Porcupine: A Highly Available Cluster-based Mail Service

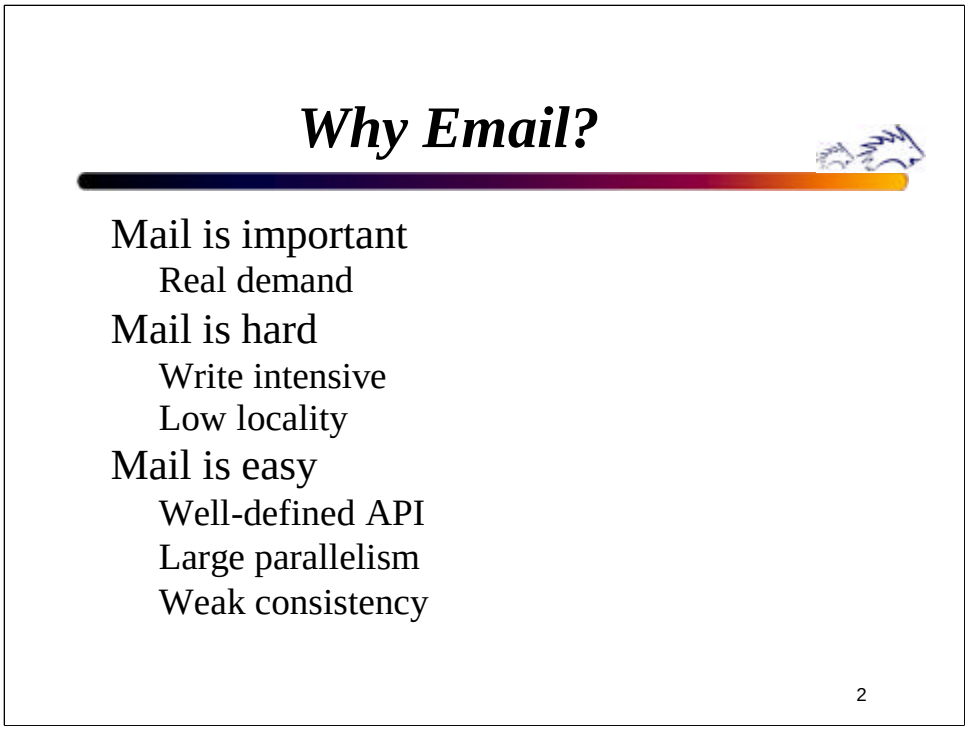
Yasushi Saito
Brian Bershad
Hank Levy

http://porcupine.cs.washington.edu/

University of Washington
Department of Computer Science and Engineering,
Seattle, WA

1

0:10



I first want to motivate this talk by discussing why we chose email as a research target. There are several reasons. First is that mail is important. The use of large scale email service are rapidly expanding these days, and we see a real demand for systems like porcupine..

The next reason that mail is a difficult problem, especially compared to a typical web service, which has been studied very extensively. Unlike simple web, mail is a type of application that is more write intensive, is disk-bound, and has lower access locality. These properties make it difficult to use techniques that are effective in scaling we services, such as proxying.

On the other hand, mail turns out to be substantially easier than distribute file systems or data base services, which are traditional solutions to write-intensive service,. First, mail has a well defined and narrow API, which is a message send and receive rather than a generic byte read/write. Also, the mail workload has large inherent parallism, so we can scale the system without worrying much about access conflicts. Finally, consistency requirement for mail is fairly weak. We experience delay in mail delivery or duplicate messages all so often. To summarize, the service requirements for mail sits somewhere between web and database, and we believed we could come up with a better architecture by not binding ourselves to either web or database.

1:35

Goals

Use commodity hardware to build a large, scalable mail service

Three facets of scalability ...

- *Performance*: Linear increase with cluster size
- *Manageability*: React to changes automatically
- *Availability*: Survive failures gracefully

3

The goal of the P project is to build a very large scale email service using just a cheap commodity hardware. When we say large, today's large mail sites, such as AOL or hotmail handles about 100 million messages per day, and given that Internet is still growing rapidly, we can expect an order of magnitude growth in the scale of email servers. That's the area we target, about 1 billion messages per day.

The key word in our project is to "scale", and we define the work along three axes. First, of course, we need the system performance to scale. Not only that, we need to scale in terms of managing. We expect the mail traffic to constantly grow, and therefore the mail server needs to grow continuously as well. We don't want human to be forced to manage such system changes manually -- it quickly becomes unmanageable as the system heterogeneity grows. Therefore, we want the system itself to react to such changes. Finally, our decision to use commodity hardware makes the system more prone to component failures. Thus, we want the system to survive gracefully various types of failures, including multiple node failures and network partitions.

