N-Way Fail-Over Infrastructure for Survivable Servers and Routers

Yair Amir Ryan Caudy Ashima Munjal Theo Schlossnagle
Ciprian Tutu

Johns Hopkins University
{yairamir, wyvern, munjal, theos, ciprian}@cnds.jhu.edu

Technical Report
CNDS-2002-5
http://www.cnds.jhu.edu

December 8, 2002

Abstract

Maintaining the availability of critical servers and routers is an important concern for many organizations. At the lowest level, IP addresses represent the global namespace by which services are accessible on the Internet.

We introduce Wackamole, a completely distributed software solution based on a provably correct algorithm that negotiates the assignment of IP addresses among the currently available servers upon detection of faults. This reallocation ensures that at any given time any public IP address of the server cluster is covered exactly once, as long as at least one physical server survives the network fault. The same technique is extended to support highly available routers.

The paper presents the design considerations, algorithm specification and correctness proof, discusses the practical usage for server clusters and for routers, and evaluates the performance of the system.

1 Introduction

Maintaining the availability of critical network servers is an important concern for many organizations. Server redundancy is the traditional approach to provide availability in the presence of failures. From the client perspective, a network-accessible service is resolved via a set of public IP addresses specified for this service. Therefore, the continued availability of a service via these IP addresses is a prerequisite for providing uninterrupted service to the client. In order to function correctly, each of the service's public IP addresses has to be covered by exactly one physical server at any given time. If no physical server covers a public IP address, the clients will not receive any service. On the other hand, if more than one physical server is covering the same IP address at the same time, the network might not function properly and clients may not be served correctly.

A sizable market exists for hardware solutions that maintain the availability of IP addresses, usually via a gateway that hides the actual servers behind a smart switch or router in a centralized manner. We present Wackamole, a high availability tool for clusters of servers. Wackamole, like the annoying carnival game, ensures that a server handles the requests that arrive on any of the service's public IP addresses. Wackamole is a completely distributed software solution based on a provably correct algorithm that negotiates the assignment of IP addresses among the available servers upon detection of faults and recoveries, and provides N-way fail-over, so that any one of a number of servers can cover for any other.

Using a simple algorithm that utilizes strong group communication semantics, Wackamole demonstrates the application of group communication to address a critical availability problem at the core of the system, even in the presence of cascading network or server faults and recoveries. We also demonstrate how the same architecture is extended to provide a similar service for highly-available routers.

The remainder of this paper is organized as follows. Section 2 introduces the system architecture. Section 3 describes the system model and the core algorithm behind the engine of Wackamole and discusses its correctnes. Section 5 analyzes practical considerations and presents two applications for the system. Section 6 presents performance results concerning the reconfiguration time of Wackamole clusters. We discuss related work in Section 7 and conclude in Section 8.

2 System Overview

Our solution has three main components, presented in Figure 1:

- An IP address control (acquire and release) mechanism.
- A state synchronization algorithm (the Wackamole Algorithm).
- A membership service provided by a group communication toolkit.

The group communication toolkit maintains a membership service among the currently connected servers and notifies the synchronization algorithm of any view changes that occur due to server crashes and recoveries, and network partitions and remerges.

The synchronization algorithm manages the logical assignment of virtual IP addresses among the currently connected members, avoiding conflicts that can occur upon merges and recoveries and covering the "holes" that can arise as a result of a crash or partition.

The IP address control mechanism enforces the decisions of the synchronization algorithm by acquiring and releasing the IP addresses accordingly. These mechanisms are highly specific to the operating system on which the Wackamole system runs.

The correctness of the system is dependent on the assumption that the group communication system provides an accurate view of the current network connectivity. If there is additional connectivity beyond the one reported by the group communication system, there may be conflicts in the assignment of IP addresses. On the other hand, if the group communication system does not detect the disconnection of a server from the current membership in a timely manner, the IP addresses that were covered by that server will be unavailable to the clients, since the system will not reconfigure without the proper notification.

3 The Wackamole Algorithm

In this section we present the state synchronization algorithm that forms the core of the Wackamole system and discusses its correctness, given the assumption that the membership notifications issued by the group communication system reflect the actual network status.

