Paper I Summary

Yima: A Second-Generation Continuous Media Server

Overview

Yima is a continuous media (CM) server. It is a research prototype, not a commercial product. It is designed to address challenges in real-time streaming for applications like news on demand, e-commerce, and scientific visualization. CM data requires managing real-time data constraints and the architecture must address the large size of CM objects.

There are two currently available commercial implementations of CM servers.

- Low-cost, single-node, consumer-oriented systems serving a limited number of users.
    - RealNetworks, Apple Computer, and Microsoft product
- Multinode, carrier-class systems such as high- end broadcasting and dedicated video-on- demand systems
    - SeaChange and nCube

Yima is a 2nd-generation CM server. It improves on earlier prototypes. The main characteristics of Yima are the following:

- complete distribution with all nodes running identical software
- no single points of failure;
- efficient online scalability allowing disks to be added or removed without interrupting CM streams
- synchronization of several streams of audio, video, or both within one frame (1/30 second)
- independence from media types
- compliance with industry standards
- selective retransmission protocol
- multi threshold buffering flow-control mechanism to support variable bit-rate (VBR) media

Yima is also a complete end-to-end system that uses an IP network with several supportable client types, ensuring scalability, fault tolerance, and media synchronization across multiple nodes. It supports multiple types of media, including video, audio, and panoramic media, with a focus on variable bit-rate (VBR) streaming.

1. SYSTEM ARCHITECTURE

Yima consists of multiple cluster nodes connected via a high-speed IP network. This keeps the per-port equipment cost low and makes the system compatible with the public Internet. The nodes in the cluster communicate and send the media data via multiple connections. Each node is connected to a local switch

and the local switch connects to both a WAN backbone for serving distant clients and a LAN environment for local clients. Yima recognizes different media types through the client software installed. These installed softwares lets various software and hardware decoders be plugged in.

2. Server Design Challenges

CM servers must store data efficiently and schedule data retrieval and delivery precisely before transmission.

- Yima-1: A master-slave design where a master node coordinates all streams.
  - Easy to manage but suffers from bottlenecks and single points of failure.
- Yima-2: A bipartite model that decentralizes scheduling and delivery functions across nodes.
  - Each node delivers local data directly to clients, reducing internode traffic.

2.1. Data placement and scheduling

There are two ways of storing data blocks into disk drives, a round robin placement and random placement. The round robin placement uses a cycle-based approach to provide efficient resource scheduling so that enables high throughput for efficient bandwidth for video objects that are retrieved sequentially.

The random deadline driven approach does not support as much as optimizations the other does. Its throughput may be lower, but it provides several benefits. It supports multiple delivery rates, simplifies scheduler design, interactive applications and automatically achieves the average transfer rate with multi zoned disks. It also recognizes data more efficiently when the system scales up or down. However, it requires a large amount of metadata to store and manage each block's location in a centralized repository. Yima uses a pseudorandom block placement to avoid this overhead. With this policy, Yima can recompute the block locations on the disk. Thus, instead of storing locations for every block, Yima does store only the seed for each file.

2.2. Scalability, heterogeneity, and fault resilience

Any CM server design must be scalable when the system grows or requirements change. CM servers use multi disk arrays to meet this requirement. Also Yima adapted modular architecture to support scalability and availability. Servers are connected via a high speed network fabric that can expand as demand increases.

Applications with CM servers require continuous operation. Yima uses disk merging to implement a parity based data redundancy scheme to provide fault tolerance and to take advantage of a heterogeneous storage subsystem. Disk merging unifies multiple physical disks into logical disks. This simplifies application management by providing a uniform characteristic and consistent view of storage, enabling the use of standard scheduling and data placement algorithms.

2.3. Data reorganization

Computer clusters distribute data across nodes uniformly. Round-robin and random placement strategies achieve even distribution. However, adding or removing nodes requires data redistribution. Random placement has much less overhead for this, as it involves moving fewer blocks compared to round-robin, preserving load balance.

Yima uses a pseudorandom number generator to produce a random number sequence to determine block locations. This number is needed because Yima cannot use the previous number sequence to find the blocks when the system scales up and new disks are added. To support disk scaling, Yima employs the Scaddar algorithm. This algorithm generates a new pseudorandom number sequence, minimizing block movements and computational overhead in the computation of the new locations. Scaddar enables efficient disk scaling without disrupting Yima's services..