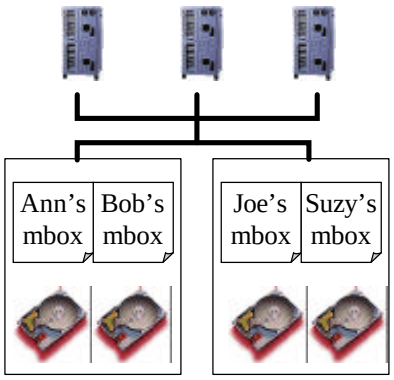
Conventional Mail Solution

Static partitioning

Performance problems:
- No dynamic load balancing

Manageability problems*:*
- Manual data partition decision

Availability problems:
- Limited fault tolerance



4

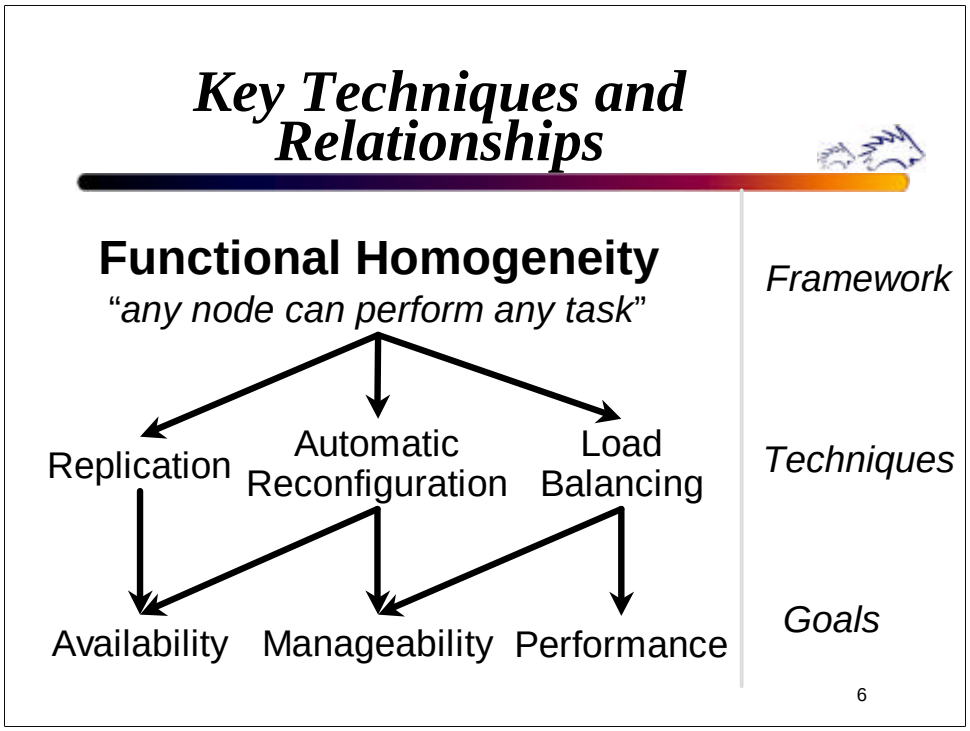
The existing servers are implemented by using a number nodes that run distributed file systems or database servers, and statically assigns each user to one of these nodes. We are not satisfied at this conventional solution because it doesn't scale in this three areas. For example, it has performance scaling problem because it cannot load balance to react to changes in traffic or configuration. It is difficult to manage since the administrator needs to know the speed or the disk capacity of each node and manually assign each user to a node. In addition, this architecture it provides little fault tolerance. If a file server goes down, all the users who have mailboxes on the server are locked out of the system.. 1:30

Presentation Outline

- Overview
- Porcupine Architecture
  - Key concepts and techniques
  - Basic operations and data structures
  - Advantages
- Challenges and solutions
- Conclusion

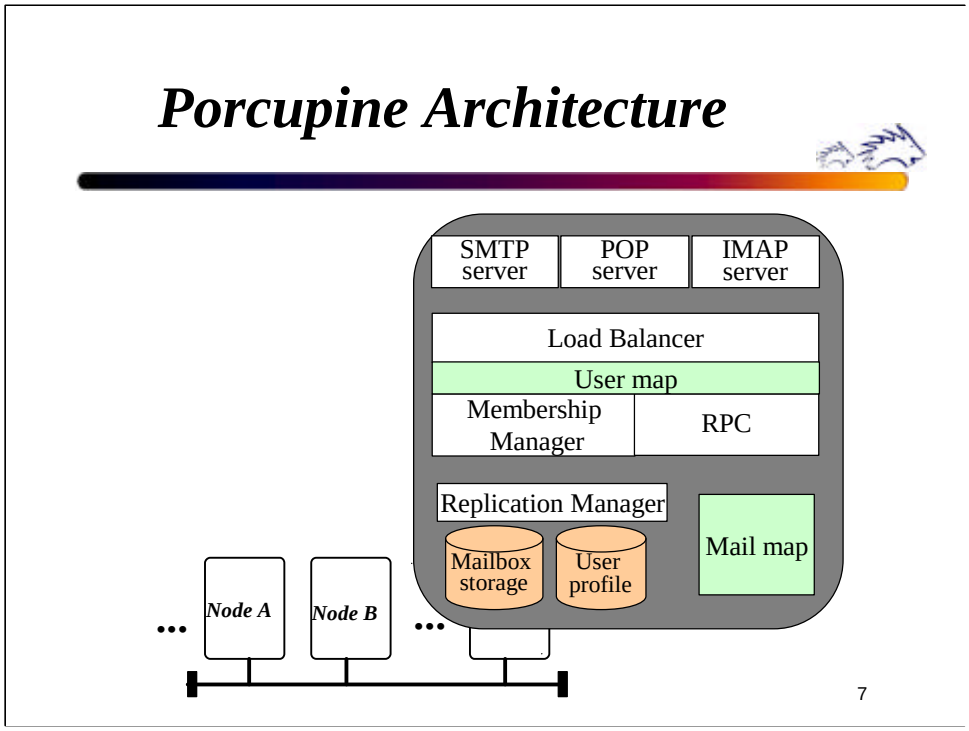
5

In the next few slides, I'll overview the architecture of Porcupine. First, I summarize key techniques used in P and how it solve our goals. Next, I explain how it works at high level, and then I discuss why we think P is better than the conventional solution. 0:30



