based routing of XML data in mesh-based overlay networks. They introduce a routing protocol that reassembles data streams sent over redundant paths to tolerate some node or link failures. Their approach provides a high level of availability but it is not clear how reliability is guaranteed during the addition and removal of subscriptions.

Rebeca [11, 12] is a prototype notification service that incorporates several routing strategies. Its topology is very similar to ours, i.e., a tree of brokers with a single root called the "root router." Rebeca also distinguishes between brokers that have local clients and those that do not. The system implements a self-stabilization algorithm based on subscription leasing. Routing table entries are valid as long as the lease of the corresponding subscription has not expired. This may lead to consumers receiving out-of-interest notifications. Also, this approach requires that consumers regularly renew their leases by resubscribing, making the system potentially unscalable to large consumer populations.

Jedi [9] relies upon a network of event servers organized in an arbitrary tree; subscriptions are propagated upward the tree, and messages are propagated both upward and downward to the children that have matching subscriptions. In a recent work [14], the authors discuss how to adapt the behavior of a publish/subscribe system to dynamic topology reconfiguration. Their work is based on the "strawman approach" [2] and aims at reducing overhead, notably by minimizing the repropagation of subscription information, while tolerating frequent reconfigurations.

3. System Overview

This section gives an overview of the XNET content routing network. We also briefly describe the most essential features of the routing protocol, which are relevant for the rest of the paper. More details can be found in [6].

System Model and Definitions. XNET, like most content-based publish/subscribe systems, is implemented as an overlay network of routing brokers. Messages (or events) are propagated through the nodes of the network, according to the messages' content and the subscriptions registered by the consumers. Each data consumer and producer is connected to some node at the edge of the network; we call such nodes *consumer* and *producer* nodes. The other nodes are called *routing* nodes. For the sake of simplicity, we consider a network with a single data source although our protocol supports multiple producers. The consumer population can be highly dynamic and does not need to be known a priori. Messages flow along a spanning tree, rooted at the producer node, whose leaves are the consumer nodes. For a given node, we denote by "upstream" and "downstream" the paths toward the producer and consumers, respectively.

Each routing broker has a set of *links*, or *interfaces*, that connect the node to its direct neighbors. We assume that there exists exactly one interface per neighbor (we ignore redundant links connecting two neighbors). Nodes communicate using reliable point-to-point communication and are equipped with failure detectors that eventually detect the failure of their communication links and neighbors but may make mistakes. As will become clear later, if a node incorrectly suspects its upstream neighbor to have failed, it might take unnecessary recovery actions that, although time consuming, do not adversely affect the consistency of the global state of the system. We assume a crash-recover model with transient link and router failures (although the duration of failures is unbounded).

XNET was designed to deal with XML data, the *de facto* interchange language on the Internet. Producers can define custom data types and generate arbitrary semi-structured events, as long as they are well-formed XML documents. Consumer interests are expressed using a subscription language. Subscriptions allow to specify predicates on the set of valid events for a given consumer. XNET uses a significant subset of the standard XPath language to specify complex subscriptions [17] adapted to the semi-structured nature of events.More details about the subscription management techniques of XNET can be found in [4, 5, 7].

We say that a subscription S_1 *covers* another subscription S_2, denoted by $S_1 \supseteq S_2$, if and only if any event matching S_2 also matches S_1, i.e., $\textit{matches}(S_2) \Rightarrow \textit{matches}(S_1)$. The covering relationship defines a partial order on the set of all subscriptions.

The Routing Protocol. The XROUTE content-based routing protocol has been designed to achieve several goals: to implement perfect routing, i.e., a message traverses a communication link only if there is some consumer downstream that is interested in that message; to be optimal in the sense that the link cost of routing an event is no more than that of sending the event along a multicast tree spanning all the consumers interested in the event; to maintain the size of the routing tables as small as possible by detecting and eliminating subscription redundancies; and to efficiently support dynamic registration and cancellation of consumer subscriptions.

Routing works in a distributed manner. Each node in the network contains in its routing table a set of entries that represent the distinct subscriptions that its neighbor nodes are interested in. For each subscription S, node N maintains some information in its routing table indicating to which neighbors it should forward an event matching S. The process starts when a publisher produces an event at its publisher node and ends when all the consumer nodes interested in that event have received it. Figure 1(a) shows the path that event e_1, published by P_1 and matching subscription S, will follow (subscriptions are represented underneath the consumers that registered them and routing table entries are listed next to the node they are associated with).

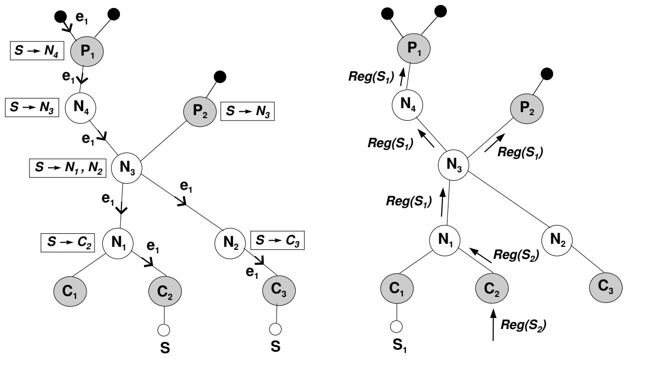

Figure 1. (a) Events flow from the producer towards consumers. (b) Subscription registrations ("Reg") are propagated upward from the consumers to the publishers ($S_1 \supseteq S_2$).