2.3. MultiNode server architecture

Yima servers are built from clusters of server PCs called nodes. A distributed file system provides a complete view of all data across nodes without requiring data replication, but required for fault tolerance. Yima clusters can operate in either master-slave or bipartite mode.

2.3.1. Master-slave design (Yima-1)

Yima-1 employs a master-slave architecture, where the master server handles communication with client servers. Slave servers, on the other hand, communicate with the master and solely execute the distributed file system service. Yima-1 software consists of the distributed file system and media streaming services. The file system consists of multiple file I/O. The media streaming server consists of a scheduler, a real time streaming protocol (RTSP) and a real time protocol (RTP).

Master runs both the file system and media service system while slaves run only file I/O modules. The system utilizes a pause-resume flow control mechanism for VBR media delivery. It is simple but can lead to potential bursty traffic. Additionally, the master node serves as a single point of failure, and heavy internode traffic exists between master and slave servers, which is communications between master and slaves.

2.3.2. Bipartite design (Yima-2)

Yima-2 addresses Yima-1's limitations by adopting a bipartite model, separating nodes into server and client groups. This provides better performance and more scalability. Yima-2 only keeps the RTSP module centralized and uses a decentralized approach that empowers each server node to directly retrieve, schedule, and transmit data to the requesting clients, eliminating the single-point-of-failure bottleneck of Yima-1's master node.
While this enhances performance and scalability, it introduces complexities such as client-side handling of data from multiple nodes, modified scheduling to satisfy the decentralized architecture of YIma-2, and a new flow-control mechanism to prevent client buffer overflow or starvation. Each client maintains contact with one RTSP module, but the server node can run an RTSP module. To ensure smooth data delivery and avoid bursty traffic, Yima-2 employs a feedback-based flow control system, allowing clients to adjust transmission rates by sending slowdown or speed up commands. Through this clients can receive a smooth data flow by monitoring the amount of data in its buffer.

3. CLIENT SYSTEM

Yima Presentation Player acts as a client application. This client receives streams via standard RTSP and RTP communications. Yima's player supports multiple media formats and runs on Linux and Windows.

3.1. Bipartite design (Yima-2)

The application uses a circular buffer to reassemble packets received from servers, while a decoding thread consumes data from the same buffer. To manage buffer overflow or underflow, the application implements two data flow control mechanisms, pause-resume and client-controlled delta ∆p.

3.1.1. Pause-resume

When the buffer reaches a maximum threshold, the pause-resume control flow mechanism pauses streaming and allows the consumer thread to decode media. Streaming resumes when the buffer level drops to a minimum threshold and it ensures smooth playback.

3.1.2 Client-controlled ∆p

In this mechanism, schedulers synchronize time across nodes using NTP. Server nodes send packets sequentially at timed intervals using a common time reference and each packet's timestamps. The client can fine tune the inter packet delivery time (∆p) based on buffer data levels. Thus, clients can control the delivery rate of received data. It is smoothing fluctuations in network traffic or server load, this mechanism ensures consistent data delivery and prevents bursty traffic.

3.2. Player media types

The Yima player supports various media types. It uses a three-threaded structure for playback. The playback thread interfaces with actual, software or hardware based media decoders.

- Hardware decoders: MPEG-1 and MPEG-2 video with two-channel audio (CineCast) and 5.1-channel Dolby Digital audio (Dxr2 PCI card). The Dxr2 also provides SP-DIF digital audio output.
- Software decoders: MPEG-4 (DivX), offering higher compression than MPEG-2. It reduces file size significantly while enabling near NTSC-quality streaming over ADSL.

3.3 HDTV client

streaming of high-definition content require high-transmission bandwidth. The authors report that this was accounted for on the server side by Yima, but more intriguing problems arouse on the client side, specifically achieving the required real-time frame rates. That was due to the use of a cost-effective software decoder. In a more recent implementation, the authors used a hardware decoder to fix the problem.