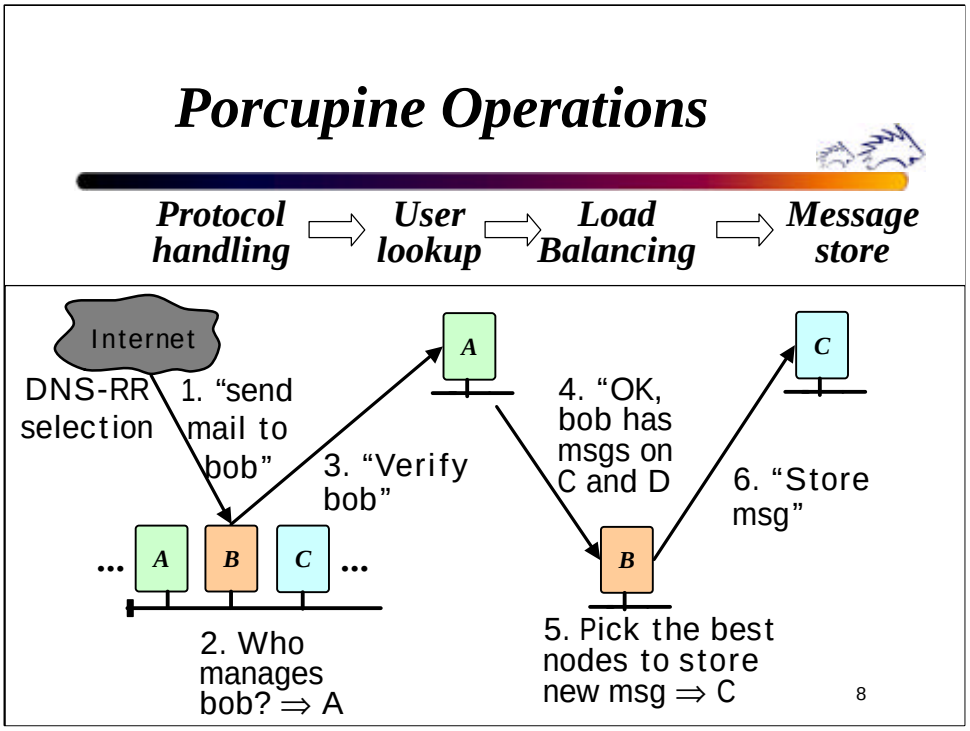
The most significant difference between P and conventional servers is that P is FH. This means that any node can perform any task, either being interaction with mail clients or mail storage, and any piece of data can potentially be managed by any node. Especially, mail messages for a single user can be scattered on multiple nodes, and they are collected only when the user request reading them. All the data and task placement decisions are made dynamically.

We have developed several techniques that takes advantage of FH. Automatic reconfiguration and replication keeps P functional after various kinds of failures. Dynamic load balancing ensures each node stores email message as their processing capability allows, and it masks skew in both workload and cluster configuration. These techniques in turn achieves the set of scalability goals I described in the earlier slide.



The picture shows the main components of the system. Porcupine consists of a set of nodes connected by a fast network. Because Porcupine is functionally homogeneous, all the nodes run the identical set of software.

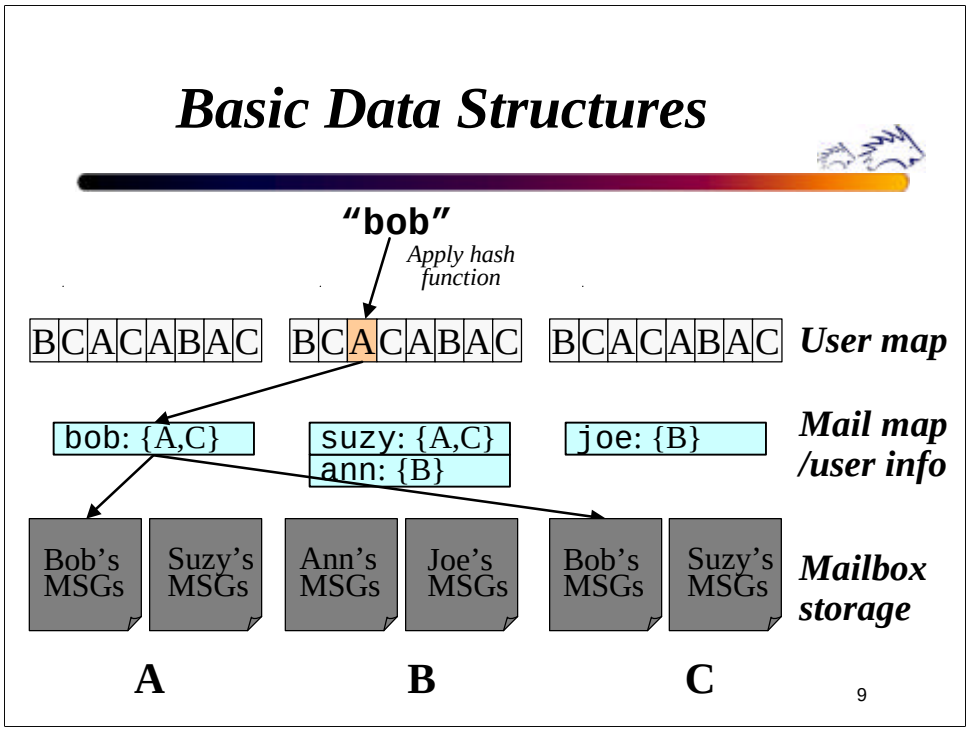
The close up figure in the left shows the structure of one node. Each node runs internet protocols such as SMTP and IMAP. In addition, it contains backend include mailbox storage, user database. The mailmap is to track the location of mail messages, and the stuff in middle is used to dispatch tasks among nodes. I'll explain how they are used in later slides.