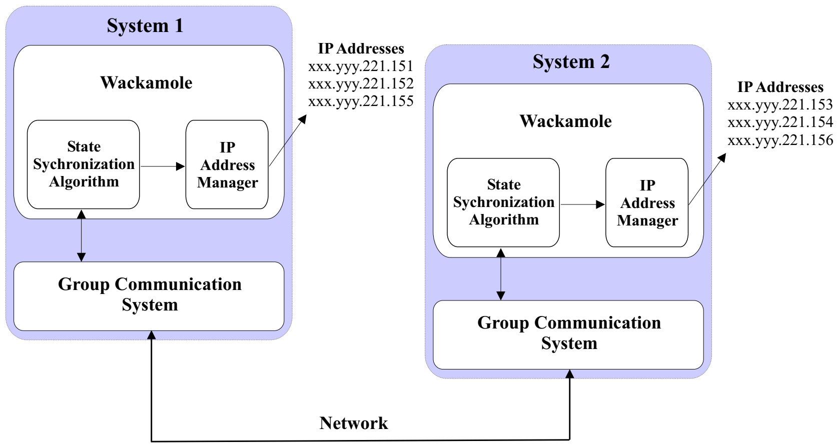

Figure 1: Wackamole Architecture

3.1 System Model

In order to formally identify the problem that Wackamole attempts to solve, we define the system model and introduce the correctness properties that the algorithm and the implemented system need to maintain.

We consider a set S=$\{s_1, s_2, ..., s_m\}$ of servers that provide service to outside client applications. The servers are all located in the same Local Area Network (LAN) but are susceptible to crashes and temporary network partitioning. During a network partition, the servers are separated into two or more components that are unable to communicate with each other.

The client applications access the services through IP addresses in the set I=$\{i_1, i_2, ..., i_n\}$. The servers in S are responsible for covering the set I of *virtual* IP addresses. We refer to the IP addresses in I as *virtual* in order to distinguish them from the stationary default IP addresses, that do not change, used by the servers for intercommunication.

The client applications are oblivious to the stationary IP addresses of the servers in S or to the possible partitioning that may exist among the servers.

In order to guarantee correct service, the following properties need to be maintained.

Property 1 (Correctness) *Every IP address in the set I is covered exactly once by a server in each subset S_k, where S_k is a maximal connected component whose servers are in the operational (RUN) state.*

Property 2 (Liveness) *If there is a time t from which a set of connected servers does not experience any network events, the servers will switch to the operational (RUN) state.*

In order to guarantee these properties we rely on the group communication system to follow the Virtual Synchrony properties [3, 9] in partitionable systems and to provide Agreed message delivery. The Virtual Synchrony property specifies that any two servers that advance *together* from one membership to the next one, will deliver an identical set of messages in the first membership. The Agreed delivery property guarantees that additionally, the messages will be delivered in the same order at all servers. Furthermore we assume that the group communication system provides a membership service that provides each server in the group a uniquely ordered list of the currently connected participants.

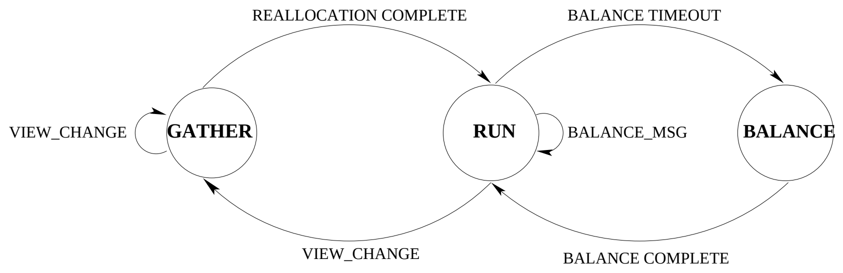

Figure 2: Wackamole Algorithm

3.2 Algorithm Specification

The algorithm runs according to the state machine presented in Figure 2.

Each server maintains a table *current_table* that contains the virtual IP allocation during the current membership. During normal operation, the algorithm is in the RUN state. In this state, each server is responsible for a set of virtual IP addresses and will answer all the requests directed to those IP addresses. While in the RUN state, the *current_table* information is conflict-free and the complete IP set is covered, maintaining the correctness guarantees of the algorithm. When the group communication system delivers a VIEW_CHANGE event, a backup of the IP table is created and a STATE_MSG is sent to every member of the new view containing the information about the IP addresses managed by the server and the identifier of the view in which it is initiated. The algorithm then moves to the GATHER state.

Algorithm 1 RUN State

```
1: when: VIEW_CHANGE do
2:     old_table = current_table
3:     send STATE_MSG
4:     state = GATHER
5: when: receive BALANCE_MSG do
6:     Change_IPs()
```

In the GATHER state each server incorporates the information received through the STATE_MSGs in the *current_table* and checks for the existence of conflicts in the IP allocation; if members from previously partitioned components are merged together, conflicts are expected since each component covers the full IP address set. ResolveConflicts() is a deterministic procedure invoked as soon as a new STATE_MSG is received, that checks whether the server that sent this message introduces any conflict with respect to the information already gathered about the set of covered IP addresses. If a conflict is detected, the server drops the addresses that are overlapping, thus restoring consistency at the network level as soon as possible.

When all the state messages (STATE_MSG) have been received, each server invokes a deterministic procedure Reallocate_IPs(). During the Reallocate_IPs() procedure, the servers make sure that all the virtual IPs are covered by a server in the current configuration. In particular, the procedure relies on the uniquely ordered membership list provided by the group communication system to distributely decide which server covers which IP address.