Streaming high-definition content requires high-transmission bandwidth. While this was managed on the server side by Yima, more intriguing challenges arose on the client side, achieving real-time frame rates. It originated from the use of a cost effective software decoder. In a recent implementation, with a Vela Research CineCast HD hardware decoder, it achieves real time frame rates up to 45 Mbps.

3.3 Multi Stream synchronization

Yima supports the synchronization of multiple independently stored media streams such as panoramic, five-channel video and 10.2- channel audio. The video channels are compressed and sent to the client while the audio channels remain uncompressed and are sent separately. Using time instance and temporal causality, all video and audio channels are synchronized at the client side and combined accurately. Yima achieves precise playback with three levels of synchronization: block-level via retrieval scheduling,

coarse-grained via the flow-control protocol, and fine-grained through hardware support. To ensure that all streams start at exactly the same time, the decoders use an external trigger to accurately initiate playback through software.

Yima client server communications protocol enables the synchronization of multiple independently stored media streams. It supports panoramic, five-channel video and 10.2-channel audio. The five video channels are encoded as MPEG-2 streams, while the uncompressed audio channels are sent separately to the client. During playback, all streams are tightly synchronized to combine five video frames into a panoramic 3,600 x 480-pixel mosaic every 1/30th of a second, with 10.2-channel surround audio synchronized to the video.Yima achieves precise playback through three levels of synchronization.

- block-level via retrieval scheduling
- coarse-grained via the flow-control protocol
- fine-grained through hardware support

The flow-control protocol maintains the same amount of data in client buffers, enabling the use of CineCast decoders and a genlock timing-signal-generator device to produce frame-accurate output. The CineCast decoders use an external trigger initiating playback through software.

4. RTP/UDP AND SELECTIVE RETRANSMISSION

Yima uses standard RTP to deliver time-sensitive data, which relies on the best-effort UDP protocol. As a result, packets may arrive out of order or be lost during transmission. To address this, Yima implements a selective retransmission protocol that attempts one retransmission of a lost RTP packet, but only if the packet can arrive in time for consumption. In Yima-2's multiple server delivery architecture, a unicast approach is used for retransmission. The client unicasts the request to the specific server node processing the requested packet. The client determines the source node for retransmission requests by identifying missing node specific sequence numbers in the received packets. To support this, each packet in Yima-2 contains both a global sequence number and a node-specific sequence number.

5. TEST RESULTS

Yima-2 demonstrates a linear scalability in the number of streams as nodes increase. Yima-2 scales linearly with the number of nodes and achieves better throughput than Yima-1. Yima2’s performance may become sublinear with larger configurations, low bit rate streams or both. However, Yima-2 scales much better compared to Yima-1. It proved the perfect scalability of the architecture and the incorporated retransmission protocol shows its effectiveness with the test results. Yima’s scalability was demonstrated through experiments involving HD streaming across long distances. Yima demonstrates the feasibility of using commodity hardware for large-scale CM services.

Paper II Summary

Manageability, Availability, and Performance in Porcupine: A Highly Scalable Cluster-based Mail Service

Overview

Porcupine is a scalable, cluster-based mail server designed to handle billions of email messages daily using commodity PCs. It provides a highly available and scalable electronic mail service. The system focuses on scalability with three key aspects: Manageability, Availability, and Performance.

1. Introduction

Porcupine uses a cluster of many small machines to enable them to work together in an efficient manner to meet the growth of highly available services.

1.1 System requirements

Porcupine meets these manageability, availability, and performance.

1. **Manageability requirements**: It should be easy to manage although a system is physically large, self-configured in loading and data distribution and self-healed with respect to failure and recovery. The system should be easy to add and replace more machines or disks and can perform at different capacities under the control of the system
2. **Availability requirements**: A part of nodes can be down at any given time and the system should deliver good service to all of its users at all times. Whole groups of users should be able to avoid failure mode despite some component failures
3. **Performance requirements**: Porcupine's performance linearly increases with the number of nodes in the system and its single node performance should be competitive with other single node systems. Porcupine targets a system that scales to hundreds of machines and can handle a few billion mail messages per day with today's commodity PCs.

Porcupine also should be functionally homogeneous for each node, dynamically schedule transactions uniformly, be automatically reconfigured whenever nodes are added or removed and replicate user data automatically for availability.

