Journal of Universal Computer Science, vol. 1, no. 2 (1995), 84-104
submitted: 10/1/95, accepted: 8/2/95, appeared: 28/2/95, ©Springer Pub. Co.

A Scalable Architecture for Maintaining Referential Integrity in Distributed Information Systems

Frank Kappe
(Institute for Information Processing and Computer Based New Media (IICM), Graz University of Technology,
A-8010 Graz, Austria
fkappe@iicm.tu-graz.ac.at)

Abstract: One of the problems that we experience with today's most widespread Internet Information Systems (like WWW or Gopher) is the lack of support for maintaining referential integrity. Whenever a resource is (re)moved, dangling references from other resources may occur.
This paper presents a scalable architecture for automatic maintenance of referential integrity in large (thousands of servers) distributed information systems. A central feature of the proposed architecture is the *p-flood* algorithm, which is a scalable, robust, prioritizable, probabilistic server-server protocol for efficient distribution of update information to a large collection of servers.
The *p-flood* algorithm is now implemented in the Hyper-G system, but may in principle also be implemented as an add-on for existing WWW and Gopher servers.

Key Words: Hypertext, Link Consistency, Distributed Information System, Internet, Gopher, WWW, Hyper-G, Scalability, p-flood.
Category: H.5.1, C.2.4

1 Introduction

The problem is quite familiar to all net surfers: Every now and then you activate a link (in the case of WWW [Berners-Lee et al. 94]) or menu item (in the case of Gopher [McCahill 94]), but the resource the link or menu item refers to cannot be fetched. This may either be a temporary problem of the network or the server, but it may also indicate that the resource has been permanently removed. Since the systems mentioned above rely on *Uniform Resource Locators* (URLs) [Berners-Lee 93] for accessing information, it may also mean that the resource has only been moved to a new location. It may also happen that a resource is eventually replaced by a different one under the same name (location).

The net effect of this is that a certain percentage of references are invalid. We may expect that this percentage will rise as time goes by, since more and more documents become outdated and are eventually removed, services are shut down or moved to different servers, URLs get re-used, etc. Obviously, it would be desirable to have some support for automatically removing such dangling references to a resource which is deleted, or at least to inform the maintainers of those resources.

For the sake of the following discussion, let us stick to the hypertext terminology of *documents* and *links* instead of the more general terms resource and reference. The techniques described will work for any relationship between any kind of object, but for explanation purposes is is easier to speak only about links and documents.

Let us assume that we would maintain a link database at every server that keeps track of all the links involving this server, i.e. emanate from and/or point to a document that resides on this server. Storing the links outside of the documents in a link database (like it is done in the Intermedia [Haan et al. 92] and Hyper-G [Andrews et al. 95, Kappe et al. 94] systems) will not only give us an efficient solution for the dangling link problem (as we will see); it also enables more advanced user interfaces for navigation in the information space, such as local maps and location feedback [Andrews and Kappe 94], and allows links between arbitrary document types.

2 Scalability

An important issue that needs to be addressed when dealing with distributed algorithms (systems, protocols) is that of *scalability*. Ideally, the behavior of a scalable system should not – or "almost not" – depend on variables like the number of servers, documents, links, or concurrent users of the system. In the Internet environment, scalability is a very important aspect of system design, since the values of these variables are already high and are continuing to grow extremely fast. Looking more closely at the issue, we may distinguish four kinds of scalability:

- **Scalable Performance:** The performance (measured by the response time perceived by the user) should not depend on the number of concurrent users or documents. This requirement cannot be met in a centralized system, and therefore implies the use of a distributed system, where users and documents are more or less evenly distributed over a number of servers which are connected by a network.
  Unfortunately, it may still happen that for some reason a large number of users access a small set of documents residing on a single server. Under such circumstances the distributed system performs like a centralized system, with all the load placed on a single computer and on a certain part of the network. Obviously, one has to avoid such situations, e.g. through the use of *replication* (placing copies of that scarce resource on a number of servers). A good example of a scalable system which relies heavily on replication is the USENET news service [Kantor and Lapsley 86]. When I read news, I am connected to my local news server which holds copies of the news articles that have been posted lately. When accessing a certain article, it does not need to be fetched from the originating site. Therefore, the response time does not depend on how many other Internet users access the same article at the same time (it does depend on the number of users connected to my local news server, though).
  When searching through a number of documents, the response time will increase with the number of documents searched. Good search engines use data structures giving $O(\log n)$ access performance, i.e. there exists a constant c so that the time t it takes to search n documents is smaller than $c \cdot \log n$. Intuitively, this means that for large n a further increase of n will have less effect on t, which is good. Therefore we may say that logarithmic performance is acceptable for an algorithm to qualify as scalable in performance.

- **Scalable Traffic:** Replication may require additional traffic to be sent over the network. Obviously, every news article has to be sent to every news server so that it can be read locally. However, it may well be that most of the articles that have been sent to my local news server are never read by anyone here. Care has to be taken that total traffic increases not more than linearly with the number of servers. For example, solutions where every server periodically sends state information directly to all other servers are not scalable, since it requires $O(n^2)$ messages to be sent (n being the number of servers this time).
- **Scalable Robustness:** By *robustness* we mean that the system should not rely on a single server or a single network connection to work at all times, nor should it assume that all servers of a given set are available at a given time. The *multi-server transaction*, *master-slave* and *distributed update control* systems described in the next section are all examples that do not scale in this respect.
- **Scalable Management:** The functioning of the system should not rely on a single management entity. For example, the Internet's *Domain Name Service* works because its management is distributed. With the current Internet growth rate of about 3 million hosts per year [Network Wizards 94] (about 10,000 per work day) centralized registration is infeasible. This requirement also suggests that configuration and reconfiguration of server-server communication paths should be automatic, as opposed to managed by a central service.

3 Related Work

3.1 Gopher, WWW and Hyper-G

In the World-Wide Web [Berners-Lee et al. 94] data model, documents are connected by links. The links are stored directly inside the documents, which has the advantage of a simple server implementation. On the other hand, the absence of a separate link database not only limits the set of linkable document types and prohibits advanced user interfaces (overview maps, 3D-navigation, etc.), it also makes it hard if not impossible to ensure the integrity of the Web. When a document is removed, it would require parsing all other documents to find the links pointing to that document, so that they could also be removed or at least the owners of the other documents informed. While such a tool would be conceivable for the local server, it is simply impossible to scan all WWW documents on all Web servers in the world, without the aid of pre-indexed link databases. The consequence is that there is no referential integrity in today's World-Wide Web, not even between documents stored on the same server.

Interestingly, the more primitive Gopher [McCahill 94] system does maintain referential integrity in the local case. When a document (which is an ordinary file on the server's file system) is deleted (or moved or modified), the menu item that refers to it (which is a directory entry) is updated as well. This is automatically taken care of by the underlying operating system (unless you try real hard to break it and use the symbolic links of UNIX). References to remote servers remain insecure, however.