Now, I want to show a birds-eye view of how P works. This picture shows the steps performed during mail delivery. A client somewhere in the Inet chooses a node to contact using some naming mechanism such as DNSRR. Because of functional homogeneity, the client can in fact pick any node, but in this example, lets suppose it picked B. The client lets B know that it wants to send a msg to bob.

The node B first needs to check who bob is. B first learns where the Bob's user information is managed, say, A in this example. B asks A about bob, and A replies that bob is a valid user and that he has mail msgs on C and D at this point. Next, B makes a load balancing decision to determine where to store this new message. Lets say C is picked . B asks C to store the new message, and the session is complete. In step 4, B can also pick a node other than C or D, say, E for load balancing purpose. In such case, E needs to let A know that E has now stored a message for bob.

This picture reveals several data structures that needs to be managed. First, each user must be managed by some node, and all the nodes must agree on who is managed where. Also, the manager of each node must keep track of there the user's messages are.



The data structures I hinted in the previous slide are actually shown here. In P, the information about a single user is managed by a single node. The entire user population is partitioned into a equivalence class using a hash function on user names. Therefore, all the users whose names have the same hash value are managed by a single node. The small in-memory table called user map is replicated on each node and it maps hash value into the node that manages the users in the class. The mgr of each user keeps track of which set of nodes store msgs for the user. We call this information mailmap. This picture shows how Bob can read his messages on node B. First, a hash function is applied to the string "bob" and user map is looked up, and the node B learns that A is responsible for managing bob. B does an RPC to the node A and get bob's mailmap, which contains the set {A, C}. Subsequently, B does another round of A and C to obtain bob's messages.

Porcupine Advantages

Advantages:

- Optimal resource utilization
- Automatic reconfiguration and task re-distribution upon node failure/recovery
- Fine-grain load balancing

Results:

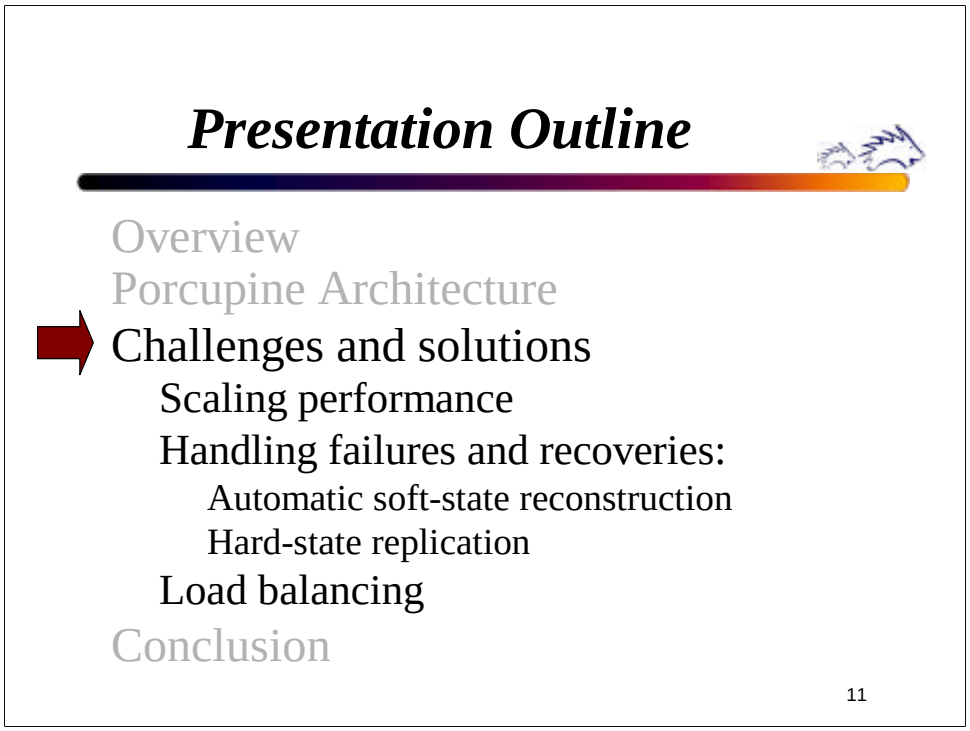
- Better Availability
- Better Manageability
- Better Performance

10

The functional homogeneous architecture of P yields many potential benefits. First, because all the data/task placement decisions are dynamic, the system can make use of node resources optimally without forcing human to decide how. Also, the system is able to react to configuration changes, such as node failures automatically by re-partitioning the tasks and data among remaining active nodes. Finally, the system can be faster than the conventional solution because it can do fine-grain load balancing during data placement.

All these advantages combine into a system that is more available, more manageable and faster than conventional systems.

0:45



We had several problems we needed to solve to realize these potential advantages. I'll discuss three of them and how we solved. First is how we scale the system performance. Next is how we actually react to failures, and third is how we react to skew in workload or system configuration using load balancing mechanism. Along the discussion of these issues, I'll mix the performance graphs of our prototype implementation to demonstrate the effectiveness of our solutions.

