Scalable and Secure Resource Location

Robbert van Renesse

Dept. of Computer Science, Cornell University
4118 Upson Hall, Ithaca, NY 14853
E-mail: `rvr@cs.cornell.edu`

Abstract

In this paper we present Captain Cook, a service that continuously monitors resources in the Internet, and allows clients to locate resources using this information. Captain Cook maintains a tree-based representation of all the collected resource information. The leaves in the tree contain directly measured resource information, while internal nodes are generated using condensation functions *that aggregate information in child nodes. We present examples of how such information may be used for cluster management, application-level routing and placement of servers, and pervasive computing. The nodes are automatically replicated, updates being propagated using a novel* hierarchical gossip *protocol. We analyze how well this protocol behaves, and conclude that updates propagate quickly in spite of scale, failed nodes, and message loss. We describe how Captain Cook can be made secure using Public Key Certificates without compromising its scalability.*

1. Introduction

We introduce the term *navigation service* for services that provide continuously updated information about resources in the Internet, and organize this information so that resources are easily located. Besides resource location, the navigation service may be used for cluster management, application-level routing, server placement, version control, and pervasive computing. A navigation service may store such information as load on each machine in the network, which machines run a particular service, as well as which clients are using particular services. Although a directory service such as X.500 [12] could be used to store and organize resource information, such an implementation would not be able to keep up with the rate of queries and updates if potentially billions of resources in the network are reporting their status changes, and billions of users are tracking these changes. The intention of X.500 is to map object names onto meta-information such as locations, whereas the primary goal of a navigation service is to locate resources based on resource information.

In this paper we present the design and implementation of a navigation service called *Captain Cook*, after the famed explorer who mapped the world before being clubbed to death. Other than a typical directory service, Captain Cook has no centralized servers. Instead, every machine on the network plays a little part in maintaining the entire service. This minimizes management and allows trivial deployment of Captain Cook.

The monitoring information is stored in one or more trees. Each leaf node in such a tree corresponds to a particular machine, and contributes part of its local Management Information Base (MIB) [13]. An interior node is generated by a so-called *condensation function* (CF), which aggregates the information of its children, and produces a MIB for the collection of its children. While leaf MIBs are updated directly, an internal MIB cannot be updated directly, but its CF can be changed on-the-fly.

The tree is distributed over the participating machines, with MIBs automatically replicated on several machines. Captain Cook uses *gossip* to guarantee that updates eventually propagate through the entire tree. Gossip protocols combine the efficiency of hierarchical dissemination with the robustness of flooding protocols. Although our current implementation is not secure, we believe that scalable security can be enabled through the use of public key certificates.

This paper is organized as follows. In Section 2, we describe how Captain Cook presents the information being monitored. Section 3 then presents four hypothetical examples to demonstrate the usefulness of Captain Cook. Section 4 describes the current implementation of Captain Cook, and analyzes how quickly updates propagate as a function

[0] ©2000 IEEE. Published in the Proceedings of the Hawaii International Conference on System Sciences, January 4–7, 2000, Maui, Hawaii. This work is supported in part by ARPA/ONR grant N00014-92-J-1866, ARPA/RADC grant F30602-96-1-0317, ARO/EPRI contract W0833-04, and NSF grant EIA 97-03470.

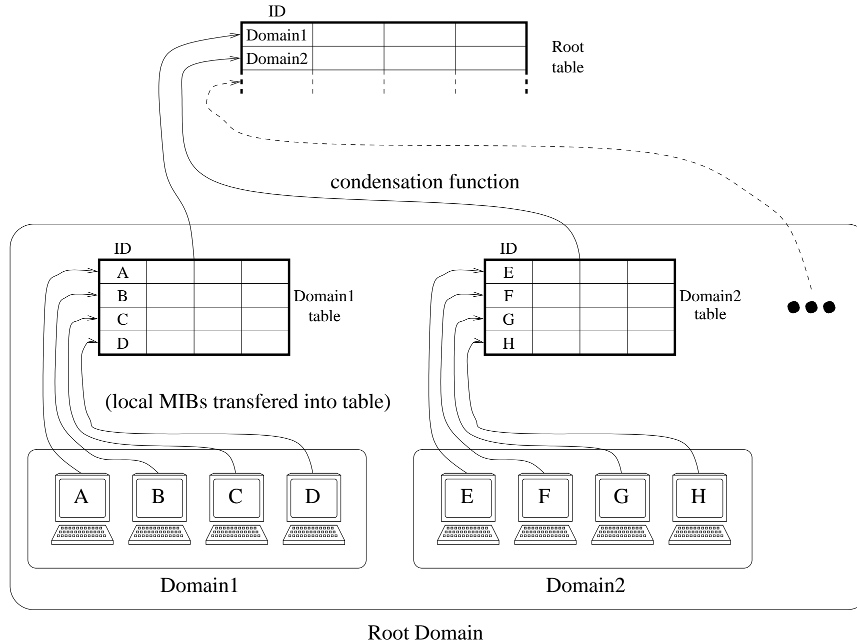


Figure 1: An example of a two-level HMIB. The top-level *root domain* has subdomains *Domain1* and *Domain2*, each having four hosts in it. The MIBs of hosts A–D in Domain1 are placed together into a table, and replicated on those hosts. This table is then *condensed* into a single MIB which is placed in the root table. Ditto for Domain2. The root table is replicated on all hosts.

of scale. A scheme to add security to Captain Cook is suggested in Section 5. We discuss related work in Section 6, and conclude in Section 7.