Porcupine runs on a cluster of thirty PCs connected by a high-speed network and can be scaled out beyond that. Performance is linear with respect to the number of nodes. The system automatically configures its workload, node capacity, and node availability. It is failure tolerant in case of a few or many failures.

1.2 Rationale for a mail application

Porcupine system is applicable for all systems which requires frequent transactions and good performance, availability, manageability at high volume. A mail application is a good example to show how to meet these requirements. Porcupine uses a PC based cluster to meet these requirements. The mail service needs to handle more than ten million messages per day. It grows continuously. It is a challenging application. The mail service is write intensive and requires consistency for mail.

1.3 Clustering Alternatives

An electronic mail service should be able to write as well as read data and be scalable. There are 2 types of existing clustering models. One is to deliver read-only data. The other one is a statically distributed system. This assigns specific users to specific machines. First is not appropriate for a service in which data is frequently written. Second is not scalable because it requires adding and configuring new machines when users grow. Transactions also are not distributed uniformly. Management resources are required more as a service grows and users are tightly coupled with machines. Thus, second method is also not applicable for a mail service

On the other hand, Porcupine provides a system structure that performs well as it scales, adjusts automatically to changes in configuration and load, and is easy to manage. With the Porcupine system, one single administrator can meet requirements of one hundred million users processing a billion messages per day. The administrator can add and remove machines or even disks for the system and can handle the failure of machines easily.

1.4 Organization of the paper

This paper describes Porcupine's architecture, implementation, and performance.

- Section 2: an overview of the system's architecture
- Section 3: how the system adapts to changes in configuration automatically
- Section 4: Porcupine's approach to availability
- Section 5: scalable approach to fine-grained load balancing
- Section 6: the performance of the Porcupine prototype on our 30-node cluster
- Section 7: some of system's scalability limitations and ares for additional work
- Section 8: related work
- Section 9: conclusion

2. System architecture overview

Porcupine is functionally homogeneous across nodes. Any node can perform any function. It simplifies system configuration. This is key to the system's manageability.

Functional homogeneity ensures that a service is always available, but it offers no guarantees about the data that service may be managing. There are two kinds of replicated states.

- Hard state: an email message and user's password. They can not be lost, so stored in stable storage.
- Soft state: the list of nodes containing mail for a particular user. It can be reconstructed from existing hard state

Porcupine replicates hard state, but most soft state is maintained on only one node at a given state. This can minimize persistent store updates, message traffic and consistency management overhead. Soft state may need to be reconstructed from a distributed persistent state after a failure, but Porcupine focuses on reducing these costs and scale with the size of the system.

2.1 Key data structures

Porcupine is a cluster-based, Internet mail service that supports the SMTP protocol for sending and receiving messages across the Internet. Porcupine consists of a collection of data structures and a set of internal operations for each node.

- Mailbox fragment: hard, mail messages, replicated
- Mailbox fragment list: soft, node lists of mailbox frags
- User profile database: hard, consistent
- User profile soft state: soft, stored on exactly on node in the cluster
- User map: soft, table mapping btw hashed user and a node
- Cluster membership list: soft, replicated on each node, node's own view of the set of nodes currently functioning as part of the Porcupine cluster

2.2 Data structure managers

Porcupine uses several key managers distributed across multiple nodes:

- User Manager:
    - Manages soft state that can be reconstructed from disk
    - Maintains mailbox fragment lists for users
    - Enables scalability by distributing user profile database access across nodes
- Membership Manager:
    - Tracks the cluster's overall state
    - Monitors which nodes are up or down
    - Maintains the user map
    - Participates in membership protocols
- Mailbox Manager and User Profile Manager:
    - Handle persistent storage

    - Enable remote access to mailbox fragments and user profiles
- Replication Manager:
    - Ensures consistency of replicated objects in local persistent storage

On top of these managers, the system runs delivery and retrieval proxies to handle incoming SMTP, POP, and IMAP requests.

The architecture's key feature is decoupling mail storage from user management. This allows for flexible and scalable distribution of information. User managers on different nodes maintain soft state for specific users, demonstrating the system's distributed approach to managing user data and mail services. This design allows for scaling out.