If the GATHER state is interrupted by a cascading VIEW_CHANGE event, the server clears its *current_table*, discarding the information already collected and reverting to the information in the *old_table*, then sends a new STATE_MSG to all members of the new configuration.

Algorithm 2 GATHER State

```
1:  when: receive STATE_MSG with current view id  do
2:     update current_table
3:     ResolveConflicts()
4:     if (received STATE_MSG from all current_table.members)  then
5:        Reallocate_IPs()
6:        state = RUN
7:  when: VIEW_CHANGE  do
8:     clear current_table
9:     send STATE_MSG
10: when: BALANCE_MSG  do
11:    ignore
```

3.3 Correctness of the Algorithm

We consider a subset of servers S'¡S that are in the operational RUN state. Between the servers in S', each virtual IP address is covered exactly once. We consider a VIEW_CHANGE event that is detected by members of S'. According to the group communication guarantees, all members of S' will receive VIEW_CHANGE notifications, even though they are possibly disconnected from each other. According to the algorithm, each server will proceed to the GATHER state and send a state message containing its local knowledge base. Let's consider a server *s* that was part of S' and is now part of S" as indicated by the group communication. The group communication system provides every *s* in S" an identically ordered list of all the servers in S".

Lemma 1 *For every connected set S' of servers in the RUN state, every IP address is covered at most once by a server in S'.*

Proof:

We consider a set of servers that are connected after a view change event, as indicated by the group communication notification. In order for the servers to advance to the RUN state, the state transfer algorithm, executed in the GATHER state, needs to complete. Therefore we consider the situation where a set of servers S' does not detect further VIEW_CHANGE events until they exit the GATHER state.

Let's assume that upon receiving the last STATE message in the GATHER state (line 4 in Algorithm 2) there exists a virtual IP address *vip* that is covered by two servers *p* and *q* in S'. According to the Virtual Synchrony and Agreed delivery guarantees of the group communication, both *p* and *q* received all the state messages that were sent during the GATHER state, therefore they received their own state messages. According to the management of the *current_table* variable from the algorithm (line 2 Algorithm 1 and lines 2,8 Algorithm 2) and the fact that only STATE_MSGs generated in the current view are considered (line 1 of Algorithm 2), the variable will accurately reflect at this point the state of the currently connected component. During this stage, following the algorithm, the servers don't acquire new IP addresses, therefore both *p*and *q* were already covering *vip* from their previous memberships. From the algorithm, in the Resolve_Conflicts() procedure (line 3 in Algorithm 2), when *p* receives the state message from *q*, it will notice the conflict in the coverage of *vip* and will adjust its IP coverage table and release *vip* if *p* appears in the membership list of S' before *q*. The same reasoning applies for *q*; therefore it is impossible for *vip* to be covered by both *p* and *q* at this point. Furthermore, both *p* and *q* will have the same view of the virtual ip coverage. Note that reaching agreement does not assume any particular relation between the initial states of *p* and q or of the other members of S'.

When all the state messages have been received each server will execute the Reallocate_IPs() procedure. During this procedure a server may acquire new IP addresses only if there is a virtual IP that is not covered by any server in S'. Since all the servers have the same view of the coverage table, they will all detect the same set of IP addresses that need to be covered. Furthermore, since they all have the same uniquely ordered list of the membership of S' the procedure Reallocate_IPs() will guarantee that each unallocated virtual IP address will be covered by exactly one server in S'. This concludes the proof of the lemma.

□

Lemma 2 *During the RUN state, every virtual IP address in the set R is covered by at least one server.*

Proof:

According to the algorithm, after a view-change, if the connectivity remains stable allowing the GATHER procedure to complete, all the connected servers will execute the Reallocate_IPs() procedure. As shown above, all servers that start this procedure in the same component will have identical views of the IP coverage and will detect the same "holes" (IP addresses that are not covered by any server in the current component). Following the algorithm, these IP's are covered at the end of the Reallocate_IPs() procedure, ensuring the complete coverage during the RUN state.

□

From the two lemmas above, we obtain the correctness property as specified in section 3.1.

We will now prove the liveness property.

Proof:

Due to the properties of the group communication delivery specification, if there is a time t from which no view-change notifications occur, then every server is guaranteed to deliver all the state messages that were sent in that membership. At that time, each server will execute the finite procedure Reallocate_IPs() and will switch to the RUN state.

□

3.4 Practical Considerations

The algorithm presented so far satisfies the correctness guarantees but can be further optimized in order to improve its performance.

From a practical perspective we want to minimize the amount of time that an IP address is covered by two or more servers in the same connected component in order to avoid network level conflicts. This is ensured by the fact that the ResolveConflicts() procedure is invoked as soon as a conflict is detected and one of the involved parties will drop the offending IP.