When some consumer registers or cancels a subscription, the nodes of the overlay must update their routing tables accordingly; to do so, they exchange pieces of information that we call *advertisements*. An advertisement carries a subscription, and corresponds either to a registration or a cancellation. From the point of view of node N, an advertisement for subscription S received from a neighbor node N' indicates that a consumer at N' or downstream from N' has registered or canceled subscription S.

The subscription algorithm works by propagating advertisements recursively across the overlay, from the consumers towards the producers, as illustrated in Figure 1(b). Subscriptions may be transformed along the propagation path due to *aggregation*, which is a key technique to minimize the size of the routing tables by eliminating redundancies between subscriptions, and consequently to improve the routing performance. For instance, at node N_1 in Figure 1(b), two subscriptions S_1 and S_2 have been registered by consumer nodes C_1 and C_2, respectively. From the point of view of node N_3, this means that some consumers downstream N_1 are interested in receiving events matching S_1 or S_2. Now, assume that $S_1 \supseteq S_2$, that is, any event matching S_2 also matches S_1. The mechanism of subscription aggregation is based on the following observation: when an event e arrives at node N_3, it is only necessary to test e against S_1, because, by definition, any event matching S_2 also matches S_1, and any event that does not match S_1 does not match S_2 either. An IP networking analogy would be that of network prefixes, where S_1 is a prefix of S_2. Because of that property, S_2 becomes redundant and can be "aggregated" with S_1 (in particular, S_2 does not need to be propagated upstream from N_1 to N_3).

4. Fault Tolerance in XNET

We have implemented several mechanisms to ensure reliable operation of our XNET system despite the occurrence of router or link failures. The primary objective of these mechanisms is to maintain a *consistent shared state* in the system, i.e., to preserve correct producers-to-consumers routing paths that reflect all the subscriptions registered by the consumers despite transient failures (note that, because of aggregation, each router has only partial knowledge of the subscriptions of its downstream consumers). A secondary goal is to ensure reliable delivery of producer messages; although desirable, this feature is of lesser importance because undelivered messages have no impact on the consistency of the content routing system.

The mechanisms described in this section take different approaches to failure recovery and offer various tradeoffs in terms of cost and benefits. They are also complementary in that they can be easily combined within the same network. We present two recovery-based approaches to reliability, which strive to maintain a consistent global state upon failure. We then discuss a third approach, orthogonal to the other two, which uses redundancy to *mask* problems and provide continuous service despite failures.

Note that, given a spanning tree rooted at a producer, the failure of a router directly affects the neighboring routers downstream from the failed node as they cannot anymore propagate subscription registrations and cancellations towards the producer at the root of the tree. In contrast, the failure of a link only affects the router downstream from the failed link; we can therefore consider router failure as a generalization of link failures, and we will only consider the former type of failures in the rest of the paper.

Also, we only focus on the case of *routing* node failures that can be dealt with transparently by the infrastructure. The failure of a producer node will prevent the distribution of events and force the publisher application to switch over to another node. Similarly, the failure of a consumer node will affect all the attached consumers and must be handled explicitly by the subscriber application.

4.1. The *Crash/Recover* Scheme

The *Crash/Recover* scheme has been designed to cope efficiently and locally with temporary router or link failures. It relies on the assumption that a faulty link or router will recover after a short time. During the downtime period, the producers and consumers can still publish and subscribe to events, i.e., the failure is transparent. After the faulty router or link recovers, the system must reach the same consistent state as if no failure had occurred.

The *Crash/Recover* scheme relies upon a few key mechanisms to cope with transient failures. First, a recovery database is maintained in stable storage on each router. When the router fails, it can recover its state before the crash. Second, the use of the TCP protocol ensures the reliable and ordered delivery of subscriptions and documents. Third, a retransmission buffer coupled with a selective positive acknowledgment scheme is implemented between a router R and its upstream router U. Its purpose is to save the changes that occurred during the downtime of U so that, when it recovers, it can catch up and "roll forward" to a consistent state that corresponds to the current consumer population. Finally, sequence numbers are embedded in all messages to detect duplicates upon recovery and guarantee routing table consistency.

Algorithm 1 On receiving $Adv(sn)$ from interface i

```
1:  if $0 < sn \leq hr_i$ then                                   {Duplicate advertisement}
2:      Send $Ack(sn)$ down interface $i$
3:  else if $sn = hr_i + 1$ then                                 {Expected advertisement}
4:      $hs \leftarrow hs + 1$
5:      Update routing table with $XRoute$ and generate $Adv_{out}(hs)$
6:      $hr_i \leftarrow hr_i + 1$
7:      $RetrBuf \overset{append}{\longleftarrow} Adv_{out}(hs)$
8:      Backup log and routing table in recovery database
9:      Send $Ack(sn)$ down interface $i$
10:     Send $Adv_{out}(hs)$ upstream
11: end if
```