While both Gopher and WWW scale quite well with respect to the number of servers, documents, and links, there is a scalability problem with respect to the number of users. When a large number of users for some reason decides to access the same document at the same time, the affected server and the network region around it become overloaded. This phenomenon (Jakob Nielsen calls it a "Flash crowd" [Nielsen 95] after a 1973 science fiction story of Larry Niven) was observed during the 1994 Winter Olympics in Lillehammer, where the Norwegian Oslonett provided the latest results and event photographs over the Web and drowned in information requests. Similar but smaller flash crowds appear when a new service in announced on the NCSA "What's New" page or in relevant newsgroups.

This problem may be alleviated by the use of *cache servers*, which keep local copies of information which has been recently requested, and give users requesting the same information again the local copy instead of fetching it from the originating site. This strategy does not work, however, in two cases:

1. When users access many different documents from a large data set (e.g., an encyclopedia, a reference database). Replication of the whole dataset would help, but this would in general require moving from URLs to URNs (*Uniform Resource Names*), which identify the document by its name (or ID) rather than location.
2. When the information is updated frequently. Some *update protocol* would be required that ensures that caches are updated so that the latest version of the document is delivered.

In the Hyper-G system [Kappe et al. 93] a full-blown database engine is employed to maintain meta-information about documents as well as their relationships to each other (this includes, but is not restricted to, links). Since the links are stored in this database and not in the documents themselves, and since modifications of documents or their relationships are only possible via the Hyper-G server, referential integrity can easily be maintained for local documents. The link database makes links *bidirectional*, i.e. one can find the source from the destination (as well as vice versa). In order to keep this useful property when a link spans physical server boundaries, both servers store the link information as well as replicas of the remote document's meta-information. This means that all updates related to the documents and the link in question have to be performed on both servers in order to keep the web consistent, thus requiring an *update protocol* between servers.

3.2 Multi-Server Transactions

A possible solution for the update problem is the so-called *multi-server transaction* or *collaborating servers* approach [Coulouris and Dollimore 88]. When a document on one server has to be modified (or deleted), the server storing the document acts as *coordinator* and contacts and informs all other servers which are involved. When all other servers have acknowledged the receipt of the update, the coordinator tells them to make the change permanent. A few more details are necessary to make sure that the transaction is committed by all servers – even in the case of a server crash in the middle of the transaction – [Coulouris and Dollimore 88], but in general, this method works and has been implemented in a number of systems (to my knowledge, first in the *Xerox Distributed File System* [Israel et al. 78]). An earlier version of Hyper-G also adopted this method [Kappe 93].

However, the multi-server transaction has scalability problems in certain situations. When for some reason many servers (say, 1000) decide to refer to a specific document (e.g., by pointing a link to it or by replicating it), all of them have to be informed and acknowledge the update before it can be performed. This not only increases network traffic and slows down things considerably, but it also requires that all servers involved have to be up and running, or the transaction cannot be completed. As the number of participating servers increases (and given the unreliability of the Internet), the probability that *all* of them are reachable approaches zero. This means that it becomes practically impossible ever to modify a heavily-referenced object.

3.3 Master/Slave Systems

In a *Master/Slave System* there is one primary server (the *master*) and a number of secondary servers (the *slaves*). The primary server holds a master copy of the replicated object and services all update requests. The slaves are updated by receiving notification of changes from the master or by taking copies from the master copy. Clients may read data from both master and slaves, but write only to the master.

This scheme is well-suited to applications where objects are read frequently and updates happen only infrequently. The Sun Yellow Pages (YP) service (nowadays known as NIS) is an example of a master/slave system.

The central master server also makes it easy to resolve conflicts between update requests and maintain consistency. The obvious disadvantage is that the master server has to be up and running in order to perform updates. Otherwise, this scheme scales very well (provided that we have a good way of propagating updates from master to slaves).

3.4 Distributed Update Control

The *Distributed Update Control* [Coulouris and Dollimore 88] scheme allows any server that holds a copy of an object to perform updates on it, without a single coordinating server, even when some servers are unreachable, and without the possibility for conflicts.

This requires that a server knows about all the other servers that also have copies (let us call this set of servers the *server-set*). In a perfect world, all the copies would be identical, but because of network failures and for performance reasons it may not be possible or desirable to immediately notify all servers of an update. We may instead adopt a looser form of consistency (*weak consistency*), in which all copies eventually converge to the same value at some time interval after the updates have stopped.

However, one still wants to be sure that all read requests are based on up-to-date copies and all updates are performed on the latest version. The trick which ensures this is *majority consensus*: updates are written to a (random) majority of the server-set (more than 50%). Before every read or write operation, the server that is in charge of performing the request contacts some other servers of the server-set and requests the object's version number (or modification time) to identify the current version. When a majority has answered, at least one of it has the current version. This is because in every two majorities there is at least one common member.

The advantage of this algorithm is its robustness: There is no single point of failure and it works even in the face of failure of almost 50% of the server-set. The downside of it is again scalability: The server-set for any object must be known to all members of the server-set, and more than 50% of the set has to be contacted before every write and even read operation. If the set contains, say, 1000 servers, we have to get a response from 501 of them!

This requirement may be relaxed for read operations if we are willing to accept *weak consistency*. Still, it is mandatory for write operations to ensure that no conflicting updates can occur.

3.5 Harvest and *flood-d*

Harvest [Bowman et al. 94] is a new Internet-based resource discovery system which supports an efficient distributed "information gathering" architecture. So-called "Gatherers" collect indexing information from a resource, while the so-called "Brokers" provide an indexed query interface to the gathered information. Brokers retrieve information from one or more Gatherers or other Brokers, and incrementally update their indexes. The idea is that Gatherers should be located close to the resources they index, while Brokers are located close to the users.

Harvest heavily relies on replication to achieve good performance. The indexes created by the Gatherers are periodically replicated to the Brokers. Since the indexes tend to be large, this has to be done efficiently.

Harvest uses a technique called *flooding* for this purpose. Rather than having a Gatherer send its indexes to all Brokers, they are sent to only k of them (e.g., $k = 2$). It is then the responsibility of the k chosen nodes to distribute the indexes to another k each, and so on. While of course the total number of indexes that have to be transferred remains the same, flooding has the nice property of distributing the network and server load over the whole network.

The flood algorithm used by Harvest is called *flood-d* [Danzig et al. 94]. *Flood-d* tries to minimize the network cost and propagation time of the flood by computing a "cheap", k-connected logical update topology based on bandwidth measurements of the underlying physical network. An important requirement was that this topology should not need manual configuration (like for example the Network News [Kantor and Lapsley 86]), but shall be computed and updated automatically. Finding a good approximation of the optimal topology is computationally expensive, however (finding the optimum is even NP-complete), especially when the replication group becomes very large. The paper [Danzig et al. 94] therefore suggests to use a hierarchical scheme of smaller replication groups. However, it is left open how this hierarchy can be found and updated automatically.