2. Presentation

The Captain Cook Service presents its information in one or more *Hierarchical Management Information Bases* (HMIBs). An HMIB is a tree of two-dimensional tables (see Figure 1). A node in an HMIB represents a *domain* of participating machines. In the case of a leaf node, there is a row in the table for each machine in the corresponding domain. In the case of an internal node, there is a row for each child domain. A row is like a traditional MIB, containing such information as average load on the machines in the domain, or the presence of a particular resource. Because of this, we use the terms row and MIB interchangeably.

The columns in a table make up the categories of information used in the rows. Each column is identified by a descriptive string, such as "average load". A column is typed, in that the information stored in each row of the column is of the same type (for example, a number, a boolean value, or an address).

A table is required to have at least an "ID" column and a "Contacts" column. Each MIB is uniquely identified by a *MIB ID*, and this identifier has to be stored in the "ID" column of the corresponding row. The "Contacts" category in a row contains a list of *contacts* for that row. In the case of a row in a leaf table, each contact in the list is a (Transport Protocol, Address) pair through which the corresponding machine may be reached. In the case of a row in an internal table, the list of contacts is a small, representative subset of contacts in the domain corresponding to that row.

Each table has an associated *condensation function* (CF). A CF is a function that takes a table as input, and, deterministically, generates as output a MIB calculated from this table. In other words, the CF aggregates the information in the table. Typical operations include taking the minimum, maximum, average, median, or sum of particular columns. For example, if the input table has a "Load" column, the CF output may include an "Average Load", as well as the "#Members" with the number of rows in the input table. Each CF is required to choose a representative subset of contacts from the input table (typically by choosing a contact from each of the first two or three rows).

A row in the internal nodes of an HMIB is generated by taking the table of the corresponding child domain, and applying its CF. In order for an HMIB to scale well, it is important that a CF reduces, approximately, the amount of information in its input table by a factor equal to the number of rows in the table or more. Hence the term condensation.

A participating machine only stores those tables in the HMIB that lie on its path to the root of the tree. That is, a machine stores the table of the leaf domain that it's in, and all the tables of the ancestor nodes. (As a result, each node stores the table of the root node.) For example, in Figure 1, machine E only stores the Domain2 and the Root tables. Because of the structure of the HMIB, and the condensing property of the CFs, the amount of information stored on each machine grows only logarithmically with the number of participating machines. Effectively, each machine knows detailed information of immediate peers in the HMIB, but the *granularity* of information about resources degrades, exponentially, with distance in the tree.

Depending on the structure of the tree and the CFs chosen, a machine may *zoom* into more detailed information starting from an internal node, eventually learning the exact location of resources. While zooming in, the machine has to use the corresponding "contacts" to get the information.

There are three sources of updates in an HMIB. A participating machine is allowed to update its own MIB in the leaf domain it belongs to. Such an update eventually propagates to the peer machines in that domain. If the update changes the output of the CF of the domain, the update also propagates to the corresponding row in the parent domain. Recursively, the update in the parent domain spreads further sideways and upwards.

The second source of updates is due to machines joining and leaving an HMIB. A machine may join into an existing leaf domain, or insert a new leaf domain into an internal domain. Machines can also leave an HMIB (cease participation), either intentionally or through a crash. MIBs of those machine are removed from the HMIB, as well as MIBs of domains that become empty. The last source of change in an HMIB is due to changing CFs. Machines can update the CFs of the domains they belong to, changing the categories of values that are being computed.

There are no strong consistency guarantees on information in the HMIB. In the absence of malicious or Byzantine behavior, updates are guaranteed to eventually propagate to the entire HMIB, or be replaced by even newer updates. But machines may see updates in different orders. Nevertheless, we find that this weak form of consistency is sufficient for many applications.

3. Examples

To illustrate the use of a navigation service, we present examples for four important areas of distributed computing: Cluster Management, Application-Level Routing, and Pervasive Computing. The examples have not been worked out into detail—each example could be the subject of an entire paper on its own (and will be if our plans develop). The intention of including these examples in this paper is only to show that the HMIB is a useful tool in distributed computing systems.

3.1. Cluster Management

A department wants to run a replicated service called "CryWolf." The department would like to run servers on three of the ten lightliest loaded machines in a cluster consisting of hundreds of machines, moving the servers around if necessary.

For this first example, we will sketch an implementation using a one-level HMIB tree consisting of a single table. The table is equipped with two columns: "Load", and "CryWolf". Each machine regularly updates its load in the Load column. Each machine that runs a CryWolf server stores the value "true" in the Crywolf column. Such a table might look like this:

ID	*Contacts*	*Load*	*CryWolf*
Amundsen	10.0.4.1:1872	.3	true
Pizarro	10.0.4.2:1475	3	false
Polo	10.0.4.3:1254	0	true
Frobisher	10.0.4.4:1535	2	false

.
.
.