0:30

Performance

Goals

- Scale performance linearly with cluster size

Strategy: Avoid creating hot spots

- Partition data uniformly among nodes
- Fine-grain data partition

12

Now our first issue is how we scale performance linearly with the cluster size, and the strategy is to partition data management responsibility uniformly among nodes to avoid creating hot spots, and to make the partition very fine grain to further reduce the chance of creating hot spots.

I actually already described the data structures such as user map and mail map that achieves these goals, so I don't repeat them. I'll just show you a graph tells how our design scales in practice.

Measurement Environment

30 node cluster of not-quite-all-identical PCs
- 100Mb/s Ethernet + 1Gb/s hubs
- Linux 2.2.7
- 42,000 lines of C++ code

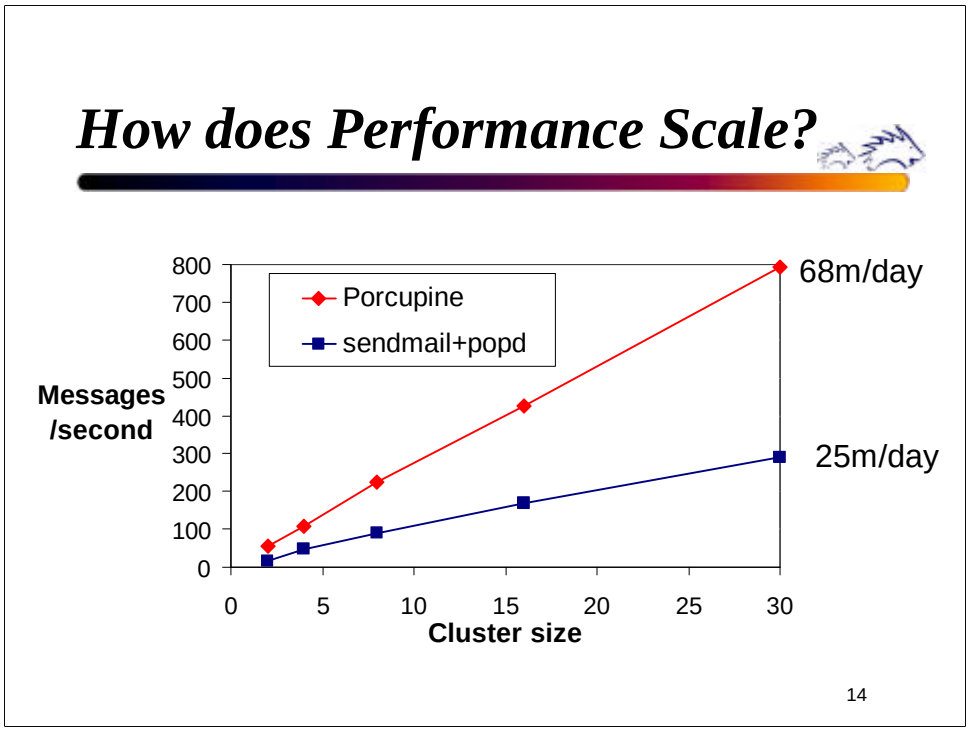
Synthetic load

Compare to sendmail+popd

13

We show the baseline performance of Porcupine to show how the system scales. We measured P a cluster consisting of 30 nodes because that's only what we had. The PC consists of a mix of various types of CPUs and disks. Traffic shape is based on that of the mail server in our department.

We also compare P to sendmail to show that we are no slower than a popular mail server despite all the additional benefits P provides. I compare ourselves against sendmail, the most popular mail server software in use today.



X axis is the number of nodes in the cluster, and Y axis is the maximum number of mail messages a cluster can handle per second. Our workload runs both senders and readers concurrently, and the readers are designed to delete messages after they are read. So, the Y numbers really represent the number of mail messages that a cluster can receive, read, and delete per second. The most important point of this graph is that Porcupine cluster scales completely linearly. This shows we achieved our goal of performance scalability although up to 30 nodes. Next, porcupine is no slower than sendmail, despite the fact that P provides all the management and availability benefits that don't exist in sendmail. In fact, P is more than twice as fast as sendmail. Sendmail is slow just because it uses the file system less efficiently than porcupine .

1:20

Availability

Goals:

- Maintain function after failures
- React quickly to changes regardless of cluster size
- Graceful performance degradation / improvement