Algorithm 3 BALANCE State

```
1: Balance_IPs()
2: send BALANCE_MSG
3: state = RUN
4: when: VIEW_CHANGE or BALANCE_MSG do
5:    delay event
6: when: STATE_MSG do
7:    impossible
```

Of similar importance to the system is the fast completion of the reconciliation during the GATHER state. The minimal task that needs to be executed in the Reallocate_IPs() procedure is the acquisition of non-allocated IP addresses, in order to guarantee the complete coverage. However, after several partitions/merges, the system may end up with a very unbalanced allocation of IP addresses among the set of connected servers. To avoid this we can modify the Reallocate_IPs() procedure to perform load-based reallocation of IP addresses. However, this would extend the amount of time the system is in a non-operational state. Therefore we introduce a re-balancing procedure which is triggered from the RUN state by a set timeout and is executed only by one member (representative) of the connected component, selected based on the order in the membership list provided by the group communication system. The representative decides on the new IP allocation based on load balancing considerations and explicit preferences specified by each server at startup and passed along through state messages. The representative then broadcasts a BALANCE_MSG containing the new IP allocation and switches back to the RUN state. Upon receiving a BALANCE_MSG, a server in the RUN state acquires or releases the necessary IP addresses. Note that the BALANCE state executes as an atomic procedure with the server ignoring any potential VIEW_CHANGE notification from the group communication until it returns to the RUN state. Furthermore, even when a VIEW_CHANGE is detected before all servers receive and apply the BALANCE_MSG the correctness of the algorithm is not endangered since the GATHER procedure does not assume anything about the starting state of the participating servers and treats any conflict as it is discovered.

Another optimization was added in order to gracefully bootstrap the system. A server s starts with the local variable *mature* unset and without being responsible for any IP addresses. Upon receiving a view change notification, s switches to the GATHER state. If during the GATHER state s receives a state message from a *mature* server, it will mark itself as mature and continue the normal algorithm execution. If all the servers that s can contact are not mature, s will remain "immature" until a certain timeout expires after which it automatically sets itself as mature, notifies the other servers, and starts managing the IP addresses. The reason for this optimization is to avoid quick IP reallocations as the cluster is rebooted.

4 Implementation

Wackamole [18] has been implemented with cross-platform interoperability in mind; it currently supports FreeBSD, Linux, and Solaris systems. To more readily accommodate its use on multiple platforms, the implementation is separated into two clearly delineated parts. The first, comprised of generic ANSI C code, implements the core algorithm as discussed in the previous section. The second, which contains platform-specific code, implements the functionality needed to manage multiple interfaces and spoof ARP caches on each supported operating system.

4.1 The Spread Toolkit

The correctness as well as the efficiency of the system depends on the use of a group communication system that provides reliable, totally ordered multicast and group membership notifications for a cluster of servers. Wackamole was implemented using the Spread group communication toolkit [16, 1].

Spread is a general-purpose group communication system for wide- and local-area networks. It provides reliable and ordered delivery of messages (FIFO, causal, agreed ordering) as well as Virtual Synchrony and Extended Virtual Synchrony membership services. These properties match the algorithm requirements specified in Section 3.1

Spread uses a client-daemon architecture. Node crashes/recoveries and network partitions/remerges are detected by Spread at the daemon level; upon detecting such an event, the Spread daemons install the new daemon membership and inform their clients of the corresponding changes in the group membership that are introduced by the failure. Clients are also notified when changes in the group membership are triggered by a graceful leave or join of any client. The Spread toolkit is optimized to support the latter situation without triggering a full daemon membership reconfiguration, but rather informing only the participating group about the new group membership. The impact of this optimized approach will become apparent in section 6.

The Spread toolkit is publicly available and is being used by several organizations in both research and production settings. It supports cross-platform applications and has been ported to several Unix platforms as well as to Windows and Java environments.

4.2 Implementation Considerations

Wackamole's state synchronization algorithm is implemented using group membership and messaging services offered by the Spread Toolkit. Immediately upon startup, the Wackamole daemon connects to a Spread daemon running on the same host. It then relies on the regular membership messages sent by Spread to determine the current set of available hosts, and to initiate state transfer upon view-change detection. Spread is also used to ensure that messages are sent in a total order among Wackamole daemons, that old messages which must be discarded upon receipt can be identified properly, and that endian conflicts across platforms are handled correctly.

As a consequence of Wackamole's tightly-coupled relationship with Spread, some of the possible fine-tuning decisions that can be made to improve Wackamole's response time to network events are dependent on the way Spread is configured. Modifying the Spread network-failure probing timeouts must be, however, done on a system-specific basis. If not done properly, this tuning can be detrimental to the performance of a Wackamole cluster by increasing the number of false-positive network failures. The impact of this tuning is analyzed in Section 6.