Algorithm 2 On receiving $Ack(sn)$ from upstream

```
1: if $Adv_{out}(sn)$ is found in $RetrBuf$ then
2:     Remove $Adv_{out}(sn)$ from log
3:     Backup log in recovery database
4: end if
```

Algorithm 3 On receiving $Back$ from upstream

```
1: Send $RetrBuf$ upstream
```

The pseudo-code of the *Crash/Recover* protocol is given in Algorithms 1, 2, 3, and 4. Consider router R with n downstream interfaces. Let D_i be the router downstream interface i. Each time D_i sends an advertisement to router R, it includes in it a strictly increasing sequence number (unique between R and D_i). Let hr_i be the highest sequence number received from D_i, i.e., R has received from D_i all the advertisements with sequence number $sn \leq hr_i$. Similarly, hs is the highest sequence number that router R sent to its upstream router. Sequence numbers are used for the positive acknowledgment mechanism and to filter out duplicate advertisements that may be received after a link or router failure.

Each router R maintains a *log* that stores the latest non-acknowledged advertisements sent to its upstream router, as well as the current values of hs and $hr_1 \cdots hr_n$. The log and the routing table of router R are backed up in a *recovery database* (see Figure 2), which is written atomically to stable storage as soon as its state is updated (line 8 in Algorithm 1 and line 3 in Algorithm 2).

When router R receives an advertisement $Adv(sn)$ from interface i, it first checks if the advertisement is a duplicate by comparing sn with hr_i (lines 1 and 3 in Algorithm 1). If that is the case, R sends an acknowledgment to D_i and ignores the advertisement. Otherwise, we have $sn = hr_i + 1$ and we process the advertisement (it is trivial to see from the algorithm and the FIFO ordering property of TCP that we cannot have $sn > hr_i + 1$). R updates its routing table, generates an outgoing advertisement for its upstream router, increments hs and hr_i, and sends an acknowledgment D_i only after local updates have been saved on stable storage (lines 4–9 in Algorithm 1); this guarantees that D_i will resend its advertisement in case R fails before the recovery database has been updated.

Algorithm 4 On recovering from failure

```
1: Recover routing table and log from recovery database
2: Send $RetrBuf$ upstream
3: Send $Back$ downstream all interfaces
```

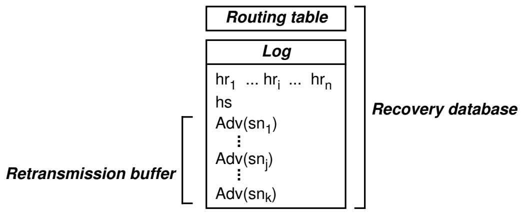

Figure 2. Format of the recovery database.

The retransmission buffer is a stack of advertisements. Each time router R is about to send an advertisement $Adv_{out}(sn)$ to its upstream router U, it appends $Adv_{out}(sn)$ to its retransmission buffer (line 7 in Algorithm 1). When U has received it *and* has updated its routing table accordingly, it sends an acknowledgment for it back to router R (lines 2 or 9 in Algorithm 1), which removes $Adv_{out}(sn)$ from its retransmission buffer (line 2 in Algorithm 2).

If router R crashes, the advertisements that it should have received during the crash duration are not acknowledged and are thus stacked in its downstream routers' retransmission buffer. When R recovers, it first restores its state from the recovery database (line 1 in Algorithm 4). Then, it sends a $Back$ message to its downstream routers, (line 3 in Algorithm 4) to trigger the delivery of the advertisements that were stacked in their logs (Algorithm 3). From the point of view of router R and the routers upstream, everything looks as if R had never failed, except that the "missed" advertisements are received in bursts. After a certain period of time, which we refer to as the *recovery delay*, those routers have updated their routing table and the global system state reflects again the current consumer population.

The fact that the retransmission buffer is backed up in the recovery database and is retransmitted upon recovery before sending the $Back$ message (line 2 in algorithm 4) handles the case when one of R's downstream router, D_i, fails while R is down. When recovering, D_i must first send to R the advertisements stored in its retransmission buffer before processing those received from its downstream routers, so as to preserve consistent ordering of the advertisements sent to R.

4.2. The *Crash/Failover* Scheme

The *Crash/Recover* scheme was based on the assumption that a failed router R will recover after a reasonably short period of time, during which its downstream routers are buffering advertisements. However, the downtime duration of router R may be very long, causing buffers to grow huge or overflow. When R eventually recovers, many advertisements will transit along the paths from R's downstream routers to the producer nodes, potentially creating bottlenecks and delaying system recovery.

The *Crash/Failover* scheme is based on the principle that the downstream routers of a crashed router R do not wait for its recovery, but instead reconnect to another router and bring back their routing tables to a consistent state. Thus we make the assumption that every router R in the network knows at least one additional router other than its direct neighbors, to which it can connect if its upstream router fails. This scheme is very similar to primary/backup replication [13] and we will refer to the additional router as the R's *backup* router, denoted by B_R. Note that, obviously, B_R cannot be located downstream from R with respect to the producer as we must maintain a valid spanning tree after reconnection.

The *Crash/Failover* protocol relies on the fact that every router R has a precise summary of all the subscriptions that its downstream neighbors are interested in. It can thus register/cancel any of these subscriptions at any time by sending an advertisement to its upstream router U, which see the advertisement as if it were the result of a consumer registering/canceling the subscription.