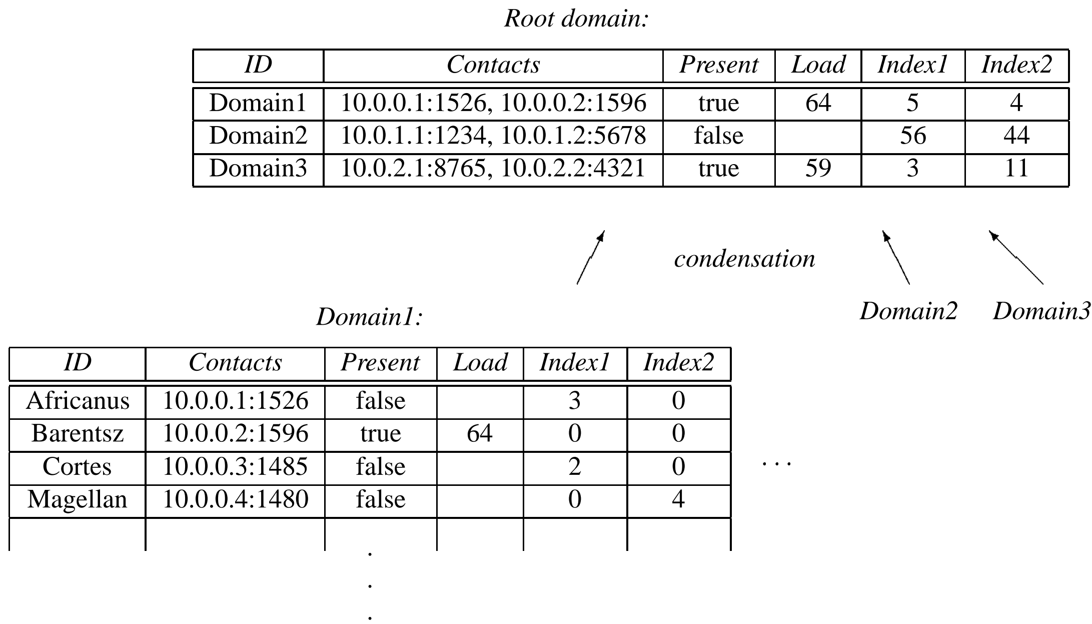
The administrator runs a simple script that tries to make sure that there are three servers running, and that they are running on three of the ten machines with the smallest loads. The script has three rules: (1) kill servers on machines other than the ten lightliest loaded ones; (2) if there are more than three servers running, kill the server on the machine with the heaviest load; (3) if there are fewer than three servers running, start a server on the machine with the least load and which does not have a server running already.

The servers themselves use the HMIB to locate each other for communication. They may also create an additional column with version information in order to synchronize. The HMIB is also useful to the clients of the service: they use the "CryWolf" column to locate the servers.

3.2. Application-Level Routing

A Digital Library (DL) service like NCSTRL [4] replicates indices of entire collections of articles on so-called "index servers" at several sites. DL clients (running on Web servers) need to choose one of the index servers for satisfying queries from Web browsers. DL managers would also like to be able to find "hot spots" so that they can decide where to place index servers.

For this, an hierarchical HMIB may be created that mimics the Internet hierarchy of hosts, subnets, and domains.


Root domain:

ID	*Contacts*	*Present*	*Load*	*Index1*	*Index2*
Domain1	10.0.0.1:1526, 10.0.0.2:1596	true	64	5	4
Domain2	10.0.1.1:1234, 10.0.1.2:5678	false		56	44
Domain3	10.0.2.1:8765, 10.0.2.2:4321	true	59	3	11

Domain1:

ID	*Contacts*	*Present*	*Load*	*Index1*	*Index2*
Africanus	10.0.0.1:1526	false		3	0
Barentsz	10.0.0.2:1596	true	64	0	0
Cortes	10.0.0.3:1485	false		2	0
Magellan	10.0.0.4:1480	false		0	4

Figure 2: This figure shows part of a two-level HMIB for use in a digital library service. The service runs two index servers: Index1 on a machine (Barentsz) in Domain1, and Index2 on a machine in Domain3. The tables describe the current location of the index servers, the loads on the index servers, and the load that each client domain contributes to the load on each index server. From this information, we can see that there is a "hot spot" in Domain2, thus the administrator may want to move one of the index servers there.

Below, we assume that there is just one collection, and two levels in the hierarchy.

Each table in the HMIB has a column for every index server, plus two extra columns called "Present" and "Load" (see Figure 2). An entry in the "Present" column contains a boolean value indicating if the corresponding domain contains an index server (or *is* an index server in case the domain is a leaf in the HMIB). For an internal MIB, an entry in this column is calculated by a CF that takes the boolean OR of the values in the "Present" column of the input table. The "Load" column in leaf nodes contains the load on the server. For internal MIBs, the CF calculates the minimum load. The index columns contain the load imposed by clients in the domain on the corresponding index servers. The CF sums these loads for internal MIBs.

The DL manager uses the index columns to determine where the load on these servers originates from in order to decide if the servers are correctly placed or need to be moved (or additional servers created). DL clients use the "Present" and "Load" columns to decide where to send queries. Selection is random, reversely weighted by the load on the servers.

3.3. Pervasive Computing

We are currently working on a system that allows students in our university to locate available desktops, printers, etc.& in labs. Students will have PDAs that contain a snapshot of the HMIB information stored at one of the participanting machines of the system. Students will be able to update their PDAs by docking them in one of various locations, or by pointing PDAs at each other and using the IR ports.

The initial design of the HMIB is essentially a three-level hierarchy. There is a leaf table for each lab, listing each machine in the lab and their availability. CFs compute the number of available machines and printers, first on a per building-level, then on the root level that covers the entire university. There can be multiple columns, say one for Unix systems, one for Windows systems, one for Macintoshes, and one for printers.