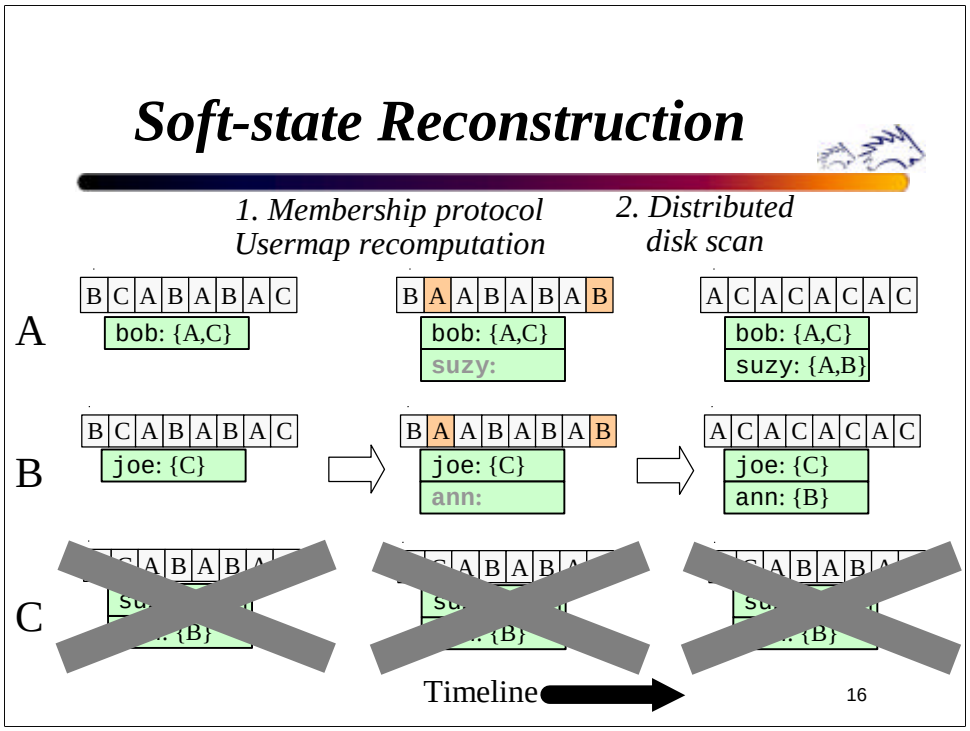
Strategy: Two complementary mechanisms

- *Hard state:* email messages, user profile
  - ⇒ Optimistic fine-grain replication
- *Soft state*: user map, mail map
  - ⇒ Reconstruction after membership change

15

Now I switch topic and talk about how P deals with node failures and additions. Our first goal is to maintain system function after failures, and that to react changes quickly regardless of cluster size. We also want the system performance to degrade or improve quickly. To achieve these goals, we use two mechanisms, depending on the nature of data. On one hand, there are types of data that users really care and must be stored on disk. We call such data hard state, and that includes email and profile. Such data are replicated on multiple nodes. Other types of data are derived from another source, most often from hard state. We call them soft state, and that includes user map and mail map. Instead of replicating, we reconstruct them from source on demand and P runs a protocol to minimize that effort.. In the following slides, I first show hot soft state reconstruction works, and then explain optimistic replication briefly.

1:15

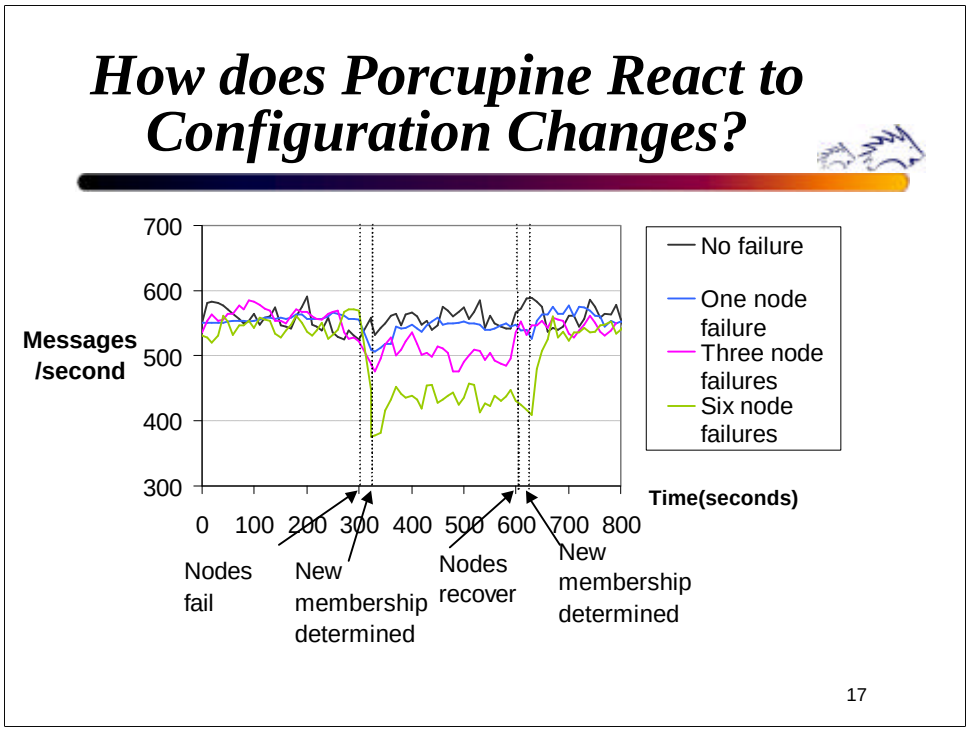


Soft state reconstruction can best be explained using pictures. This picture shows a three node cluster. The upper boxes in each node represents the user map that shows which user is managed by which node. The green boxes below are actual user information. Thus, this three-node cluster manages 4 users.

Now suppose C crashed. Here, we need to move all the users managed by C, suzy and ann, to either A or B. This is done in two steps. First, one of the remaining nodes initiates the membership protocol and determines who's alive and who's dead, and at the same time, the user map is reconfigured by removing nodes that are found to be dead adding nodes that found to be recovered. The new membership and the new user map are broadcast to the nodes and we reach this middle stage.

Here, since the user map has changed, the users managed by C are moved to A and B, but at this stage, their mail maps are empty. Thus, in the next step, each node scans its local disk and finds out all the messages that belong to these moved users.