4 An Architecture for Referential Integrity

Let us assume that we maintain a link database at every server which keeps track of all the links local to the server as well as those that go in and/or out of the server, i.e. emanate from and/or point to a document residing on another server. Maintaining referential integrity is relatively easy in the case of local links. We will now concentrate on the issue of maintaining integrity in the case of links which span server boundaries.

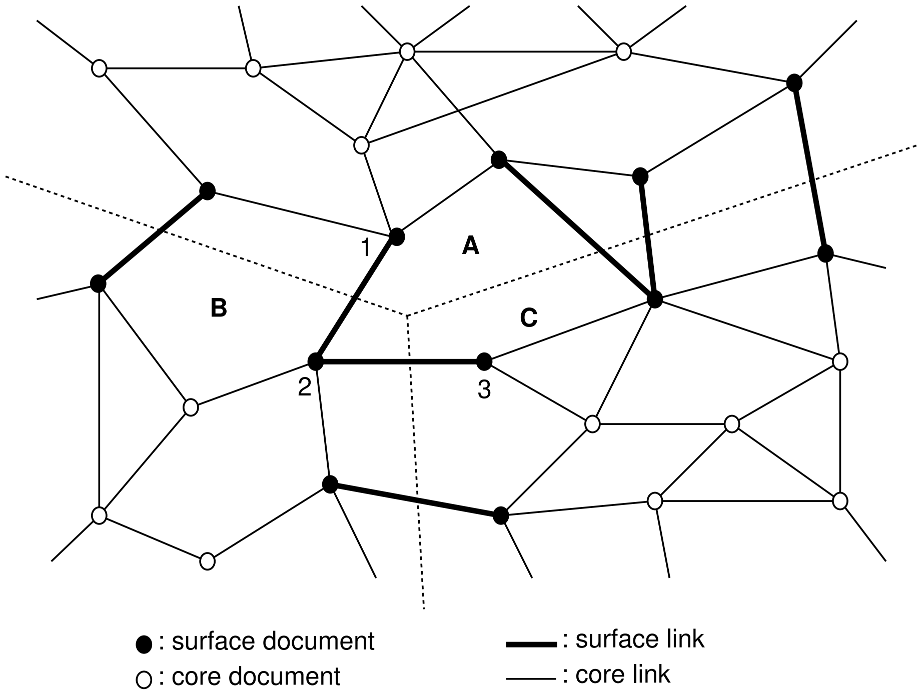

Figure 1: Partitioning the Web among Servers (see text)

Figure 1 illustrates this situation. The hyperweb is partitioned by server boundaries (the servers are labeled A, B, and C in the figure). Links which span server boundaries are shown as thicker edges. We will call these links *surface links*, and documents connected to other servers by such links shall be called *surface documents* (the others are called core links and core documents, respectively). Although not apparent from Figure 1, a server's surface will typically be small compared to its core.

In order to keep the useful property of *bidirectional links*, the link information of surface links must be stored in both affected servers. For increased performance, the servers also keep replicas of the other surface document's meta-information. In figure 1, server A stores document 1 plus a replica of document 2's meta-info and the link between them, while server B stores document 2 plus replicas of documents 1's and 3's meta-info and the links from 1 to 2 and from 2 to 3.

In this setup, documents on different servers are interconnected as tightly as the documents on a single server. The bidirectional links enable more advanced navigation techniques (the link map shown in Figure 1 can actually be computed and shown to the user), but also simplify maintenance of the hyperweb: when I choose to remove document 2, the system can inform me that this will affect document 1 on server A and document 3 on server C (among others on server B). I may either use this information to manually modify the affected documents and links, or let the system ensure automatically that at least the links from 1 to 2 and from 2 to 3 are removed as well.

The problem which remains is how to inform the other servers that document 2 has been removed. As already mentioned, an earlier implementation of Hyper-G used the knowledge about what documents are affected to directly engage the other servers in a multi-server transaction, in order to remove document 2 and all links to and from it. As was also discussed earlier, this approach has problems when many servers must participate in the transaction (because many links point to the document).

Therefore, we decided to adopt a *weak consistency* approach, whereby we accept that the hyperweb may be inconsistent for a certain period of time, but is guaranteed to converge to a consistent state eventually. Of course, we would like to keep the duration of the inconsistency as short as possible.

Like in the master/slave model, updates may only take place at a well-defined server. Unlike the master/slave model, this server is not the same for all operations, but depends on the document or link being modified (or removed or inserted): For documents, it is the server which holds the document; for links, it is the server which holds the document where the link emanates (in our example, server B would be responsible for updates of document 2, while the link from 1 to 2 would be updated by server A). This reduces the problem of overload of the master, while eliminating the problem of conflicting updates (they are handled one after the other). One disadvantage remains: the master server must be available at update time. However, since for security reasons users wishing to update document 2 must have write permission for document 2 (this is checked by server B which holds document 2), this fact is inevitable and we have to live it, anyway.

Updates of core documents or core links require no further action (integrity is maintained by the local link database). However, other servers need to be notified of updates happening at a server's surface (i.e. updates of surface documents or surface links). We chose to use a flood algorithm similar to the one employed by Harvest to propagate updates from the master to the slaves (i.e. all other servers), because of its scalability (the traffic generated does not depend on the number of references to the object in question), because it does not require that the recipients are available at update time, and because it can be used for other purposes as well (like distributing server addresses and statistics, and maintaining the consistency of replicas and caches).

5 The p-flood algorithm

The *flood-d* algorithm described in [Danzig et al. 94] is optimized for minimizing the cost of the flood. This makes sense because it is designed for applications which need to flood large amounts of data. Our application – sending update notifications – sends only small messages ("document 2 removed" can be encoded in a few bytes), and hence has somewhat different requirements:

- **Speed:** Messages should propagate fast in order to minimize the duration of inconsistencies.
- **Robustness:** The protocol should *guarantee* eventual delivery of every message to every server, even when some servers are down. When a server that

has been unavailable comes up again, it should receive all the messages it has missed in between.

- **Scalability:** The time it takes to inform all servers should not depend heavily on the number of servers. Likewise, the amount of traffic generated should not depend heavily on the number of servers. Of course, since every message must be sent to every server at least once, $O(n)$ is a lower bound for the total traffic generated.
- **Automatic:** We do not want to configure flood paths manually (like in the News service).
- **Priority:** Since we intend to use the protocol for other purposes as well, it would be nice to have a priority parameter attached to every message that determines its acceptable propagation delay and bandwidth consumption.