A student who needs a desktop first checks the local leaf domain to find an available machine, and, if unsuccessful, ascends the tree until s/he finds a domain with available machines. The student may then descend into a particular domain using the contact information in the HMIB, or, more likely, decide to walk over to the corresponding lab or building before trying again.

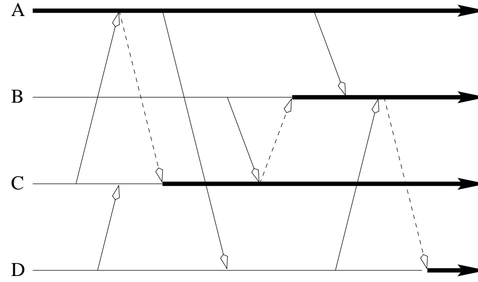


Figure 3: Four hosts are gossiping among each other. Time goes from left to right. Host A has an updated MIB. The gossip messages are indicated by solid lines, while responses that include the update are shown as dashed lines. Eventually, all hosts receive the update.

4. Implementation

Captain Cook applies a gossip protocol in order to disseminate changes to the information stored in an HMIB. The basic idea is as follows (see Figure 3). Each row is timestamped using the wall clock time of its last update. Periodically, each machine chooses a random row in its local tables, and, using the "Contacts" information stored in that row, sends a gossip message. The gossip message contains a list of (MIB ID, timestamp) pairs for all the rows stored at the gossiper. On receipt of such a message, the receiving machine compares this list to the information it has stored locally, and can decide which of the MIBs it has are more up-to-date than the gossiper's (using the timestamps). It then returns a list of updates to the gossiper. The gossiper merges these updates into its own HMIB. As we will demonstrate, this protocol propagates updates quite quickly through the entire tree (in spite of various benign failures), but care need be taken to scale the protocol properly.

More precisely, each machine gossips every T_{gossip} seconds. At such a time, the machine picks a random row in the table of its leaf domain. It then picks a random contact from the list of "Contacts" in that row to send the gossip message to. (Actually, the choice is not entirely random—it is constrained by the intersection of transports that the sender and receiver support.) If the machine happens to pick itself as a contact, it repeats the process for the parent domain, and so on. (If the machine keeps picking itself, even when it reaches the root domain, it terminates by not sending a gossip message at all.) Thus, on average, each participant sends, and receives, about one gossip message every T_{gossip} seconds.

This strategy ensures that most gossip exchanges are in the local domains, and the gossip exchange rate decreases exponentially with the distance between any two participating machines in the tree. Within any particular domain at any level in the hierarchy, there is about one gossip message from every row to every other row once every T_{gossip} seconds. To see why this is so, observe that every machine in a domain of m machines (with m rows in the corresponding table), gossips in its parent's domain with a probability of $1/m$ (assuming for simplicity that there are only two levels). Since all m machines are therefore gossiping at a rate of $1/(m \times T_{gossip})$ gossips per second, the combined rate in the parent domain is $1/T_{gossip}$ gossips per second.

We are assuming that clocks are loosely synchronized for three reasons. First, if a sender and a receiver of a gossip message do not have synchronized clocks, the receiver may not determine correctly which information it has is more up-to-date than the sender's. Worse yet, a machine may replace new information with older information. Second, if machines gossip at widely different rates, due to inconsistent notions of T_{gossip}, the analysis of the protocol becomes much more complex. Finally, we will use synchronized clocks to detect and remove failed machines and empty domains from the HMIB (Section 4.2).

4.1. Analysis

We will now analyze how fast updates propagate in the face of failed participating machines and occasional message loss. We assume that failures are benign and stochastically independent. Gossip protocols can be analyzed using epidemiological theory [2, 5, 3, 14]. A typical way of doing this is using stochastic analysis, based on the assumption that the execution of the protocol is broken up into synchronous rounds during which each participating machine gossips once. A participant that has new information is called *infected*. For analysis, initially only one participant is infected, and this is the only source of infection in the system. At each round, the probability that a certain number of participants is infected, given the number of participants already infected, is calculated.

We will base our analysis on [14]. It observes that gossip protocols are not run in synchronous rounds in practice. Each participant gossips at regular intervals, but the

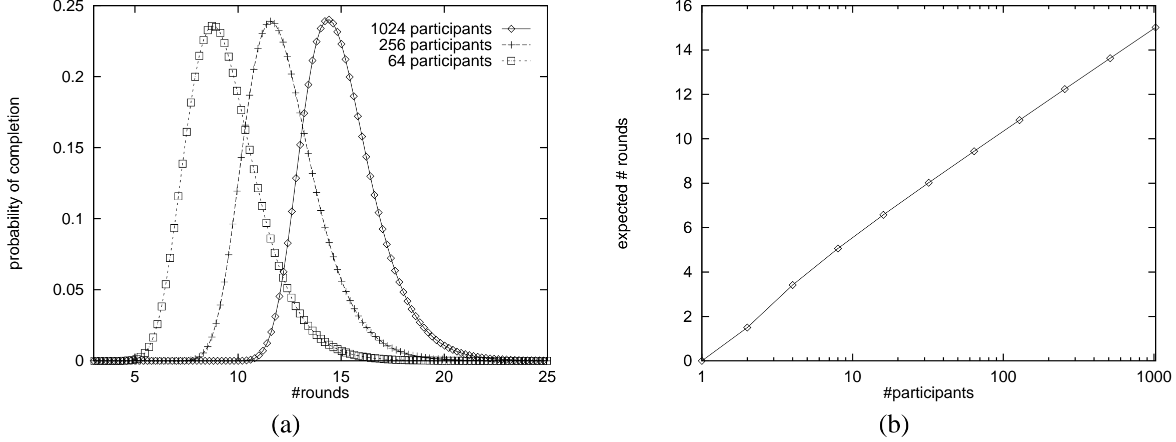


Figure 4: (a) The distribution of the number of rounds necessary to infect all participants; (b) The expected number of rounds necessary to infect all participants as a function of the number of participants.