The second step looks very expensive, but it really isn't because each node needs to scan only a portion of disk that corresponds to user map entries that have moved. In fact, total cost of recovery does not depend on the cluster size. This is because the total amount of mail map information that needs to be recovered from disks is equal to those stored on the crashed node, and that does not depend on the cluster size. 1:40



In this picture, we show how Porcupine reacts to failures in practice by artificially crashing and recovering nodes. The X axis is the event timeline, the unit is seconds, and Y axis is the system performance. We ran our 30-node cluster into a steady state, and then artificially crashed 1, 3, or six nodes at time 300 seconds, and later, at time 600 seconds, we recovered all the crashed nodes. First, please take a look at the throughput during failures. The level of performance decrease is about proportional to the number of nodes failed, and also the performance restores to the original level after the nodes recover. This means P achives its goal of graceful performance degradation and improvement. Also, please notice how quickly P is able to reach the steady state after either crash or recovery. This shows our failure recovery mechanism is working quickly and efficiently.

1:00

Hard-state Replication

Goals:

- Keep serving hard state after failures
- Handle unusual failure modes

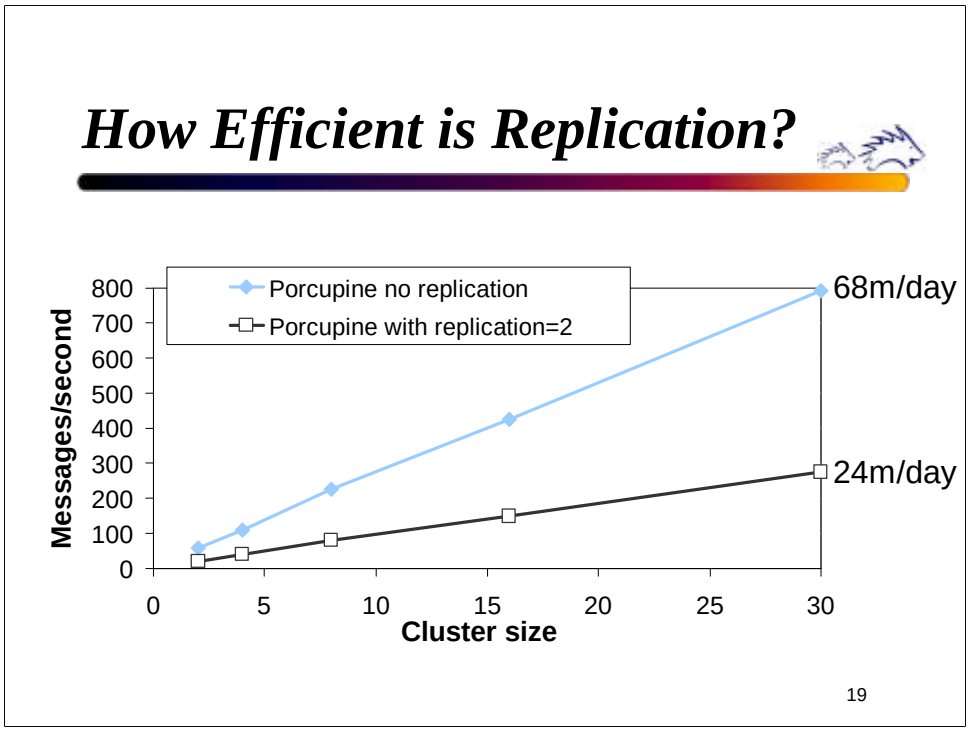
Strategy: Exploit Internet semantics

- Optimistic, eventually consistent replication
- Per-message, per-user-profile replication
- Efficient during normal operation
- Small window of inconsistency