A Wackamole daemon that becomes disconnected from Spread will drop all of its virtual interfaces and enter a cycle in which it periodically attempts to reconnect to Spread, because it cannot ensure correctness without the services Spread provides. This behavior allows clusters to survive changes to the Spread daemons with which they communicate, such as version changes and re-initializations for configuration modification, taking into account the fact that Spread may be used for multiple applications concurrently.

In order to provide continuous service to the Wackamole daemons, Spread must bind to IP addresses that are not subject to Wackamole's management. Consequently, it is possible to run Spread on a separate Network Interface Card (NIC) than the one being used for the virtual IP addresses managed by Wackamole. Also, Wackamole does not provide failure detection of any of the applications that may be relying on its management, e.g. HTTP servers. Either of these two situations can cause failures that are not detectable by the Spread membership service. Currently, this problem is not addressed by Wackamole's implementation; possible solutions include run-time checks on the availability of the NIC or the specific applications that use Wackamole.

Another practical aspect of the Wackamole implementation is the addition of an input channel to allow administrative control of a cluster's behavior. Also, the way Wackamole handles network failures can be modified, such that all decisions are made by a deterministically chosen representative and imposed upon the other daemons, rather than made independently by each daemon through a deterministic decision process. This will enable changing the way virtual address allocation decisions are made without breaking version compatibility.

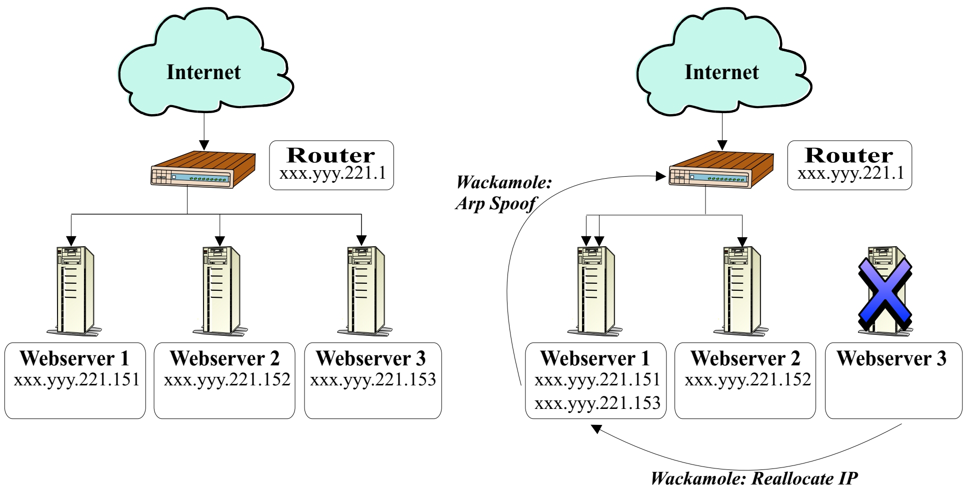

Figure 3: **N-Way Fail-Over for Web Clusters**: The IP of the failed server is reassigned to one of the available servers and the router is informed of the ARP change.

5 Practical Applications

The two primary applications for which Wackamole was developed are web clusters and fail-over routers. The implementation of Wackamole takes these applications into account and can be fine-tuned to make appropriate trade-offs in either situation. We show how Wackamole provides availability for these applications.

5.1 N-Way Fail-over for Web Clusters

Web clustering is the application that drove the creation of Wackamole. In combination with Domain Name Service (DNS), Wackamole provides the functionality to enable websites served by multiple IP addresses and/or hosted on a cluster of machines to be highly available. The generic management of virtual addresses has already been discussed. However, this class of application requires Wackamole to perform an additional task: ARP spoofing.

While IP addresses are used for routing on wide area networks, on local area networks Media Access Control (MAC) addresses are used. An IP address is resolved to a MAC address using the Address Resolution Protocol (ARP). In and of itself, this is not a problem for Wackamole. However, ARP data is cached on an IP address basis. This cache must be updated for any virtual address that is moved from one host to another, on each host that has cached an ¡IP address, MAC address¿ pair for that virtual address.

Since we assume that we are managing a local area cluster, all requests to the web server must come through a router. That router's ARP cache must be updated in order to ensure that it correctly forwards packets to the appropriate machine whenever Wackamole alters the allocation of virtual addresses within the cluster. Consequently, part of Wackamole's platform-specific code deals with spoofing of ARP reply packets to force updates to the router ARP cache.

An example layout for a Wackamole-assisted web cluster (Figure 3) consists of a number of web servers and a single router through which outside requests are made. Each of the web servers must be running a Spread daemon, likely on a private interface, and must be running a Wackamole daemon, to ensure that IP addresses are correctly allocated. Each server must also be responsible for notifying the router to update its ARP cache when it assumes responsibility for a new virtual address.