intervals are not synchronized with each other. [14] subdivides rounds into *micro-rounds*, with one micro-round for each participant. It assumes that no more than f participants fail during the dissemination of the infection, and, conservatively, that all f participants have failed from the start. It also assumes that the initially infected participant does not fail ([14] argues that the effect of this assumption on the results of the analysis is negligible). Besides being more realistic, the advantage of these assumptions is that in each micro-round at most one participant can get infected, greatly simplifying the analysis.

Initially, we ignore the hierarchy, and consider a "flat" system of n participants. In a micro-round of p, p initiates a gossip exchange with another participant q. p is infected only if q is infected and not crashed, p was not infected already, and neither message in the gossip exchange got lost. Let k be the number of infected participants, and $P_{success}$ be the probability that a gossip exchange is successful (*i.e.*, no message is lost and the exchange completes within a micro-round). Then the probability that this exchange increments the number of infected participants is:

$$P_{inc}(k) = \frac{k}{n} \times \frac{n-f-k}{n} \times P_{success} \qquad (1)$$

The probability that the number of infected participants in micro-round $i+1$ is k is:

$$\begin{aligned} P(k_{i+1} = k) &= P_{inc}(k-1) \times P(k_i = k-1) \\ &+ (1 - P_{inc}(k)) \times P(k_i = i) \end{aligned}$$

(with $0 < k \leq n-f$, $P(k_i = 0) = 0$, $P(k_0 = 1) = 1$, and $(P(k_0 = k) = 0$ for $k \neq 1)$.

We are interested in $P(k_r = n - f)$, the probability that all participants are infected as a function of the number of rounds r. We have plotted this for n is 64, 256, and 1024 participants and $f = 0$ in Figure 4(a). From the analysis we can also calculate the expected number of rounds necessary to infect all participants. This is shown in Figure 4(b) as a function of the number of participants. Upper bound analysis on gossip protocols reveals that gossip slows down by $O(\log n)$ [2, 3].

To see how the speed of infection depends on the number of failed participants, we have plotted the effect of failed participants on the expected number of rounds in Figure 5(a), for 64, 256, and 1024 participants. As can be observed, gossip is quite resilient to failed nodes. Even if half the number of participants fail, the slow down is less than a factor of two. Gossip propagation is also highly resilient to large probabilities of message loss (Figure 5(b)).

This analysis cannot be generalized easily to the hierarchical gossip protocol used by Captain Cook. To see the effect of adding a hierarchy, we have simulated the spread of infection in a two-level hierarchy, The participants were spread evenly over a number of domains, ranging from one domain containing all n participants to n domains each containing one participant. It is easy to see that both extreme cases result in behavior approximately identical to that of "flat" gossiping, but that the spread of infection of the configurations in between results in slower propagation.

The result of the simulation for 64, 256, and 1024 participants ($f = 0$) is shown in Figure 6(a). It plots the average number of rounds necessary to infect all participants as a function of the number of domains. The maximum is reached when the number of domains, and the number of participants per domain, are equal ($\sqrt{n}$). Although worst for performance, this balanced case is the best choice from a scalability point of view, minimizing the maximum number of participants per domain. In such a tree, the total

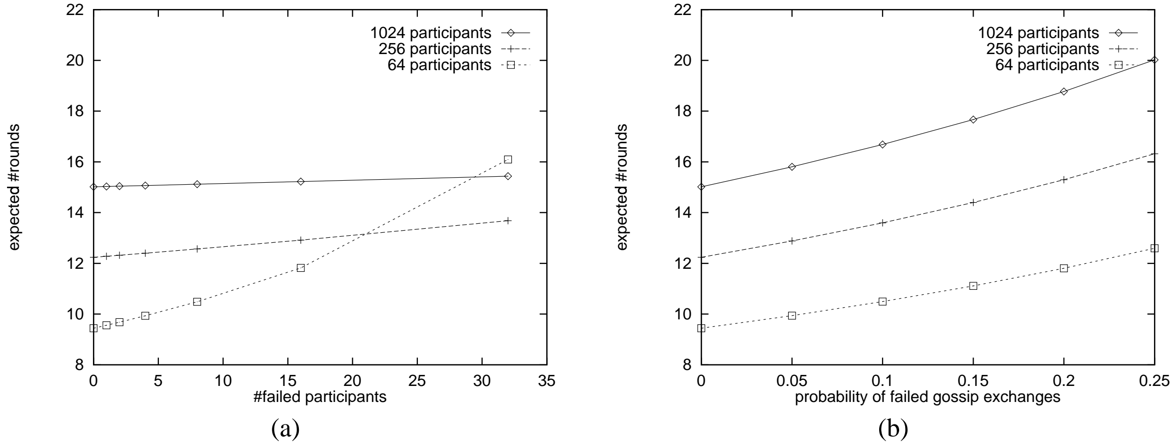


Figure 5: (a) The expected number of rounds necessary as a function of the number of failed participants; (b) The expected number of rounds necessary as a function of the number of failed gossip exchanges.

number of participating machines supported is m^l, where m is the number of participants in a domain, and l is the number of levels. With 4 levels, each having just 100 participants, the system will support 100,000,000 participating machines. Obviously, (non-hierarchical) "flat" gossip with 100,000,000 machines would not work. The size of the information stored at each machine, as well as the size of the gossip messages would be too large, and, even ignoring the size of the gossip messages, the load on the backbone network would be too high.