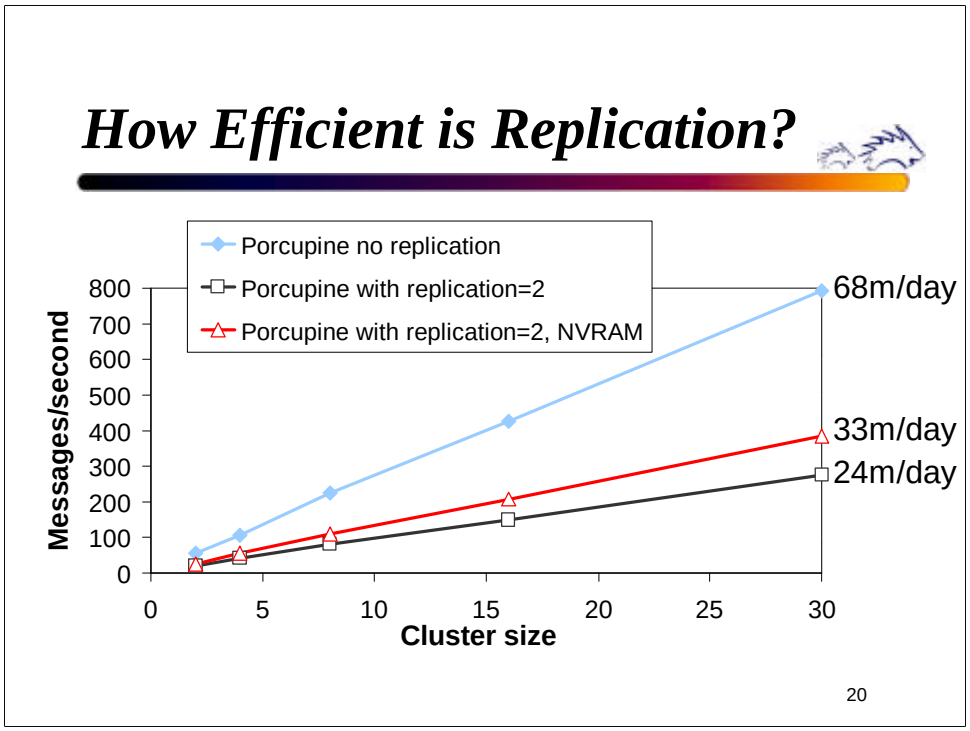
18

Now I want to explain briefly how keep messages and user profile available after crash. Our goals here are to make the hard state available, of course, and because we are using cheap hardware, we also need to handle unusual failures like multiple node failures, network partitions or long-lasting node failures. Finally, to make the service non-blocking even when that means presenting user an stale data..

The scheme we came up with is a form of optimistic replication that only ensures eventual consistency. Optimistic replication satisfies all our requirements nicely. The basic algorithm works by letting a coordinator, which is usually the node that initiated the update, push updates to other replicas. The crash of replicas other than the coordinator is handled by the coordinator wait until the replica recover. The coordinator crash is handled by letting one of other replicas quickly becoming the coordinator and complete the propagation. Thus, the common case operation is simple and very efficient. In case of failures, the protocol ensures that replicas will quickly converge into the newest value without every blocking access to clients.



I show evaluation of our replication mechanism. Here, X axis is the number of nodes in the cluster, and Y axis is the performance. The blue line shows the baseline P performance without replication, and lower black line shows what happens when each msg is replicated on two nones. First, please see that the replicated P still scales perfectly. However, its performance is lower than what we expect because it really should be half of the baseline. This is because of the disk logging overhead that happens inside the replication engine. This overhead can be eliminated by using a fast, separate log disk or using NVRAM as a file buffer.



So we pretended that we have such hardware and simply removed synchronous disk flushing from the disk logging routines. We get this red line. This line still scales perfectly, and in addition, its performance is exactly what we expected, the half of the baseline case. So, to summarize, this graph shows that our replication algorithm is scalable, and is very efficient.

Load balancing: Deciding where to store messages

Goals:

- Handle skewed workload well
- Support hardware heterogeneity
- No voodoo parameter tuning

Strategy: Spread-based load balancing

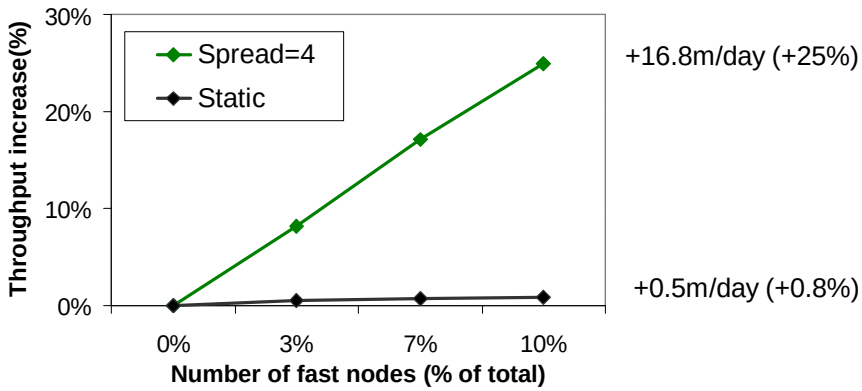
- *Spread:* soft limit on # of nodes per mailbox
  - Large spread ⇒ better load balance
  - Small spread ⇒ better affinity
- Load balanced within spread
- Use # of pending I/O requests as the load measure

21

Our final topic is how we load balance, or how we decide data placement. Our goals are is to handle workload skew, such as a few users receiving gigantic amount of mail for a short period of time, and also to handle configuration skew, such as a few nodes are much faster than the remaining.

The algorithm we came up with is what we call spread-based load balancing. Spread is the limit on the number of nodes per single user's mailbox. The limit is soft is a sense that it can be violated when nodes crash. Spread represents a single parameter that administrator needs to tune. When the spread is large, the system has more nodes to choose from, thus we can balance load better. If the spread is small, the system needs to access fewer disks per mailbox, thus we can more streamline the disk head movement and also reduce the size of the mail map. P chooses node to store messages from the spread, using the number of pending I/O requests as the measure of a node's load.

How Well does Porcupine Support Heterogeneous Clusters?



22

Before explaining this graph, we also did experiments on how the system reacts to workload skew, but I won't present them in this talk. Please refer to the paper if you are interested. This graph shows how P reacts to heterogeneity in cluster. The X axis is the heterogeneity. 0 is the baseline and all the nodes run at approximately at the same speed. 3,7,or10 means these percentage of nodes have very fast disks, about three times faster than the rest. The Y axis is the relative performance improvement over the baseline case.

got the cluster find special resources.

Conclusions

Fast, available, and manageable clusters can be built for write-intensive service

Key ideas can be extended beyond mail

- Functional homogeneity
- Automatic reconfiguration
- Replication
- Load balancing

23

Ongoing Work

- More efficient membership protocol
- Extending Porcupine beyond mail: Usenet, BBS, Calendar, etc
- More generic replication mechanism

24

Also we are looking a little more ahead and trying to apply the ideas to services that demand stronger data consistency, such as auctioning. Especially, we are looking at more generic replication algorithm.