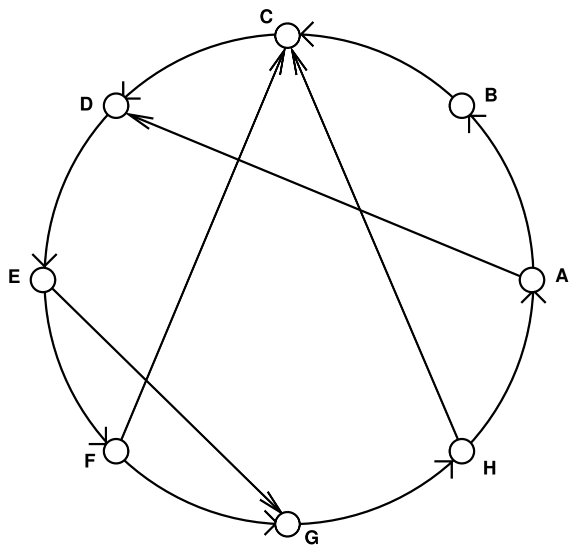

Figure 2: One step of the *p-flood* algorithm ($p = 1.5$)

The *p-flood* algorithm is a probabilistic algorithm which fulfills the above requirements. Figure 2 illustrates its behavior. The servers are arranged in a circle (for example by sorting them according to their Internet address; see section 7.1 for a discussion how this can be done in a better way). Every server knows all other servers (updates of the server list will of course be transported by the algorithm itself).

Servers accumulate *update messages* which are generated either by the server itself (as a result of modification of a surface document or surface link), or are received from other servers, in their *update list*. Once in a while (every few minutes) the update list is sent to p other servers ($p \geq 1$). We will call this time period a *step* of the algorithm. For $p = 1$, updates are sent only to the immediate successor, otherwise they are also sent to $p - 1$ other servers that are chosen at random. If p is fractional, they are sent to other servers only with probability $p - 1$. For example, $p = 1.3$ means that one message is sent to the successor, and another one with probability .3 to a random server; $p = 3.2$ means that it is sent to the successor, two other random servers plus one other random server with probability .2.

Figure 2 shows one step of the p-flood algorithm with $p = 1.5$. Note that at every step the operations described above are performed by all servers in parallel, i.e. within the step time period every server performs one step (the clocks of the servers do not have to be synchronized). We may observe that at every step $p \cdot n$ update lists are sent (n being the number of servers).

The higher the value of p, the shorter the time it takes to reach all servers, but the higher the amount of traffic generated (it happens that the same message is received more than once by some servers). The algorithm in principle allows the assignment of different values of p to individual messages, so we may call p the *priority* of the message.

After a message has successfully been transmitted to a server's immediate successor, it is removed from the sending server's update list and not sent again to any server in future steps. This ensures that messages are removed after they have been received by all servers and keeps the update lists relatively short. Messages are time-stamped using a per-server sequence number, so that duplicates can be discarded and messages can be processed in the correct order by the receiver.

What happens when a server is down or unreachable? Since a message must not be discarded from the update list until it has successfully been sent to the successor (we assume that a reliable transport protocol like TCP is used and that receipt is acknowledged), the message will effectively wait there until the successor comes up again. Almost immediately after that, the accumulated update messages will be sent. In a way, every server is responsible for delivering messages to its successor. The penalty is that when a server is down for a long period of time, its predecessor's update list grows.

Setting the priority $p = 1$ (send messages only to the successor) will effectively block update messages in case of an unreachable server and is therefore not feasible. A higher values of p not only speeds up the propagation of the messages significantly, but also contributes to the robustness of the algorithm. In the example of Figure 2, a crash of server B would not inhibit update messages from server A being propagated to the other servers.

A few extensions of *p-flood* are necessary for real use. They are described later in section 7, in order to not burden the reader with additional complexity at this point.

6 Simulation Results

This section presents data gathered by running extensive simulations of *p-flood*. We will first concentrate on "perfect world" simulations (i.e. all servers are reachable) and then look at the effect of network and server faults.

6.1 The Behavior of *p-flood* in the Perfect World

In this first set of experiments we want to find out how weak exactly our weak consistency is, i.e. how long it takes to arrive at a consistent state after updates have stopped, how this time depends on the number of servers and the priority factor p, and how much traffic is generated over time.

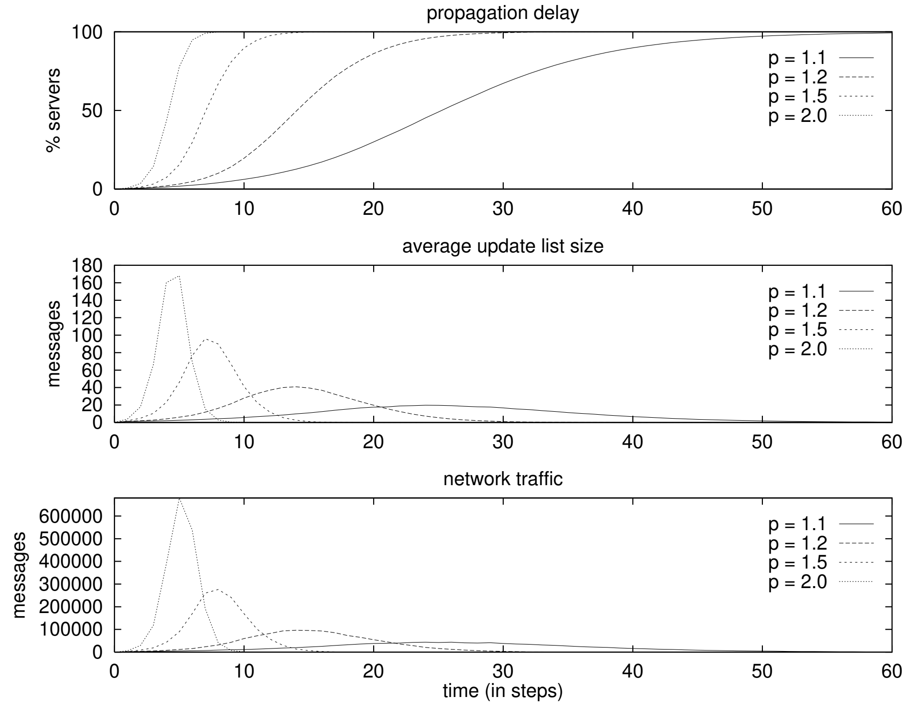

Figure 3: Performance of *p-flood* at different values of p ($n = 1000, m = 1000$)

Figure 3 gives us a feeling of how *p-flood* performs. It is assumed that m update messages have been generated at the n different servers before the simulation starts, and we watch their propagation to the other servers, in particular how long it takes until they arrive there. It turns out that it does not matter whether all m updates are made on a single server or whether they are distributed randomly over the n servers, but the random placements gives smoother curves, so I have chosen this method for producing the graphs.

The top graph shows how the update information is propagated to the 1000 servers, using different values of p. A higher value of p gives faster propagation, e.g. at $p = 2$ and $n = 1000$, 50% of the servers are reached after about 4 steps, 99% after 7 steps, and the last one is typically updated after 10-13 steps. The price for faster propagation is a higher load on the servers and networks: The middle graph shows the average size of the update list held at each server, and the bottom graph shows the traffic in messages that is sent at each step.

Since every message has to be sent to every server at least once, every algorithm that delivers every message to every server will need to transmit at least $m \cdot n$ messages, so we will call this number the *optimum* traffic. Under perfect-world conditions, the total traffic sent by *p-flood* is $p \cdot n \cdot m$ messages, or $p \cdot optimum$. The point is that the flood algorithm distributes this traffic nicely over time and over the whole network, as opposed to the trivial solution where every server simply sends all its updates to all other servers (which requires only *optimum* messages to be sent). The lower the value of p, the more network-friendly the update.