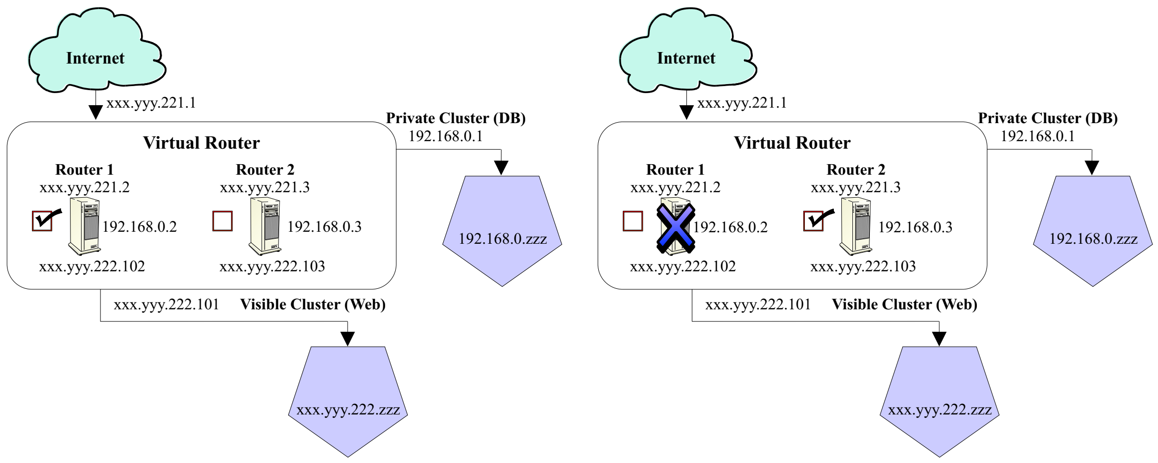

Figure 4: **N-Way Fail-Over for Routers**: At any point, a single physical router acts as the virtual router, managing the virtual addresses xxx.yyy.221.1, xxx.yyy.222.101, 192.168.0.1.

5.2 N-Way Fail-Over for Routers

Router management is another application that has emerged as a common use for Wackamole. An example layout for this application of Wackamole consists of multiple physical routers that act as a single virtual router as depicted in Figure 4. An indivisible set of virtual addresses on different interfaces is allocated to the physical router currently acting as the fail-over router. In the figure, these IP's are xxx.yyy.222.101, 192.168.0.1, and xxx.yyy.221.1 which represent the logical IP addresses of the router in the three networks that it serves. The picture also shows the stationary IP addresses of each physical router, on each of the three networks. These IPs are depicted in the figure inside the Virtual Router box.

If the interface through which the machine is connected to Spread fails, or the machine itself crashes, the set of virtual IP addresses will be reallocated to a different machine. The set of physical routers running Wackamole, each of which is potentially ”the” router, can be conceptualized as a single virtual router.

For the most part, the presented Wackamole architecture can support this application without additional modifications beyond what is needed for web clustering. However, a router needs to simultaneously exist on multiple networks in order to route packets between said networks. Therefore a set of virtual IP addresses must be considered as a single entity. As a result, Wackamole was modified to support grouping of multiple IP addresses, possibly on different interfaces, as an indivisible set of virtual addresses. This enables the correct handling of situations where a single host being managed by Wackamole must be accessable on multiple virtual addresses.

Furthermore, the notification mechanism for ARP spoofing must be enhanced in order to update the ARP cache of every host which has resolved the MAC address of the virtual router. To facilitate the necessary notification, each Wackamole daemon periodically sends data from its ARP cache to all other daemons. This makes it possible for a daemon to approximately know the set of machines that must be notified when it assumes responsibility for a virtual IP address. Obviously, this approach does not scale well to very large LANs. We are investigating the potential of applying garbage collection techniques to make the ARP spoof notification more accurately targeted towards only hosts that require such notification.

Our solution provides the additional benefit of allowing a heterogenous set of physical routers to collaborate in forming a virtual router. Using a variety of architectures and operating systems for

Parameter Name	Default Spread	Tuned Spread
Fault-detection timeout	5	1
Distributed Heartbeat timeout	2	0.4
Discovery timeout	7	1.4

Table 1: Spread timeout tuning (seconds)

the physical routers provides increased protection of the virtual router against security exploits that may target specific platforms.

6 Performance Results

In order to assess the performance of Wackamole, the most relevant measurement is the length of the service interruption perceived by a client when the server with which it is communicating is made unavailable by a fault. For this reason, we report results for the average availability interruption time when a computer running Wackamole fails and its virtual addresses must be reallocated to another computer, as measured from a client.