2.3 A mail transaction in progress

In failure-free operation

2.3.1 Mail delivery

An external Mail Transfer Agent (MTA) connects to any Porcupine cluster node via SMTP. Porcupine's node is homogeneous, so no special request on a front end is needed. Delivery proxy handles message storage. It retrieves the recipient's mailbox fragment list from the recipient's user manager and selects the best node. It can choose alternative nodes if initial options are poor. The proxy then forwards messages to the selected node's mailbox manager. It handles message replication if required based on the user profile.

2.3.2 Mail retrieval

An external Mail User Agent (MUA) can contact any Porcupine's cluster node via POP or IMAP. A retrieval proxy authenticates the request through the user manager to discover the user's mailbox fragment list. Then, it contacts mailbox managers on nodes storing the user's mail to retrieve message digest information. The proxy fetches specific requested messages from appropriate nodes. If deletion is requested, the proxy handles message deletion requests. When a node's last user message is deleted, the node removes itself from the user's mailbox fragment list.

2.4 Advantages and tradeoffs

The architecture decouples delivery and retrieval agents from storage services and user managers, enabling mail delivery and retrieval even when certain nodes are unavailable. This setup supports dynamic load balancing, allowing any node to store mail for any user without a single node being permanently responsible. User mail can be replicated arbitrarily and independently of other users. If a user manager goes down, other manager nodes can take over the failure's.

The system needs to limit the spread of a user's mail and widen it to deal with load imbalances and failure. By doing so, the system acts like a statically partitioned system for efficiency, but dynamically adapts to failures or load imbalances as needed. The key tradeoff in the system lies in balancing mail distribution across nodes. While distributing a user's mail widely improves fault tolerance and load

balancing, it complicates operations: mail delivery requires frequent updates to the user's mailbox fragment list, and retrieval increases aggregate load as mail is stored on more nodes.

3. Self management

Porcupine should be self managed with diverse changes including node failure, node recovery, node addition and network failure. Its goal is to manage change automatically.

3.1 Membership services

Porcupine's cluster membership service provides the basic mechanism for tolerating change, recognizing node failure and recovery, notifying other services of membership changes in the system, and distributing new system states. This service assumes a symmetric, transitive network, and assumes that if no failures occur for a certain period of time, the nodes will converge to a consistent set of memberships.

The cluster membership service uses a variation of the Three Round Membership Protocol (TRM) to detect changes. In TRM, in the first round, when a node changes its configuration and detects this, that node becomes the coordinator and sends a "new group" message along with a proposed epoch ID using its Lamport clock. In the second round, all nodes reply with their proposed epoch IDs, and the coordinator defines new memberships based on the responses received. In the third round, the coordinator broadcasts the new membership and epoch ID to all nodes.

Once membership is established, the coordinator periodically sends probe packets over the network. Probes help merge partitions. The coordinator starts the TRM protocol when it receives a probe packet from a node, not included in its current membership list. Newly booted nodes act as a coordinator for a group as a only member. Its probe packets are sufficient to notify others.

There are several ways for a node to detect the failure of another node. It uses a timeout during remote operation. Nodes periodically "ping" their next highest neighbor in IP address order. If the ping is not responded to after several attempts, the pinging node becomes the coordinator and initiates the TRM protocol.

3.2 User map

The user map distributes management responsibilities evenly across the active nodes in the cluster. When membership changes. The user map is replicated across nodes and is recomputed during each membership change, a side effect of TRM protocol. The coordinator removes failed nodes. It then relocates to buckets in the user map, and the changed buckets contain the current epoch ID. The new user map is delivered as part of the final message of the TRM protocol.

3.3 Soft state reconstruction

The soft state of each node must be reconfigured according to the new user management responsibilities after the user map is reconfigured. Soft state includes a user profile and a list of mailbox fragments for each user. Reconstruction occurs in two fully distributed but unsynchronized steps.

In the first step, each node compares its previous and current user maps to identify any buckets having fresh assignments. Buckets with different previous epoch IDs and current epoch IDs are considered fresh. Each node independently proceeds to the second step, sending the soft state corresponding to the hard state for that bucket maintained on the sending node.