Clearly, there is a tradeoff between fast propagation and peak network load. Figure 3 suggests that a good setting of p is somewhere between 1 and 2.

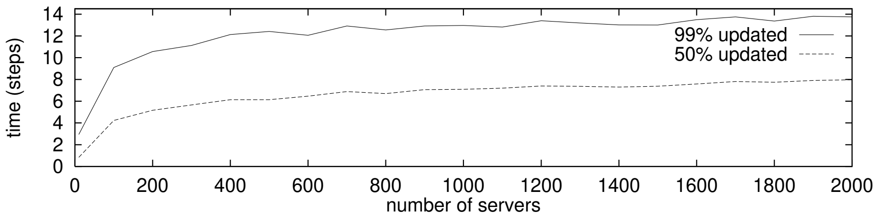

Figure 4: Time to update 50% (99%) of the servers ($p = 1.5$) by n

Figure 4 demonstrates the remarkable scalability of *p-flood* with respect to the number of servers. The time to reach 50% and 99%[1] of the servers is plotted against the number of servers. The logarithmic performance of *p-flood* is clearly visible, meaning that that *p-flood* is well-suited for use in the context of very large server groups.

Figure 5 plots the propagation delay (again, reaching 50% and 99% of the servers) versus the priority p for a constant number of servers ($n = 1000$).

When I first ran the experiments, I was surprised to see that the messages traveled about twice as fast as I had expected. For example, for $p = 1$ it takes about 500 steps (not 1000) to reach all 1000 servers, for $p = 2$ it takes about 4 (not 9; $2^9 = 512$) steps to reach the first 500 servers, etc.

It turns out that this happens because the clocks in the servers which determine the step time are *not* synchronized. When one server's timer expires and the server then sends its update list to another server, the other server's timer will in general not expire a full step time later, but after some random time interval between 0 and the step time. On average, it will expire after half of the step time, which explains the observed effect.

In the simulation, the behavior of the parallel server processes was modeled on a single computer, i.e. for every step the individual servers performed their update operations one after the other. If implemented carelessly, this serialization could lead to wrong results: In the example of figure 2, processing the servers in the order A-B-C-D-E-F-G-H would always propagate all updates to all servers in one single step! Instead, reality was modeled more closely by processing the servers in the order given by a randomly chosen permutation of the servers,

[1] The time to update 100% of the servers is of course always an integer value and subject to a great deal of randomness. The 99% value can be interpolated and is less affected by randomness, so I decided to use this value to get smoother curves in the graphs. The 100% value is about 3-6 steps higher.

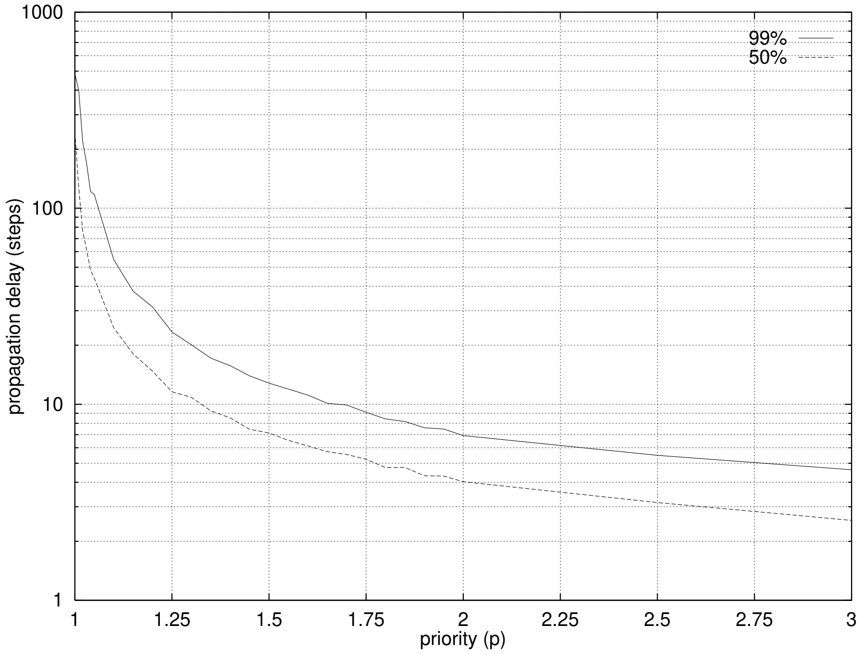

Figure 5: Time to update 50% (99%) of the servers, by p ($n = 1000$)

mimicking the random expiration of the server's timers. Still, messages will travel slightly slower in reality, especially if the step time is not large compared to the average transmission time of updates lists.

6.2 Network and Server Failure

Since one of the major requirements for our flood algorithm was its robustness with respect to network and server failures, we will now take a close look to this issue.

First, let me make the distinction between *soft* or transient errors and *hard* or persistent errors. Soft errors are temporary network errors due to routing problems or network overload. This type of errors occur rather frequently in the Internet, but usually only for a short period of time. Hard errors last longer (e.g, as a result of a hardware failure) but fortunately happen less frequently.

Figure 6 shows the effect of soft errors on the propagation delay and the traffic generated. The propagation delay (i.e. the time to reach 50%/99% of the servers) increases only slowly with increased soft error rate. A soft error rate of 10% means that in every step 10% of the update propagations will fail (10% of the servers are unreachable), which are chosen randomly. In the next step, another random set of 10% fail, but is unlikely that the two sets are identical.

The bottom graph of Figure 6 shows how traffic increases with increased soft error rate. The set of messages that is sent increases, but the number of acknowledged messages (i.e. that are sent successfully) remains constant. Since

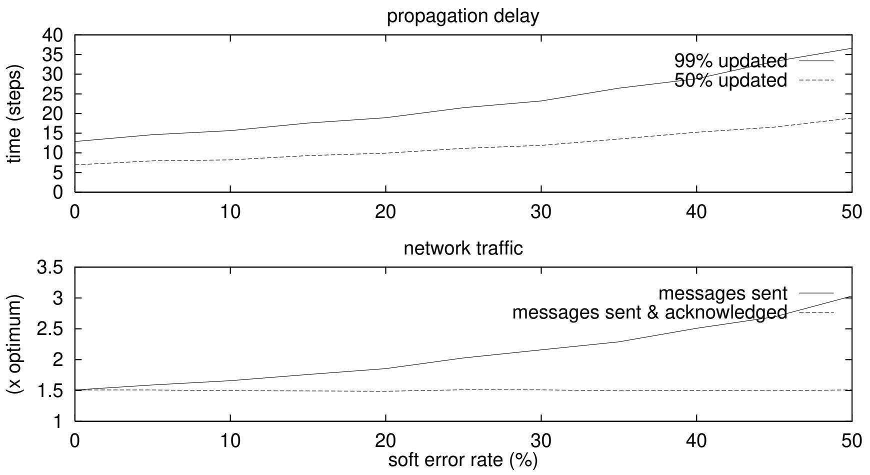

Figure 6: Effect of soft errors on propagation delay and traffic ($p = 1.5, n = 1000$)

p-flood detects duplicate messages and messages that arrive out of order itself, we could in principle use an unreliable protocol like UDP to transmit messages and acknowledgments. However, UDP is very much subject to soft errors (e.g., packets dropped). Using TCP, the transport protocol repairs a large number of soft errors itself. When a server is temporarily unreachable, this will usually already been detected during the opening of the connection, and the messages will never be sent in this case. This means that for TCP the "messages sent" graph is not significant, i.e. the number of messages actually sent is constant with respect to soft errors.