To understand the behavior of the slowdown due to hierarchy, we compared the ratio between the number of rounds necessary in a flat gossiping system with that of a balanced two-level hierarchy, with the number of participants ranging from 1 to 1024 participants. The result is shown in Figure 6(b). For a 1000-fold increase in membership, the slowdown due to the hierarchy is less than a factor of two. It appears that the slowdown grows no worse than logarithmically with the number of participants.

4.2. Membership

In a large distributed application, chances are that machines will be joining and leaving at a high rate. Just looking at crash rates, the combined rate goes up linearly with the number of machines. Keeping track of membership in a large distributed system is no easy matter [14]. In Captain Cook, membership is simpler than in other membership protocols because the granularity of information about membership degrades with distance in the HMIB tree. That is, participants know the membership in their local domain, but typically only how many members there are in other domains. (What membership information is available of other domains depends on their CFs.) In this subsection, we will first concentrate on how crashes are detected, and then on how new or recovering machines are joined into the navigation service.

The idea of failure detection in Captain Cook is indeed simple. A "Clock" column is added to each table in the HMIB. Participating machines update this value in their local MIB each time they send a gossip message. For internal tables, a CF computes the "Clock" value by taking the median time in the input table. (In the absence of Byzantine or malicious failures, the maximum or minimum clock value would work just as well.) When a participant notices that a clock value is slow by a predetermined value T_{fail}, it removes the row from the table it is in.

This algorithm will always detect and remove failed participants and empty domains within T_{fail} seconds, assuming that clocks are synchronized. However, setting T_{fail} aggressively will lead to mistaken failure detections, because an updated "Clock" value in the HMIB may not have fully propagated within T_{fail}. We can extend the analysis to calculate the probability that an infection has not propagated through the entire system after r rounds, such as done in [14]. Using this analysis, a number of rounds r can be chosen to get a suitable low probability of mistake, and from that T_{fail} can be calculated to be $r \times T_{gossip}$.

Either because of true network partitions, or because of setting T_{fail} too aggressively, it is possible that the HMIB splits up into two or more independent pieces. New and recovering machines that also form independent pieces. When such a machine wants to join the HMIB, it starts out with a degenerated tree consisting of a list of tables, each having one row. We need a way to glue the pieces together.

We first look into a situation in which a single new machine knows about an existing participant somewhere in the HMIB. It starts by sending a normal gossip message to that

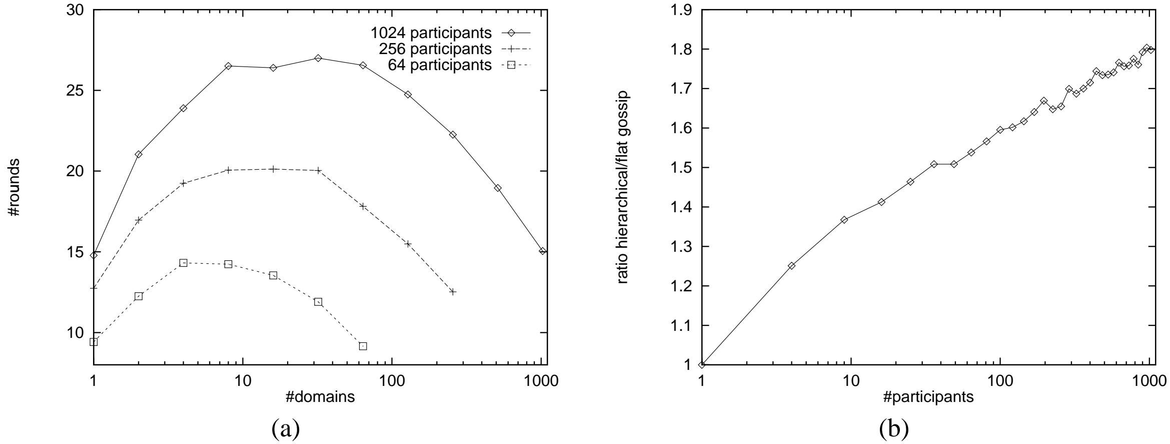


Figure 6: (a) The average number of rounds necessary to infect all participants as a function of the number of domains. The participants are spread evenly over all domains; (b) The ratio between the average number of rounds necessary to infect all participants using a two-level hierarchy, and the average number of rounds necessary in a one-level hierarchy.

machine. When a participant receives a gossip for a domain that it does not store itself, but it does know contacts for that domain (because the corresponding MIB ID is listed in one of its tables), it returns to the gossiper a *contacts* message with a list of the contacts for that MIB ID. These contacts are "closer" to the gossiper in the tree than the participant that received the gossip message. The gossiper, on receipt of a *contacts* message, tries again using one of the included contacts, and will eventually reach the closest participant in the tree.