The second step locates mailbox fragments belonging to users in the freshly managed bucket to the new managed bucket. The new manager includes this node in those users' mailbox fragment lists. Then, the node retrieves the relevant data from the user profile database and transfers it to the new managed node. If your database is replicated, only the replica with the largest IP address performs data transfer. The hard state of all nodes is stored in a directory for each bucket, allowing quick review and collection when changes occur.

The cost of rebuilding soft state during reconfiguration is constant regardless of cluster size and is determined by the number of node redistribution. As the cluster size increases, the reconfiguration linearly increases, but the workload on each node remains about the same.

3.4 Node addition

It is easy to add a new node to the system with this automatic reconfiguration structure. Porcupine software is installed on a node. The membership protocol notices this and adds it to the cluster, and other nodes are notified.

3.5 summary

This dynamic reconfiguration protocol ensures the availability for any given user and facilitates the reconstruction and distribution of soft state.

4. Replication and availability

Porcupine replicates the user database and mailbox fragments to ensure their availability. Porcupine provides the same guarantees and behavior. Data remains secure as long as all replicated nodes are not lost.

4.1 Replication properties

There are five high level properties for Porcupine's replication.

1. Updates from anywhere: Updates can be initiated from any replica, increasing availability and eliminating the need for accurate failure detection.
2. Eventual consistency: During a failure, temporary inconsistencies may occur between replicas, but consistency is eventually restored. This increases availability while keeping data accessible.
3. Total update: Object updates completely overwrite existing data. Email message modifications are infrequent so lowering costs and simplifying synchronization.
4. Lock free: There are no distributed locks.
5. Ordering by loosely synchronized clocks: Uses the loosely synchronized clocks to order operations.

Update anywhere ensures that availability is guaranteed. Any incoming messages are never blocked. In other words, any Porcupine node can act as a delivery agent. Any node can receive the message, and if a failure occurs during delivery, another node will handle it. Messages can be processed more than once but it is a reasonable price to pay for the system availability.

Allow consistency attribute means eventual consistency In other words, temporarily deleted messages may reappear or data may be inconsistent, but all replica inconsistencies are reconciled.

Lock free attribute is that multiple mail reading agents may see inconsistent data.

User Profile database is replicated and it is consistent in the end. User profiles also use the same replication mechanism and are ultimately consistent. User database entry may be inconsistent because of operations ordered with loosely synchronized clocks, but becomes consistent.

4.2 Replication manager

A replication manager exchanges messages among nodes to ensure replication consistency, using two interfaces. One for the creation and deletion of objects is for the higher level delivery and retrieval agents and the other for interfacing to the specific managers is for maintaining replicated objects.

4.3 Sending and retrieving replicated mail

When a user's mail is replicated, the mailbox fragment list records the set of nodes from which each fragment is replicated. When retrieving mail, the retrieval agent connects to the least loaded node for each fragment and retrieves the entire mail content.

When creating a new replicated object, the agent creates an object ID and a set of nodes where the object cloned. The object ID is a unique string. Mail messages are in the format (type, username, messageID). A type is the object type and an username is the recipient. MessageID is a unique identifier in the mail header. .

4.3 Updating objects

A delivery or retrieval agent can send an update message to any replica manager in the set in the intended replica set and object ID. Delivery agents store messages, while retrieval agents delete or modify them. The receiving replica acts as the update coordinator and propagates updates to other replicas.

The replication manager maintains a persistent update log with entries containing:

1. **Timestamp**: a tuple <wallclock time, nodeId>. Timestamp uniquely identifies and orders updates.
2. **Target-nodes**: the set of nodes to receive the update.
3. **Remaining-nodes**: the set of peer Nodes that have not yet acknowledged the update. Remaining nodes initially equal to target nodes. It is pruned as acknowledgments arrive.

The replication manager processes the log by pushing updates to remaining nodes. When updates are acknowledged by all peers, the coordinator removes the update entry from its log and notifies peers to do the same.

Updates are synchronized by discarding older versions in favor of the newest. If the coordinator fails, the initiating agent selects another coordinator. For new objects, a new set of replicas is chosen. This ensures replication despite failures. Logs are written to disk by the coordinators and participants before updates are applied to maintain consistency.