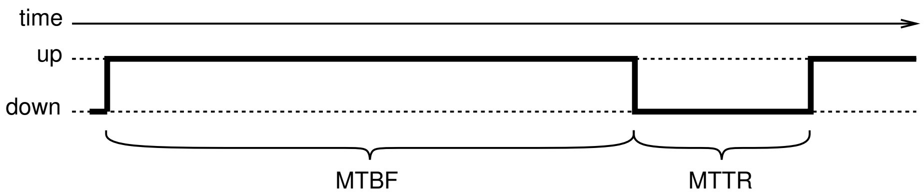

Figure 7: MTBF and MTTR

Hard errors are usually described by the two variables *Mean Time Before Failure (MTBF)* and *Mean Time To Repair (MTTR)* (Figure 7). Then *uptime*, i.e. the fraction of time the server is up, is defined as

$$uptime = \frac{MTBF}{MTBF + MTTR}$$

In our simulations, we will measure MTBF and MTTR in units of steps. For $MTTR = 1$ we have soft (transient) errors, larger values of MTTR mean that a same server is down for a longer period of time. It is expected that server *uptime* will be well over 90% (this is already a bad value; it means that a server will be unreachable for 2.4 hours per day). In the beginning, $MTTR/(MTBF+MTTR)$ servers are marked as down, with the time they remain down chosen randomly between 0 and MTTR. The others are assigned a time they remain up between 0 and MTBF. It is assumed that the servers which are down also carry update messages (they could have been accumulated before they went down; the servers could be only unreachable from the outside but running). During the simulation, servers that come up remain up for MTBF steps, those that go down remain down for MTTR steps.

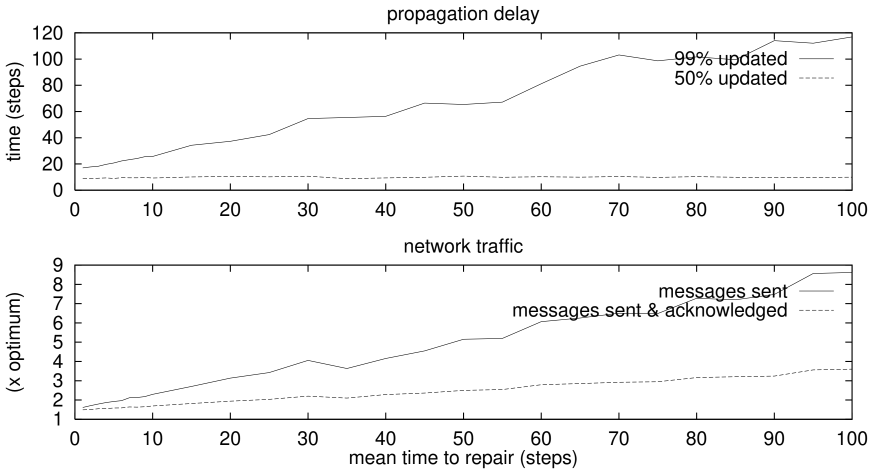

Figure 8: Effect of hard errors ($p = 1.5, n = 1000, uptime = 90\%$)

In Figure 8, *uptime* is constantly 90%, and MTTR varies between 1 and 100. The top graph shows the effect on the propagation delay. Because of the probabilistic nature of *p-flood* there is almost no effect on the remaining servers, which is why the time to reach 50% of the servers remains almost constant. Of course, there is an impact on the time to reach 99% of the servers (because only 90% are available), which grows linearly with MTTR. The number of messages sent also grows linearly. This time the number of messages which are sent and acknowledged (i.e. the ones that would be sent when we use TCP) also increases, but only slowly.

Figure 9 takes a closer look at the effect of hard errors on the performance of *p-flood*. In order to make the effects clearly visible, 10% of the servers (100) are held down (or unreachable) until step 50, when they all come up simultaneously. The graphs can be divided in three phases:

During the first phase (until about step 17), updates propagate almost as usual (only slightly slower than the $p = 1.5$ curve in Figure 3), but level off when

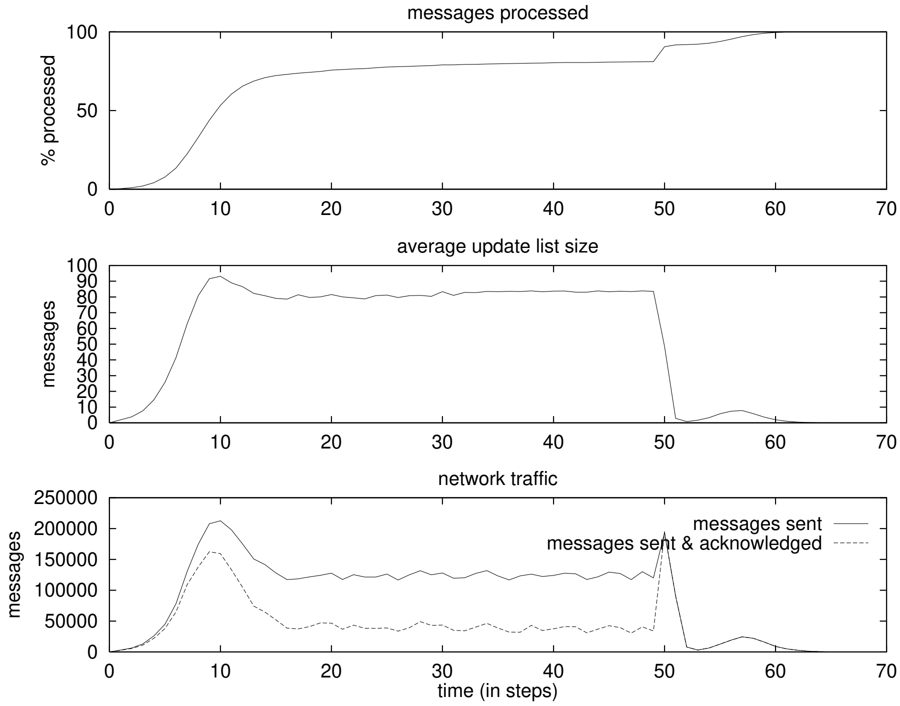

Figure 9: 10% of the servers are down until step 50 ($p = 1.5, n = 1000, m = 1000$)

approximately 81% of the updates are processed (because 90% of the updates are then processed by 90% of the servers; remember that the 10% unreachable servers also generated 10% of the updates).