Consider a router R, its upstream router U linked to R via interface I, the set of R's downstream routers $\{D_i\}_{i \leq n}$ and their respective backup routers $\{B_{D_i}\}$. When a downstream router D_i detects that its upstream router R has failed and is unlikely to recover soon (e.g., after a reasonably long timeout), it switches over to its backup router B_{D_i} as new upstream router and registers all the subscriptions stored in its routing table, as if they had just originated from "real" consumers. Note that there are typically far less subscriptions than consumers downstream from D_i because of subscription aggregation, and only the subscriptions that have not been aggregated need to be registered. Once every router D_i has reconnected to B_{D_i}, U can cancel all the subscriptions that were registered through interface I from its routing table to reestablish perfect routing on the path from the producer to U. Clearly, after the recovery procedure has completed, the system state is again consistent with the consumer population. Note that, if D_i has incorrectly suspected R to have failed (e.g., because of a link failure) and has switched over to B_{D_i}, R will cancel all the subscriptions that were registered by D_i.

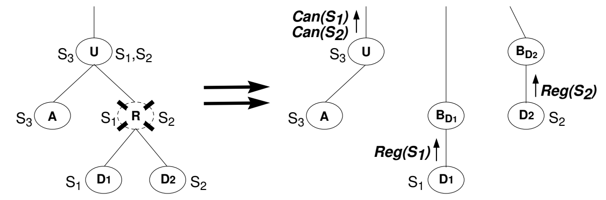

Figure 3. Recovering from the crash of router R with the *Crash/Failover* scheme.

Figure 3 illustrates a simple *Crash/Failover* scenario (the subscriptions that each node is interested in are represented next to the interface they came from; the state before the failure is represented on the left and recovery phase on the right). Routers D_1 and D_2 are interested in subscriptions S_1 and S_2 respectively, while router A is interested in S_3. When R crashes, routers D_1 and D_2 connect to their backup router B_{D_1} and B_{D_2} and register their subscription S_1 and S_2 ("Reg" messages). Thereafter, U can remove all subscriptions previously registered by R from its routing table and propagate the changes upstream ("Can" messages).

The *Crash/Failover* protocol can be advantageously combined with the *Crash/Recover* protocol to deal with temporary link of node failures. If the failure duration reaches a predefined threshold, then the affected routers will switch over to a backup. The subscriptions received from downstream routers are buffered and processed after completion of the reconnection phase. Note that, in the case of simulataneous failures, it might not be possible to use the *Crash/Failover* protocol (e.g., because backup routers have also failed) and the system has to wait for some of the crashed routers to recover.

4.3. Masking Failures with *Redundant Paths*

The *Crash/Recover* and the *Crash/Failover* schemes suffer from two major drawbacks. First, the service is interrupted for the duration of the failure or until the overlay network has reconfigured. Second, they generate upon recovery an upstream traffic of advertisements that can be important, which each advertisement involving routing table updates at the traversed routers. To alleviate these drawbacks, we can combine these schemes with a masking strategy based on *Redundant Paths*, which improves availability by providing uninterrupted service despite failures. In particular, events can be delivered reliably and timely even though some of the routers fail.

The *Redundant Paths* strategy is based on the same principle as active replication [13]. It makes the assumption that each router R has at least one alternate route to the producer. The routing information that corresponds to router R

is replicated in the routing tables of the alternate routes. The implementation of the *Redundant Paths* strategy does not require other modifications to the XROUTE protocol than sending advertisements to all upstream routers (rather than a single one).

If router R has n alternate routes to the producer, it is resilient to the failure of at least $n-1$ upstream routers (in the case of multiple producers, R should have n alternate routes to each producer, but those routes may share common subpaths). When some routers on a route fail, the routers on the other routes are still consistent with the consumer population and R will keep receiving documents from those routes.

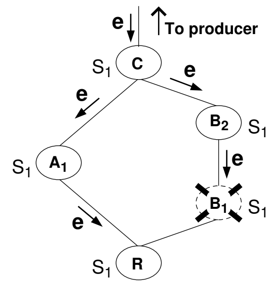

Figure 4. The *Redundant Paths* strategy.

As previously mentioned, it is important to note that the *Redundant Paths* strategy increases the availability (liveness) of the system, but does not deal with recovery. It should be combined with the *Crash/Recover* or *Crash/Failover* protocols to ensure consistent recovery from a failure. The major drawback of the *Redundant Paths* strategy is that every subscription and event will be sent over multiple routes and thus increase bandwidth utilization. Further, routers and consumers must detect and filter out duplicate events.

Figure 4 shows an example of the *Redundant Path* strategy. Router R has two routes to the producer: via routers A_1 and C, and via routers B_1, B_2 and C (the remaining part is common to the two routes and is not shown in the example). Router R is resilient to the failure of router A_1, and to the simultaneous failures of routers B_1 and B_2. In the example, router B_1 crashes and R still received event e via the route $C \rightarrow A_1 \rightarrow R$.