If the coordinator fails after responding to the initiating target but before completing updates, any remaining replica can take over as coordinator. In such cases, duplicate updates are discarded using timestamps.

The update log remains small because the log never contains more than one update to the same object, and updates are propagated quickly and deleted after all replica acknowledgments.

However, with a node failing for a long time, it may lead to updates log for others growing indefinitely. To prevent this, updates are retained for a maximum of one week. A node has to be restored by deleting all of its hard state before rejoining the system.

4.5 Summary

This provides high availability with the use of consistency semantics, which is strong enough to service Internet clients using non-transactional protocols. Inconsistencies may occur for a short time, but it becomes consistent in the end.

5. Dynamic load balancing

Porcupine uses dynamic load balancing to distribute workloads across nodes to maximize throughput. Clients contact an initial node for mail delivery or retrieval, and that node uses load balancing services to select the best set of nodes for the request.

The load balancer meets four goals:

1. **Fine-grained decisions**: Optimize load balancing at the level of individual message delivery.
2. **Support a heterogeneous cluster**: Each node has different power capacity.
3. **Automatic**: No need to be manually adjusted as system grows
4. **Balancing between load and affinity**: Messages should be stored on idle nodes but prioritize nodes with existing mail for the recipient to reduce memory usage, improve disk access, and minimize inter-node RPCs.

Delivery and retrieval proxies make load balancing decisions without a centralized service. Nodes track the load on other nodes independently. Load data is collected via RPC packets and a virtual ring.

It is best to avoid disk operations because they are slow. A delivery proxy tends to distribute a user's mailbox across many nodes. This resulted in low throughput, so it is good to limit user's spread. A spread-limiting strategy controls how widely a user's mail is distributed, balancing between load and data access efficiency.

6. System evaluation

Use a 30-node cluster to test. This evaluates Performance, Availability and Manageability.

- **Performance**: The system works well on a single node and scale linearly. The system outperformed standar SMTP and POP servers.
- **Availability**: Replication and reconfiguration have low cost
- **Manageability**: The system shows node failure resilience while providing good performance. Adding a new machine improves performance. Automatic dynamic load balancing handles highly skewed workloads efficiently.

The system exhibits linear scalability, meaning performance improves proportionally as nodes are added. It maintains high availability, continuing to function even with multiple node failures. Tests on a 30-node cluster demonstrated that Porcupine outperforms traditional mail systems by minimizing bottlenecks related to file locks and temporary storage. Synthetic workloads showed the system's ability to handle uneven workloads and heterogeneous configurations efficiently.

7. Limitations and future work

Porcupine provides a practical solution for large-scale, distributed email services with built-in manageability and resilience. However, there are some aspects to think about further. Porcupine's communication patterns are flat, so it is good to investigate more network-aware load balancing. Porcupine's membership protocol may be required as the system grows, so the membership protocol for larger clusters needs to be adjusted.

8. Related work

Porcupine is a scalable, fault-tolerant email system that addresses the limitations of traditional email systems.

Grapevine was often challenged to balance users across mail servers. It had limitations in scalability and flexibility. Porcupine addresses these limitations by using a flat namespace and dynamic load balancing Porcupine uses optimistic replication for both mail and the user database.

This paper examines alternative data consistency models (BASE) for web services that are less rigid than traditional ACID database semantics. Porcupine's approach differs from previous work. It supports write-intensive applications with persistent data, and distributes services uniformly across nodes. Unlike previous load-balancing studies that assumed complete independence of incoming tasks, Porcupine balanced write traffic while considering message affinity.

Porcupine uses an optimistic approach for a replication mechanism. It allows more aggressive update retirement by using a single timestamp. Unlike previous file systems, Porcupine leverages specific data structure semantics to improve performance and decrease complexity

9. Conclusion

The research describes a large cluster mail system, Porcupine. Porcupine meets three primary goals

1. **Manageability requirements**

2. **Availability requirements**
3. **Performance requirements**

Porcupine achieves scalability and reliability through four key architectural techniques, functional homogeneity, automatic reconfiguration, dynamically adapting system configuration and replication. These techniques enable Porcupine to meet three requirements above.