In the second phase (steps 18 to 49) the system is in a more or less stable state. Since this state may last for a long time (until the servers come up again; in our experiment this is step 50), it is worth analyzing what exactly happens during this phase. We may observe that the set of servers can be partitioned in three disjoint groups:

1. Those that are down. If we call d the fraction of servers that are down, then the number of servers in this group is $d \cdot n$ (in our example: $d = 0.1, n = 1000 \Rightarrow d \cdot n = 100$).
2. The predecessors of the servers that are down. Since a predecessor may also be down (with probability d), in which case it belongs to the previous group, the number of members of this group is $(d - d^2) \cdot n$ (in our example: 90).
3. The others, i.e. those that are up and not predecessors of an unavailable server. The number of such servers is $(1 - d)^2 \cdot n$ (in our example: 810).

The members of group 3 are able to flush their whole update list already during the first phase, i.e. their update list size during phase 2 is zero. Members of group 2 keep messages destined for the group 1 members. The number of such messages is $(1 - d) \cdot m$ at each server. The unavailable servers carry m/n

messages each. The average update list size (as plotted in the middle graph of Figure 9) during phase 2 is therefore

$$\frac{(d-d^2)\cdot n\cdot(1-d)\cdot m + d\cdot n\cdot\frac{m}{n}}{n} = m\cdot(d-2d^2+d^3+\frac{d}{n}) \quad (=81.1)$$

The members of group 2 constantly try to send messages to their successors (which are group 3 members) and to other random servers. At $p = 1.5$, every group-2-server sends .5 messages per step to random nodes (which will succeed with probability $(1-d)$), plus one message per step to its successor (which is guaranteed to fail). The members of group 1 and 3 send nothing. The total number of messages sent at each step of phase 2 can therefore be calculated as $p\cdot(d-d^2)\cdot n\cdot(1-d)\cdot m$, in our example $1.5\cdot 90\cdot 900 = 121,500$. Out of these, $(1-p)/p$ are successful (in our example, one third or 40500). The calculation corresponds nicely with the simulated results shown in the bottom graph of Figure 9. Again, the number of messages sent and not acknowledged is insignificant if we use TCP as transport protocol, because the fact that a server is down is discovered already when attempting to open the connection, and nothing will be sent if the open fails.

In the third phase, after the unavailable servers come up at step 50, the members of group 2 immediately succeed in propagating their update lists to their successors, which causes the dramatic effects shown in Figure 9 shown around step 50. After that, it takes a few more steps to propagate the updates that were kept by the unavailable servers to all servers.

6.3 Traffic Estimates

Let us now try to estimate the (additional) amount of network traffic that would be caused by applying the described architecture to a distributed information system. In order to do so, we assume the following values of variables:

- 1,000 servers (n).
- 100 surface objects (documents and links) per server, i.e. we assume a total of 100,000 surface objects. Note that the number of core objects would be much higher but is irrelevant here.
- 10% of the surface objects are updated per day, i.e. a total of 10,000 updates messages (m) have to be propagated every day. Again, updates of core objects are irrelevant.
- Note that while the traffic generated is dependent on the number of servers and documents, it does not depend on the number of users of the system.

Then, the total number of messages sent per day is $p\cdot optimum$, with *optimum* $= n\cdot m = 10^7$ messages (every message shall be delivered to every server). A message is a few bytes long (say, 10). At $p = 1.5$, we would generate network-wide traffic of 1.5×10^8 bytes (150 MB) per day, or 4.5 GB per month.

On the other hand, the NSFnet currently (Nov. 94) transmits about 22,462 GB per month [Merit Network Information Center 94]. If we assume that 25% of the whole (non-local) Internet traffic pass the NSFnet (i.e. the whole traffic is about 90,000 GB/month), this means that the update messages of our information system would cause an additional 0.005 % of network traffic; in other words, the effect is negligible.

Another nice property of *p-flood* is that the traffic is distributed evenly over the whole set of servers, which means that the amount of traffic seen by a single server does not increase when the number of servers is increased[2]. It only depends on the number of updates of surface objects. Of course, it may is possible to operate servers in an isolated fashion, in order to avoid being molested by any traffic at all.

Since every update message has to be propagated to every server, the nature of *p-flood* can be compared to the USENET news service. However, the messages are much smaller than news articles. The numbers given are for perfect-world performance. Consult Figures 6 and 8 to see the effect of soft and hard errors on network traffic.

7 Extensions of *p-flood*

The description of *p-flood* in section 5 was a bit simplistic for the purpose of understanding the simualtion results. However, the following details have to be addressed when actually implementing *p-flood*:

7.1 Arranging the Servers

A potential weakness of *p-flood* is its random usage of logical Internet connections, without knowledge of the underlying physical network. There is no preference of fast links over slow ones, as in *flood-d*. On the other hand, random selection of flood paths propagates the updates faster than the cost-based selection [Danzig et al. 94], which tends to use the same links again and again.

However, *p-flood* chooses its propagation paths in both non-probabilistic (the immediate successor) and probabilistic (among the other servers) ways. The amount of randomness is controlled by the p parameter. Since for reasonable values of p (see the simulations in section 6) most traffic runs over the static circle of servers (see figure 2), clever arrangement of servers in this circle can vastly reduce network cost and delay, without giving away the advantages of fast propagation and robustness by random choice of some of the flood paths.

Computing the optimal circle using actual bandwidth measurements would be difficult, since it would require gathering a fully connected matrix of bandwidth between servers. Furthermore, the measurements would have to be repeated quite frequently, because global network utilization changes with the time of the day. Hand-configuring is not considered an option. Therefore, we choose a more pragmatic, heuristic approach:

Servers are sorted according to their reversed fully-qualified domain name. Server i then is the successor of server $i-1$, and the first server is the successor of the last one. Sorting by reverse domain name (i.e. by last character first) results in all servers in for example Belgium (domain .be) being neighbors in the circle, followed by the servers in Germany (domain .de) and so forth. Within Germany, the servers located in, e.g., Berlin will be neighbors (domain -berlin.de). Since

[2] As has been shown before, the total Internet traffic generated *does* depend on the number of servers.

in most cases local connectivity is cheaper and faster than international connections, this simple trick will result in better use of the available bandwidth[3]. No computations (other than sorting) and measurements are necessary.

7.2 Adding and Removing Servers

When a server is added to or removed from the server list, *p-flood* itself – with a high priority p – is used to notify all servers. The servers modify their server list accordingly (using the sort order described in section 7.1).

During propagation of server list updates (addition and removal of servers, moving a server to a different host) it is important that a server uses its old server list for flooding, until the message has been acknowledged by the successor. Simple modifications of server attributes (e.g., description, Internet address, e-mail of administrator) do not require such precautions.