4.4. Reliability of Published Events

Under normal operation, the reliable delivery of published events is ensured by TCP. Guaranteed delivery in the case of failures can be implemented in the same manner as subscriptions in the *Crash/Recover* scheme, by using acknowledgments in combination with a retransmission buffer and a persistent data storage. However, this approach has a high cost in terms of memory and bandwidth requirements as the event publishing rate is typically much higher than the subscription registration rate. Further, events published in content-based networks often need to be delivered timely or not at all, and buffering them is essentially useless; in such cases, one should use the *Redundant Paths* strategy to ensure timely event delivery despite failures. Note again that events do not modify the shared state of the system and the loss of some of them only affects the quality of service experienced by the consumers.

5. Performance evaluation

A major part of our efforts were devoted to building working prototypes and conducting extensive experimental evaluation of our XML content routing network and its various components. We deployed application-level routers on the PlanetLab global distributed platform [15] to simulate a realistic content based network overlay at Internet scale. We conducted extensive performance evaluation of our XNET system to test its efficiency and reliability. The resource utilization of the various components of XNET has been studied in [5, 6, 7].

5.1. Experimental setup.

Network topology. The network topology consists of 21 machines of the PlanetLab network, an open distributed platform for developing, deploying, and accessing planetary-scale network services. PlanetLab was the testbed of choice for us, as it enabled us to experiment with the real conditions of the Internet, especially its unpredictability. Although we had only 22 nodes in our overlay, results are representative of larger networks: As a router only knows its direct neighbors, scalability does not directly depends on the number of routers, but on the consumer population. The machines used in the experiments were running a customized version of Linux. They all had at least 512 MB of memory and a 1.2 GHz processor, but they were used concurrently by several users running similar experiments and their load was very uneven. In practice, as the processing and memory requirements of XNET are moderate, application-layer routers can be easily deployed on low-end machines with limited resources. Each of the 21 PlanetLab machine was hosting a router. As illustrated in Figure 5, 12 of the routers are consumer nodes (boxes), 1 is a producer node (hexagon), and the remaining 9 are routing nodes (circles). The extension of the country where the machine is located is indicated under the node numbers and the average measured link delays are indicated next to every link (upstream delay above, downstream delay below). The routers are organized in a spanning tree rooted at the producer. Each node implements the protocols of our XNET system, that is: the XROUTE routing protocol, the XSEARCH subscription management protocol, and the XTRIE filtering algorithm.

Overlay statistics. Table 1 provides some network statistics about our experimental overlay. All measures are averages over several runs executed at different times. The link delay was measured as the round-trip time to send a packet to a machine and receive a reply over TCP (it does not include the TCP connection establishment time as we are using persistent connections).

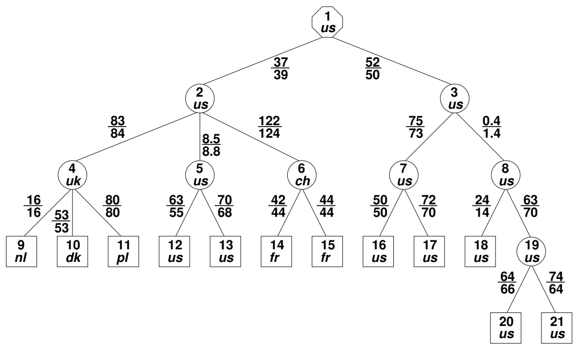

Figure 5. Experimental network topology.

The average minimal routing delay was computed by injecting at the producer an XML document with a single "wildcard" element matching all consumer subscriptions. Consequently, the document was forwarded to all the consumers with minimal process time at the routers. We measured the delay experienced by each consumer to receive the document and we computed the average over all consumers and over $1,000$ runs. This measure gives a lower bound on the routing delay.

Metrics	Value
Average link delay	54.135 ms
Standard deviation of link delays	28.18 ms
Maximal link delay	6 $\longrightarrow$ 2: 123.67 ms
Minimal link delay	2 $\longrightarrow$ 5: 8.47 ms
Average minimal routing delay	160.63 ms
Average minimal update delay (consumer $\longrightarrow$ producer)	169.64 ms
Maximal producer throughput ("single-element" docs)	53.13 docs/s
Maximal producer throughput ("normal-size" docs)	30.28 docs/s
Maximal upstream (consumer) throughput	18.56 sub/s

Table 1. Overlay statistics.

We computed the average minimal update delay as the time necessary to propagate a "wildcard" subscription (requiring negligible process time at the routers) from the consumer to the producer. We computed the average over 100 runs at each consumer and over all consumers. The resulting value gives a lower bound of the update time of the network when a new consumer subscribes to the system.

The maximal producer throughput was computed by sending a burst of $1,000$ documents and measuring the delay between the time the first document was sent until the last document was received by the last consumer. We ran the test both with minimal "single-element" documents and with "normal-size" documents containing 22 tag pairs. The first measure corresponds to the maximal network throughput at the producer, while the second gives an upper bound of the producer rate with realistic event workloads.

We finally computed the average upstream throughput in the same way as we did for the maximal producer throughput: for each consumer, we registered 100 "single-element" subscriptions in a burst and measured the delay until the network has been updated. We then computed the average over all the consumers. This value gives an upper bound of the consumers' arrival rate.