For example, say that machine D in Figure 1 has not yet joined, and gossips initially to machine E. D includes in its gossip message the timestamps for its local versions of the tables for Domain1 and the Root domain. Upon receipt, E recognizes that it does not have the table for Domain1, but it does know contacts for Domain1 (because they are listed in the Root domain's table). E then returns those contacts to D.

Now we want to generalize this idea to the situation where we may have partitions of more than one machine. Partitions have no knowledge of the membership of other partitions. The strategy that we apply is to broadcast gossip messages. We call such gossip messages *beacon messages*. Care need be taken that not too many (or too few) of these messages are sent, or that too many participants respond. For this, each table in the HMIB has an "#Members" column that counts the total number of participants in the HMIB by summing the "#Members" columns in the child domains. Using the total number of participants, the rate of sending beacon messages and responses can be regulated.

The current implementation of Captain Cook sends two kinds of beacon messages: IP multicast messages sent to all participants, and IP broadcast message sent only on the local subnet. Multicast beacon messages are sent at a rate of approximately one beacon message per ten seconds. For this, every participant tosses a coin, weighted by the total number of participants, every ten seconds to decide whether to send a multicast or not. A broadcast beacon message is sent by a participant each time it hasn't received any message for a while. The "while" is chosen at random from an interval between 5 and 15 seconds. These broadcast beacon messages are particularly useful to participants that do not support IP multicast.

Upon receipt of a beacon message, a recipient tosses a coin, weighted by $\rho/\#Members$, to decide whether to send back an update or not. This way, the expected number of update messages back to the sender of the beacon message is ρ, typically chosen to be three or less.

4.3. Updating CFs

Internal MIBs cannot be updated directly, as they are calculated by their corresponding CF. However, Captain Cook *does* make it possible to replace CFs on-the-fly without having to recompile and restart the participating machines. Captain Cook's CFs are coded in a safe, Forth-like language [9]. The language does not allow arbitrary memory access, and also limits the amount of computation that may be done. Captain Cook has a built-in interpreter for this language.

The CFs may be stored in the HMIB itself, and this is the basis on which they can be updated. Each table has a "CF" column that contains (CF, timestamp) pairs. In leaf nodes, participants can place a new CF in their "CF" entry, and they use the most recent CF for the calculation of the parent MIB. The CF of an internal table is output by the CFs

of its children, and can only be replaced by updating those CFs.

5. Security

Gossip is interesting from the perspective of security. On the positive side, it is hard for an adversary to stop the flow of updates to provoke denial-of-service. On the negative side, it is easy to introduce and spread invalid updates of existing MIBs, or generate bogus MIBs. We wish to prevent this, yet maintain the scalability of the Captain Cook service and its immunity against denial-of-service. We are only concerned here with integrity, not confidentiality, of information.

The basic idea is to replace each MIB ID by a public key certificate. Each MIB is assigned a public/private key pair. The private key is given to all machines that store the MIB, and they use it to sign all updates of the MIB. A certificate for a MIB contains the MIB ID and its public key. The certificate in turn is signed by a certification authority (CA), which has a well-known public key. (Note that different HMIBs can use different CAs.)

On receipt of an update, a machine checks to see if the certificate is correctly signed by the CA, and the update is correctly signed according to the public key in the certificate. If not, the update is ignored.

Since the private key for a MIB is given only to the machines that store the MIB, only machines covered by that MIB can update it. This is a good start, since an adversary that compromises a machine in a tree with constant fan-out will only be able to influence $O(1/\log n)$ of the information stored in the HMIB. And assuming it cannot compromise the CA, the adversary cannot generate bogus MIBs.

However, since a compromised machine will have the private key for its MIB in the root domain, the adversary can update the corresponding row in the root table. This MIB is responsible for all the machines covered by that MIB. Worse yet, the private key for this MIB is replicated on all those machines, making it fairly easy for an adversary to obtain it.

There are at least two approaches to addressing this problem. The first is to use a threshold signature scheme [6], requiring a quorum of machines to sign MIBs. Although this may be the preferred solution in the long run, it requires some limited form of agreement, and we feel that the technology is not quite ripe for this approach.

The second approach, and the one we have currently taken, is to simply not give the private key of a MIB to all machines covered by the MIB, but only to a select few. All machines will still be able to update their own MIBs, and to gossip about updates of other MIBs, but only machines that have the private key of an internal MIB will be able to sign the output of the CF that was used to calculate that MIB. Disadvantages of this approach include reduced fault tolerance, increased managerial complexity, and slower propagation of updates.

In addition to using cryptographic techniques, it is a good idea to design CFs so that they are invulnerable to a small percentage of incorrect input. For example, rather than average load, it is often better to report median load, eliminating outliers potentially generated by malicious participants.

6. Related Work

Much work has been done in the area of scalable mapping of names of objects (often machines) onto location and other meta-information of the objects. The best known examples are DNS [8] and X.500 [12]. DNS maps a hierarchical machine name space onto so-called resource records. The collection of resource records basically forms a MIB, and therefore DNS is not unlike Captain Cook, be it that the set of resource records in DNS is fixed. X.500 is a more general directory service than DNS, but still focuses on machine and user names.