In our experiment we place a simple server process on each computer using Wackamole. The server responds to UDP packets by sending a packet containing its hostname. A client process on another computer is instructed to continuously access a specific virtual address by sending UDP request packets at a specified interval, and record the hostname of the server that responds as well as the time since the last response was received. For our experiments, the ideal interval between requests was found to be 10 milliseconds. When a fault is induced by disconnecting the interface through which Spread, Wackamole, and the experimental server access the network, the client will stop receiving responses to its requests. When Wackamole completes the IP address reallocation procedure and the client's ARP cache is updated, the client resumes receiving responses to its queries, this time from the computer that has aquired its target address. The time elapsed between the receipt of the last response from the disabled computer and the first response from the new server is the *availability interruption time* from the experimental client's point of view. While there is a small possibility for error in this measurement due to the interval between requests and fluctuations in the network, our measurements represent an upper bound on the actual interruption time.

As discussed, Wackamole depends upon the Spread group communication toolkit for notification of membership changes. For this reason, the availability interruption time measures the total time to complete four actions: Spread's detection of membership changes, Spread's daemon and process group membership installation, Wackamole's state transfer and virtual address reallocation, and Wackamole's ARP spoofing.

In light of this dependency, we performed two sets of experiments. The first set uses the default Spread settings with timeout intervals designed to perform adequately on most networks, for a variety of applications. The second set uses a fine-tuned version of Spread, in which we adjusted the relevant timeout intervals specifically for the Wackamole application and our network setup. Both experiments were run on a 100Mbit Ethernet LAN cluster, maintaining 10 virtual IP addresses in a cluster, and varying the number of servers from 2 to 12.

Table 1 shows the differences between the two experiment setups. The timeouts presented in the table cover the major components of the time it takes Spread to notify Wackamole of network faults. The distributed heartbeat timeout specifies an interval after which a Spread daemon notifies other daemons that it is still in operation. The fault-detection timeout begins at approximately

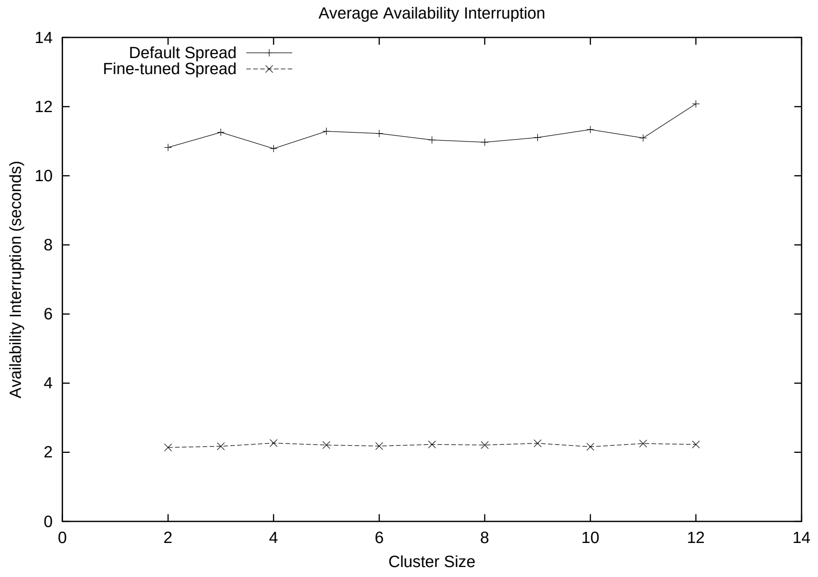

Figure 5: Average Availability Interruption with Varying Cluster Size

the same time as the distributed heartbeat timeout; after the fault-detection timeout expires, if a daemon has not specified that it is operating, Spread assumes a fault has taken place and attempts to reconfigure. Because a fault could occur at any time during the heartbeat interval, the actual time to detect a failure ranges from *failure-detection timeout - distributed heartbeat timeout* to *failure-detection timeout*. The discovery timeout is the time spent performing this reconfiguration by determining the currently available set of Spread daemons and installing this configuration view at each daemon. Thus the time it takes the default Spread to notify Wackamole of a failure (ignoring the minor overhead of Spread's group membership procedure) ranges from 10 seconds to 12 seconds. For the tuned Spread, this time ranges from 2 seconds to 2.4 seconds.

Figure 5 displays the average availability interruption time when varying the cluster size, for each version of Spread. We notice that the Spread timeouts account for the majority of the interruption time recorded in our experiments.

These results were obtained using a cluster of servers under low average load. Both Wackamole and Spread can be used in production on highly-loaded machines as well. However, it is recommended that both daemon processes be run with high priority (real-time priority under Linux) in these types of environments in order to avoid false positive errors. This has no adverse impact on the cluster performance as Spread and Wackamole hardly consume resources when used for this application.

Also relevant is the availability interruption time when a Wackamole daemon leaves voluntarily, not as the result of a failure. This is experienced when Wackamole daemons are taken offline for administrative or policy reasons. However, we found that this time interval is difficult to measure precisely, because it is more susceptible to context switch times and other low-level fluctuations. In general, our measurements suggest a conservative upper bound of 250 milliseconds of availability interruption on our experimental cluster; most of our measurements actually recorded an interruption time as small as 10 milliseconds.