Parameter	Value
Subscription	$h = 10, p_* = 0.1, p_{//} = 0.05, p_\lambda = 0.1, \theta = 1$
Document Size	22 tag pairs
Documents arrival rate	Poisson with rate $\lambda_{doc} =$ 1/s
Consumers arrival rate	Poisson with rate $\lambda_{sub} =$ 1/s
Consumer population	$P = 1,000$ to $50,000$
Crash duration	$D = 1$ to 10 min
Faulty router	2 and 19; 8
Backup routers	3; 2 and 7

Table 2. Parameters of the experiments.

Parameters of the experiments. The parameters of our experiments are summarized in Table 2. We generated tree-structured subscriptions and XML events using the custom generators described in [6]. The subscription parameters control the maximal height (h) of tree patterns, the probabilities of having wildcard and ancestor-descendant operators (p_* and $p_{//}$) and more than one child (p_λ) at a given node, as well as the skew θ of the Zipf distribution used for selecting element tag names. The size of documents was set to 22 tag pairs. We used the NITF (News Industry Text Format) DTD [8] as input to the XPath and XML generators. This application scenario models a single provider producing various types of news reports. Subscriptions represent the interests of individual consumers for some types of documents (e.g., financial news, sports, stories about a specific celebrity). For the sake of simplicity, we assume that each consumer registers only one subscription: a consumer with two subscriptions is considered as two distinct consumers.

The parameters λ_{doc} and λ_{sub} control the arrival rate of documents and consumers, respectively. P defines the size of the existing consumer population, i.e., the number of consumers that are registered in the system when the experiment starts. D controls the duration of a failure before recovery. Finally, we have simulated the failure of various routing nodes of the network and experimented with several configurations of backup routers.

5.2. Performance Under Normal Operation

Routing delay. We are interested in measuring the average routing delay, that is, the average time taken by an event to traverse the network and reach all the consumers that are interested in that event. The protocol of the experiment is as follows: we first populate the network with random subscriptions injected at arbitrary consumer nodes until the consumer population reaches P. We then inject events at the producer node at rate λ_{doc}. For each event, we compute the average routing delay (i.e., producer-to-consumer latency) that was experienced by each consumer node that received the event. Results are average values of $1,000$ runs and are shown in Figure 6. We can see that the routing delay remains small (less than 180 ms) even with large consumer populations. The excellent scalability of the system is mainly due to the high efficiency of the filtering algorithm XTRIE. Indeed, the routing delay is very close to the measured minimal routing delay (Table 1), which indicates that the delay is essentially due to the link delays and not the processing time at the routers.

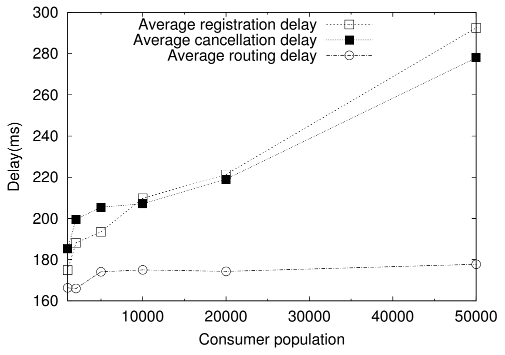

Figure 6. Routing/subscription delay.

Registration and Cancellation delays. To assess the performance of subscription management, we have measured the average delay experienced by a new consumer registering a subscription, given a preexisting population of a given size. This delay corresponds to the time necessary to update all the routers that are affected by the subscription. Given a prepopulated system with P consumers, we generated $1,000$ random subscriptions (which may contain duplicates to model distinct consumers having the same interests) and injected each of them in turn at a consumer node chosen uniformly at random, at a rate of λ_{sub}. After injecting a subscription, we canceled it to maintain a stable consumer population during the whole experiment. We measured for each registration the time necessary to update all the routing tables, and we computed the mean value. To study the cost of subscription cancellations, we proceeded similarly except that, for each of the $1,000$ measurements, we canceled a random subscription. Results are shown in Figure 6.

We observe that the average delay for registering or canceling a subscription increases with the size of the consumer population, but at a moderate rate. Even for large consumer populations, the average delay for a new registration or cancellation remains reasonably small (less than 300 ms). The measured minimal update delay of 170 ms (Table 1) indicates that link delays represent more than 75% of the overall registration or cancellation delay for the considered consumer population sizes. We also observe that the slope of the two curves decreases with the consumer population. This can be explained by the fact that, as the consumer population grows, new subscriptions have higher probabilities of being aggregated and the processing overhead becomes smaller. The process of canceling a subscription is more sensitive to this phenomenon, as can be seen in the Figure (refer to [7] for mode details).

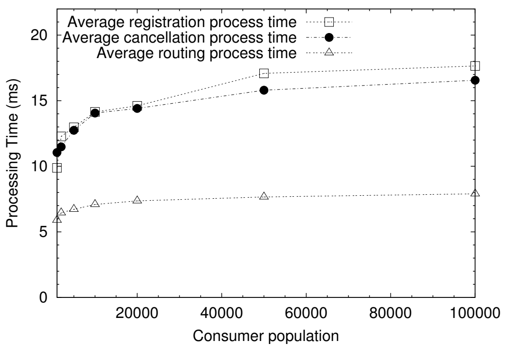

Figure 7. Routing/subscription process time.