The Globe system [15] is an example of a service that maps arbitrary object names onto object identifiers, and then onto location. Globe also supports locating objects that move around. Other projects that provide mobile object location are Lighthouse Locate [10] and Awerbuch and Peleg's algorithm for locating cell phones [1].

Although an HMIB can be configured to map names onto meta-information, this is not the primary intention of Captain Cook. Captain Cook monitors resource information, summarizes it, and allows clients to zoom into areas of interest. In other words, Captain Cook maps resource information onto resources, rather than names onto resource information. Locating a mobile object giving its object identifier is not a service that Captain Cook provides (at least, not in a way that scales with the number of objects).

Particularly influential to the design of Captain Cook is the Clearinghouse directory service [5]. Clearinghouse was a competitor to DNS, used internally for the Xerox Corporate Internet. Like DNS, it maps hierarchical names onto meta-information. Unlike DNS, it does not centralize the authority of parts of the names space to any particular servers. Instead, the top two levels of the name space are fully replicated and kept eventually consistent using a gossip algorithm much like Captain Cook's. Other than Captain Cook, Clearinghouse does not apply condensation functions or hierarchical gossiping, and thus scalability is inherently limited. The amount of storage grows $O(n)$, while the total bandwidth taken up by gossip grows $O(n^2)$ (because the size of gossip messages grows linearly with the number of members). Clearinghouse has never been scaled to more than a few hundred servers. (Neither has Captain Cook

at this time, but analysis and simulation indicate that this should not present any problems.)

More recent work applies variants of the Clearinghouse protocol to data bases (*e.g.*, Bayou's anti-entropy protocol [11] and Golding's timestamped anti-entropy protocol [7]). These systems suffer from the same scalability problems, limiting scale to perhaps a few thousands of participants.

7. Conclusion

In this paper we presented Captain Cook, a scalable service that continuously monitors resources in the Internet. The obtained resource information is aggregated and condensed, and stored in a highly replicated tree. Effectively, the quality of locally available resource information degrades with distance, but users can "zoom in" to particular places of interest and locate resources based on the resource information. Updates in the tree are propagated through a novel hierarchical gossip protocol. Since there are no centralized servers — every machine plays a small part in the storage of the tree and the propagation of changes — Captain Cook is easily deployed. We have illustrated the potential use of Captain Cook in such areas as Cluster Management, Digital Libraries, and Pervasive Computing.

We have implemented Captain Cook, with the exception of security. For security, the updates will be signed and accompanied by a public key certificates of some well-known Certificate Authority that authorizes the signatures. We anticipate that much of the future work on Captain Cook will revolve around security.

One of the most complex issues in the deployment of Captain Cook is the design of the tree in which the monitoring information is to be stored, and the design of the condensation functions that generate the internal nodes of the tree. Although condensation functions can be dynamically changed and adapted to new uses, we anticipate that most distributed applications will choose to use a private instantiation of Captain Cook, if only for reasons of security. Another reason is that different applications might be interested in using different metrics of distance for degrading the quality of information.

Another problem to address is scaling in the amount of information per MIB. If MIBs are large (and they will be if they contain public key certificates), but updates to MIBs are small, then update messages will contain a lot of redundant information. We have designed a three-way handshake to reduce most of this redundancy. We will implement and analyze its effectiveness shortly.

Acknowledgments

This work benefitted tremendously from discussions with Yaron Minsky, Tim Clark, Ken Birman, Zhen Xiao, Oznur Ozkasap, Mark Hayden, Lili Qui, Dexter Kozen, Meng-Jang Lin, and Keith Marzullo. I would also like to thank Tim, Ken, and Yaron for comments on this paper.

References

[1] B. Awerbuch and D. Peleg. Online tracking of mobile users. *J. ACM*, 42(5):1021–1058, Sept. 1995.

[2] N. Bailey. *The Mathematical Theory of Infectious Diseases and its Applications (second edition)*. Hafner Press, 1975.

[3] K. Birman, M. Hayden, O. Ozkasap, M. Budiu, and Y. Minsky. Bimodal multicast. Technical Report 98-1665, Cornell University, Dept. of Computer Science, Jan. 1998.

[4] J. Davis and C. Lagoze. The Networked Computer Science Technical Report Library. Technical Report TR94-1595, Department of Computer Science, Cornell University, July 1996.

[5] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 1–12, Vancouver, British Columbia, Aug. 1987. ACM SIGOPS-SIGACT.

[6] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91 Proceedings*, volume 576 of *Lecture Notes on Computer Science*, pages 457–469. Springer-Verlag, Aug. 1992.

[7] R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, Fall 1992.

[8] P. Mockapetris. The Domain Name System. In *Proc. of IFIP 6.5 International Symposium on Computer Messaging*, May 1984.

[9] C. Moore. Forth: a new way to program a mini-computer. *Astronomy and Astrophysics Supplement*, 15:497–511, 1974.

[10] S. Mullender and P. Vitányi. Distributed match-making. *Algorithmica*, 3:367–391, 1988.

[11] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the Sixteenth ACM Symp. on Operating Systems Principles*, pages 288–301, Saint-Malo, France, Oct. 1997.

[12] S. Radicati. *X.500 Directory Services: Technology and Deployment*. International Thomson Computer Press, 1994.

[13] W. Stallings. *SNMP, SNMPv2, and CMIP*. Addison-Wesley, 1993.

[14] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. of Middleware'98*, pages 55–70. IFIP, Sept. 1998.

[15] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, pages 104–109, Jan. 1998.