7.3 Recovery after Catastrophic Events

When operating a large number of servers it may and will happen (not too often, hopefully) that catastrophic events occur which result in loss of information (for example a head crash on the server's disk). In such cases, operation needs to be resumed from a backup copy of the information base. If the backup copy is i days old, then the restarted server has lost all its updates of the last i days. This is inevitable, though.

However, other servers may also have a now obsolete picture of our server's surface. For example, somebody may have recently (less than i days ago) created a new document in our server, with a link pointing to (or from) another document on another server. The document has now disappeared and of course this link also has to be destroyed in order to keep a consistent state. In other words, the other servers also have to roll back to the situation i days ago.

In such a situation, our server may flood a special message that contains its whole surface (documents and links), thus requesting all other servers to check this picture against their view of our server, and adjust their information about our server accordingly.

7.4 Repairing Inconsistencies

Under certain conditions an inconsistency in the hyperweb may occur. For example, let us assume that a link is made from a document on server A to a document on server B. The document on server B was not previously on the surface. At about the same time (i.e., before the update message reflecting this operation arrives at server B) server B deletes the document the links is going to point to. Since it is not on the surface there is no need to inform other servers about the deletion, so server A will not be notified and will keep its link.

Server B can detect this inconsistency when the update message from server A eventually arrives, since it requests creation of a link to a non-existing object.

[3] Unfortunately, host names in the US domains (.edu, .com, .gov, .mil, .net) in general do not give any hints on the host's geographical location, with the exception of the new .us domain).

It may now flood a "document removed" message for this non-existing object, as if it had been on the surface.

Alternatively, we may choose to live with such (rare) inconsistencies for a while, and have all servers periodically flood their whole surface, like after a catastrophic event (section 7.3). This would serve as a fall-back mechanism that deals with all kinds of inconsistencies and errors, including unforeseeable hardware and software errors in the update server. Since these messages may be rather long, they should be sent infrequently and with low priority. The exact time and priority will have to be determined when we have a feeling of how often such problems occur.

8 Summary

The paper presents a scalable architecure for guaranteeing referential consistency in large, distributed information systems, for example distributed hypermedia systems like Gopher, WWW, and Hyper-G.

Server objects (documents, links) are divided into surface and core objects. We assume that servers are able to maintain referential integrity for core objects (like the Hyper-G server) themselves, so only surface objects need to be treated specially when modified. A fast, robust, prioritizable flood algorithm, *p-flood*, is specified to propagate messages containing update information about surface objects between servers.

Extensive simulations of *p-flood* show that the protocol is scalable, fast, and can cope with spurious errors and persistent failure of servers and networks. The traffic generated is negligable compared to other sources of Internet traffic.

The architecture described and *p-flood* is now being implemented in the Hyper-G system, but can in principle be applied to any kind of distributed information system. 1

References

[Andrews and Kappe 94] Andrews, K., Kappe, F.: "Soaring Through Hyperspace: A Snapshot of Hyper-G and its Harmony Client". In Herzner, W., Kappe, F. (editors), Proc. of Eurographics Symposium on Multimedia/Hypermedia in Open Distributed Environments, Graz, Austria. Springer (1994), 181–191.

[Andrews et al. 95] Andrews, K., Kappe, F., Maurer, H.: "Hyper-G: Towards the Next Generation of Network Information Technology". Information Processing and Management (1995). Special issue: Selected Proceedings of the Workshop on Distributed Multimedia Systems, Graz, Austria, Nov. 1994.

[Berners-Lee 93] Berners-Lee, T.: "Uniform Resource Locators". Available on the WWW at URL `http://info.cern.ch/hypertext/WWW/Addressing/URL/Overview.html` (1993).

[Berners-Lee et al. 94] Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., Secret, A.: "The World-Wide Web". Communications of the ACM, 37, 8 (1994), 76–82.

[Bowman et al. 94] Bowman, C. M., Danzig, P. B., Hardy, D. R., Manber, U., Schwartz, M. F.: "Harvest: A Scalable, Customizable Discovery and Access System". Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder (1994). Available by anonymous ftp from `ftp.cs.colorado.edu` in `/pub/cs/techreports/schwartz/Harvest.ps`.

[Coulouris and Dollimore 88] Coulouris, G. F., Dollimore, J.: "Distributed Systems: Concepts and Design". Addison-Wesley (1988).

[Danzig et al. 94] Danzig, P., DeLucia, D., Obraczka, K.: "Massively Replicating Services in Autonomously Managed Wide-Area Internetworks". Technical report, Computer Science Department, University of Southern California (1994). Available by anonymous ftp from `catarina.usc.edu` in `/pub/kobraczk/ToN.ps.Z`.

[Haan et al. 92] Haan, B. J., Kahn, P., Riley, V. A., Coombs, J. H., Meyrowitz, N. K.: "IRIS Hypermedia Services". Communications of the ACM, 35, 1 (1992), 36–51.

[Israel et al. 78] Israel, J. E., Mitchell, J. G., Sturgis, H. E.: "Separating Data from function in a Distributed File System". In Lanciaux, D. (editor), Operating Systems: Theory and Practice, 17–27. North-Holland, Amsterdam (1978).

[Kantor and Lapsley 86] Kantor, D., Lapsley, P.: "Network News Transfer Protocol – A Proposed Standard for the Stream-Based Transmission of News. Internet RFC 977". Available by anonymous ftp from `nic.ddn.mil` in file `rfc/rfc977.txt` (1986).

[Kappe 93] Kappe, F.: "Hyper-G: A Distributed Hypermedia System". In Leiner, B. (editor), Proc. INET '93, San Francisco, California. Internet Society (1993), DCC–1–DCC–9.

[Kappe et al. 93] Kappe, F., Pani, G., Schnabel, F.: "The Architecture of a Massively Distributed Hypermedia System". Internet Research: Electronic Networking Applications and Policy, 3, 1 (1993), 10–24.

[Kappe et al. 94] Kappe, F., Andrews, K., Faschingbauer, J., Gaisbauer, M., Maurer, H., Pichler, M., Schipflinger, J.: "Hyper-G: A New Tool for Distributed Hypermedia". In Proc. Distributed Multimedia Systems and Applications, Honolulu, Hawaii. IASTED/ISSM, ACTA Press, ISBN: 0-88986-194-3 (1994), 209–214.

[McCahill 94] McCahill, M. P., Anklesaria, F. X.: "Evolution of Internet Gopher". Information Processing and Management (1995). Special issue: Selected Proceedings of the Workshop on Distributed Multimedia Systems, Graz, Austria, Nov. 1994.

[Merit Network Information Center 94] Merit Network Information Center: "NSFNET Backbone Statistics". Up-to-date figures are available by anonymous ftp from `nic.merit.edu` in `/nsfnet/statistics` (1994).

[Network Wizards 94] Network Wizards: "Internet Domain Survey". Up-to-date figures are available on the WWW at URL `http://www.nw.com/zone/WWW/top.html` (1994).

[Nielsen 95] Nielsen, J.: "Multimedia & Hypertext: The Internet and Beyond" Academic Press (1995).