Individual Router Performance. Finally, we have measured the performance of an individual router when dealing with subscriptions and published events. Experiments in this paragraph were on a 1.5 GHz Pentium IV machine with 512 MB of main memory running Linux 2.4.18 that was not part of the PlanetLab network.

Figure 7 shows the average process time for registering or canceling subscriptions and for routing XML documents, given existing downstream consumer populations of various sizes. Each result is the average of $1,000$ runs. The results are consistent with those of Figure 6 and corroborate the assertion that the link delays in PlanetLab are responsible for the largest portion of the overhead in the registration, cancellation, and routing delays.

5.3. Performance of the *Crash/Recover* Scheme

Under normal operation (with no system failures), we have just observed that our XNET system is highly efficient and scalable. We now study its behavior when faults occur. We first concentrate on the *Crash/Recover* scheme.

Consider a router R that has crashed at time t_{crash} and recovered at time $t_{recovery}$. We want to measure the recovery delay $D_{recovery}$ until the *whole system* has recovered. Indeed, during the downtime of router R, its downstream neighbors buffer the advertisements (consumer registrations or cancellations) that should be sent to R. Upon recovery, R and its upstream routers must "catch up" by handling all buffered advertisements. The recovery delay is computed as the delay between the recovery time of R ($t_{recovery}$) and the time when the whole system has been updated and reflects the current consumer population.

The protocol of the experiment is the following: considering the system with a preexisting consumer population P and under a consumer arrival rate of λ_{sub}, we kill router

R_i at time t_{crash} and restart it at $t_{recovery}$. We then measure the delay $D_{recovery}$ until the system is up-to-date with no advertisement in the buffers. We are particularly interested in the ratio between $D_{recovery}$ and the crash duration $D_{crash} = t_{recovery} - t_{crash}$.

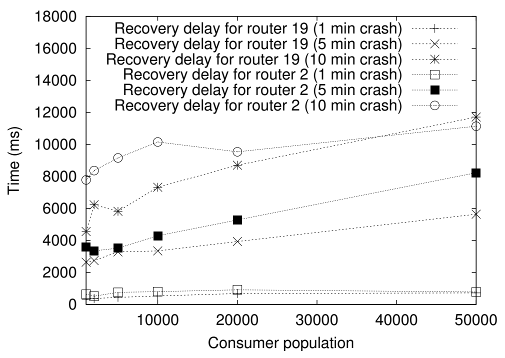

Figure 8. Recovery delays for routers 2 and 19 after crashes of various durations.

We first experimented with the failure of router 2 under various consumer populations and crash durations. We then repeated the same experiments with the failure of router 19. We chose these routers to figure out if the level of the router in the tree topology has an impact on the efficiency of the recovery mechanism.

Figure 8 shows the recovery delays resulting from the crashes of routers 2 and 19, for various crash durations and consumer populations. Table 3 presents the absolute values, in seconds, of the recovery delay D_2 and D_{19} of routers 2 and 19, respectively, as well as the ratios R_2 and R_{19} of the recovery delay to the crash duration ($R_i = \frac{D_i}{D_{crash}}$).

A first observation is that, independently of the failing router, the crash duration, or the existing consumer population, the system is able to recover in a few seconds (typically less than 10 seconds). We can also note that, unsurprisingly, the recovery delay increases with the crash duration because the system needs to process more buffered advertisements to catch up; it does not, however, exceeds 2% of the crash duration. The recovery delay also increases with the consumer population. This is consistent with the observations made in the failure-free experiments. Finally, we observe that there is no significant difference between the recovery delay for router 2 and that for router 19. This can be explained by the fact that router 2 is a high level router and must process more buffered advertisements, but the updates of its routing table are simpler because subscriptions have likely already been aggregated along the way. Router 19 is a low level router and must process fewer advertisements, but it systematically needs to perform more costly aggregation operations (its downstream routers are consumer nodes and hence do not aggregate subscriptions). Therefore, it appears that the distance of the failing router from the producer node does not have a strong impact on the recovery efficiency of the system.

D_{crash}	P	$1,000$	$2,000$	$5,000$	$10,000$	$20,000$	$50,000$
1 min	D_2	637	514	757	799	913	778
	R_2	1.06	.85	1.26	1.33	1.52	1.29
5 min	D_2	3582	3351	3515	4290	5280	8228
	R_2	1.19	1.11	1.17	1.43	1.76	2.74
10 min	D_2	7781	8361	9153	10148	9533	11136
	R_2	1.29	1.39	1.52	1.69	1.59	1.86
1 min	D_{19}	305	374	440	530	681	721
	R_{19}	0.51	0.62	0.73	0.88	1.13	1.20
5 min	D_{19}	2638	2743	3282	3345	3932	5638
	R_{19}	0.88	0.91	1.09	1.12	1.31	1.88
10 min	D_{19}	4559	6228	5818	7322	8700	11709
	R_{19}	0.76	1.03	0.97	1.22	1.45	1.95

Table 3. Recovery delay as function of the consumer population and the crash duration.