7 Related Work

There are two areas that relate to the work presented in this paper On one hand our solution benefits from extensive research in the areas of group communication and distributed algorithms. On the other hand, various other techniques have been employed to provide availability for critical services.

Research in the group communication area has lead to the implementation of several systems which provide properties similar to those required by the Wackamole algorithm. Among such systems we mention Horus [17], Ensemble [8], Totem [10]. The Wackamole algorithm uses a design similar to the state machine approach for maintaining consistent state in distributed systems [14, 11]

Wackamole as a fail-over solution is designed to preserve the IP presence of a service. The Virtual Router Redundancy Protocol (VRRP) was designed to perform a similar task for routers. VRRP specifies an election protocol that dynamically assigns responsibility for a virtual router to one of the VRRP routers on a local area network. VRRP design is chaired by an IETF working group and has been formalized into an Internet Standard RFC 2338 [13]. A similar protocol is the Hot Standby Router Protocol (HSRP) developed by Cisco [12]. In essence, HSRP elects one router to be the active router and another to be the standby router. The active and the standby routers send hello messages. The standby router is the candidate to take over the active role if the active router faults. All other routers are monitoring the hello messages sent by the active and standby routers. Routers may be assigned priorities. The router with the highest priority will be come the active router after initialization. After an certain Active timeout elapses without hearing hello messages from the active router, the standby router takes over. Similarly if a Standby timeout elapses, a monitoring router (if such exist) with the lower IP address takes over the stanby role. By default, hello messages are sent every 3 seconds and the Active and Stanby timeouts are set to 10 seconds.

Aside from IP fail-over, front-end high-availability and load-balancing devices are often used in front of mission critical networked services to provide uninterrupted service in the event of a system failure. These devices perform application level checks against machines in the cluster and keep track of which machines are providing service. They present a virtual IP address to which client connect, and then dynamically set the local endpoint of the IP connection to an active machine in the local cluster. These devices are in common use today to support most large Internet sites and are provided by a variety of vendors. Such devices include Cisco's Arrowpoint [4], Foundry's ServerIron [6], F5's BIG/ip [2], CoyotePoint's Equalizer [5], and Linux Virtual Server [15].

While these components may provide more than just high-availability (specifically load balancing), they themselves must be made highly available – a single of any such component is a single point of failure. Each vendor has its own method of providing High availability between two of their devices, but an application independent protocol such as VRRP or Wackamole could just as easily be used to accomplish this.

Many services need high availability and only remedial load-balancing techniques such as multiple DNS A records. For these architectures, using an IP fail-over protocol directly on the machine providing the service in question, reduces the need for complicated, expensive and otherwise unnecessary HA/LB components.

The PolyServe Matrix HA [7] product provides a service similar to Wackamole. The technical details of the implementation or the soundness of the protocols cannot be assessed as the product and procotols are unreleased. Until recently the Polyserve solution only offered pairwise fail-over, where every server is covered by one other specific server. The latest version of the software is reporting use of the Spread Toolkit and provides N:M, N:N, and N:1 IP failover.

8 Conclusions

This paper presented a software-based distributed solution for providing high availability for clusters and routers at the IP addressing level. The core algorithm relies on a group communication service to monitor the currently connected membership and reallocate virtual IP addresses that are accessible to client machines, between the avaible servers. We presented the algorithm and discussed its correctness. We discussed two classes of practical applications of the system and provided experimental performance results.

The Wackamole system is available as an open-source tool since August 2001 (www.wackamole.org). During the past 15 months the system was downloaded more than 800 times and is actively used in production environments for both the web-cluster and router availability applications described in this paper. This work demonstrates how sound academic research can readily make an impact in production environments.

References

[1] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.

[2] BIG/ip. http://www.f5.com/f5products/bigip/.

[3] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the ACM Symposium on OS Principles*, pages 123–138, Austin, TX, 1987.

[4] Cisco/Arrowpoint. http://www.cisco.com/en/us/products/hw/contnetw/ps792/index.html.

[5] Equalizer. http://www.coyotepoint.com/equalizer.htm.

[6] Foundry/ServerIron. http://www.foundrynet.com/products/webswitches/serveriron/.

[7] PolyServer Matrix HA. http://polyserve.com/products.html.

[8] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[9] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *International Conference on Distributed Computing Systems*, pages 56–65, 1994.

[10] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[11] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[12] Hot Standby Router Protocol. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ics/cs009.htm.

[13] Virtual Router Redundancy Protocol. http://www.ietf.org/html.charters/vrrp-charter.html.

[14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[15] Linux Virtual Server. http://www.linuxvirtualserver.org/.

[16] The Spread Toolkit. http://www.spread.org.

[17] R. van Renesse, K. P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[18] Wackamole. http://www.wackamole.org.