5.4. Performance of the *Crash/Failover* Scheme

We finally study the overhead induced by the *Crash/Failover* scheme upon the failure of router 8, which represents a medium level router in the tree topology. We considered two different scenarios for the reconnection of the downstream routers 18 and 19 to their backup routers. In the first scenario, the backup router for both routers 18 and 19 is router 3 (i.e., the closest non-failed upstream router). In the second scenario, router 18 is redirected to router 7 while router 19 is redirected to router 2. Figure 9 shows the new network topologies resulting from both scenarios.

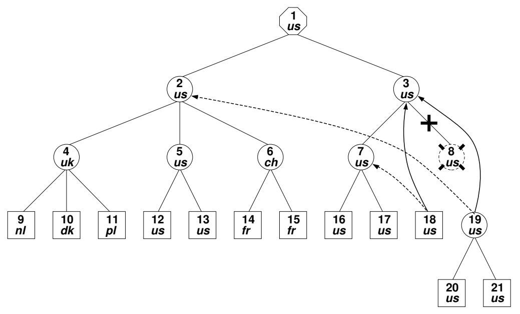

Figure 9. New network topologies for scenarios 1 (plain arrows) and 2 (dashed arrows).

The protocol of the experiment is the same for both scenarios. We first kill router 19 at time t_{crash}. We then redirect the downstream routers 18 and 19 to their backup routers, as explained in section 4.2. We measure the time $t_{recover}$ when the whole system has been updated and reflects the current consumer population. The recovery delay $D_{recovery}$ is the difference between $t_{recover}$ and t_{crash}. For each scenario, we experimented with preexisting consumer population of various sizes. Also, all the experiments were conducted under a constant consumer arrival rate λ_{sub}. Figure 10 summarizes the results that we obtained.

We observe that the recovery delay for both scenarios remains reasonably small, typically less than 1 minute. Also, we can see that the delay increases with the consumer population. This is explained by the fact that the routing tables grow with the consumer population and, during the recovery phase, a portion of the routing tables of routers 18 and 19 must be registered and a portion of that of router 8 must be canceled. In addition, because of subscription aggregation, the routing table updates that must be performed upon recovery are more costly than that of "ordinary" consumer subscriptions. Finally, we observe that the recovery delay for scenario 1 is significantly higher than that for scenario 2 for most consumer populations. This is due to the contention on router 3 and its upstream links, which become bottlenecks in scenario 1; in contrast, the load is split between distinct routers in scenario 2.

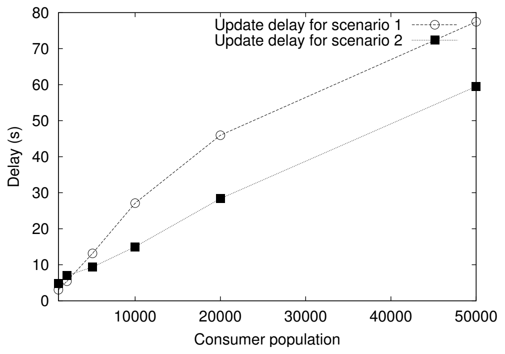

Figure 10. Update time for scenarios 1 and 2.

By comparing the results obtained with the *Crash/Failover* and the *Crash/Recover* schemes, we can conclude that the former should be preferred only for small consumer populations and long crash periods. In systems with large consumer populations, the *Crash/Recover* scheme is more adequate provided that the crashed router eventually recovers. We do not discuss the performance of the *Redundant Paths* strategy as it does not introduce recovery overhead.

6. Conclusion

Our XNET content-based publish/subscribe system integrates several novel techniques to efficiently deliver messages to large consumer populations. In this paper, we have specifically addressed the issue of reliability and evaluated different approaches to fault tolerance, designed to preserve global state consistency and recover from router or link failures. The key principle underlying our recovery mechanisms is that the local state of a router can be reconstructed from the state of its neighbor routers. We have conducted an extensive and realistic performance evaluation of our system by deploying it on the PlanetLab testbed. Experimental results demonstrate that XNET does not only offer very good performance and scalability under normal operation, but can also quickly recover from system failures.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of ICDCS*, 1999.

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

[3] A. Carzaniga, M. Rutherford, and A. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Mar. 2004.

[4] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proceedings of VLDB*, Aug. 2002.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *VLDB Journal*, 11(4):354–379, 2002.

[6] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of NCA*, Apr. 2003.

[7] R. Chand and P. Felber. Efficient subscription management in content-based networks. In *Proceedings of DEBS*, May 2004.

[8] I. P. T. Council. News Industry Text Format.

[9] G. Cugola, E. D. Nitto, and A. Fugetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, Sept. 2001.

[10] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[11] G. Muhl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, TU Darmstadt, Sept. 2002.

[12] G. Muhl, L. Fiege, and A. Buchmann. Filter similarities in content-based publish/subscribe systems. In *Proceedings of ARCS*, 2002.

[13] S. Mullender, editor. *Distributed Systems*, chapter 7 and 8. Addison-Wesley, 2nd edition, 1993.

[14] G. Picco, G. Cugola, and A. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *Proceedings of ICDCS*, 2003.

[15] Planetlab. http://www.planet-lab.org.

[16] A. Snoeren, K. Conley, and D. Gifford. Mesh Based Content Routing using XML. In *Proceedings of SOSP*, Oct. 2001.

[17] W3C. XML Path Language (XPath) 1.0, Nov. 1999.