A Hot–Failover State Machine for Gateway Services and Its Application to a Linux Firewall

Harald Roelle*

Munich Network Management Team, University of Munich, Oettingenstr. 67, D-80538 Munich, Germany, roelle@informatik.uni-muenchen.de

Abstract. Nowadays, companies of any size rely on their IT–infrastructure since it provides connectivity to the outside world. Services like firewalls, being positioned between the own domain and a foreign one, form a premises for higher level services. Therefore, such gateway services must be considered as especially mission–critical. While there exist high availability solutions for special service types, a generic solution which can be applied to arbitrary gateway services, especially for smaller sized scenarios, is missing. Fault tolerance in terms of high availability is addressed by this paper through the concept of redundancy. Presenting a generic state machine for monitoring and takeover processes, it leads to an universally applicable logic. The state machine's basis is derived from requirements posed by the generic scenario of gateway services. Furthermore, our solution's practical applicability is shown by presenting an implementation carried out for a Linux–based firewall system.

1 Introduction

Today, IT–equipment and provided services represent mission–critical parts in a company's business infrastructure. Serving as a fundament for advanced services, this especially applies to those involved in connectivity to foreign domains. Against the background of increasing security threats even small and mid–range companies cannot abstain from complex gateway services like firewalls and application–level gateways. Hence, for such services, fault tolerance in terms of availability is gaining rising importance in environments of any size. While large companies already afford adequate solutions, in smaller scenarios the problem is often disregarded.

Especially in lower budget environments, nowadays off–the–shelf hard– and software components (like industry–standard PCs running multi–purpose operating systems like Solaris, Linux or Windows) are used to realize gateway services. But with a rising number of mission–critical components the probability of service failures increases equally. As a common and widespread solution, this problem can be solved by

* The author wishes to thank the members of the Munich Network Management (MNM) Team for helpful discussions and valuable comments on previous versions of this paper. The MNM Team directed by Prof. Dr. Heinz-Gerd Hegering is a group of researchers of the University of Munich, the Munich University of Technology, and the Leibniz Supercomputing Center of the Bavarian Academy of Sciences. Its web–server is located at http://wwwmnmteam.informatik.uni-muenchen.de/

M. Feridun et al. (Eds.): DSOM 2002, LNCS 2506, pp. 181–194, 2002.

hot–standby redundancy, which means to supply additional systems with the ability to immediately take over service provisioning in case of a failure. While this approach is often feasible from a financial viewpoint, it poses problems in management: To realize seamless service availability, failures have to be detected and service provisioning has to be shifted, preferable automatically, to a different system. Furthermore service functionality has to be replicated on the redundant systems.

As gateway services like firewalls and application–level gateways often are not dependent on a persistent state, service replication is fairly straightforward by spreading static configuration data among systems in the redundancy cluster. Therefore this paper concentrates on failure detection and hot service takeover. It presents a reasonable generic solution for realizing fault tolerance by redundant systems, particularly for smaller environments based on off–the–shelf hard– and software components. For being universally applicable, the whole logic regarding the processes of monitoring and takeover is depicted by a state machine. Thereby an eye is kept on being directly implementable and easily customizable for concrete services with minimal additional effort.

In the following section an abstracted scenario is used to derive requirements for the solution. Afterwards Section 3 presents related work and compares it to these requirements. Section 4 presents our solution and the following Section 5 shows an implementation of our solution for a packet filtering firewall. Finally Section 6 concludes the paper und indicates further work.

2 Scenario and Requirements for a Generic Solution

For developing a widely applicable solution an abstract scenario is introduced in this section. By means of this scenario, important terms will be defined and requirements for a generic solution will be identified afterwards.

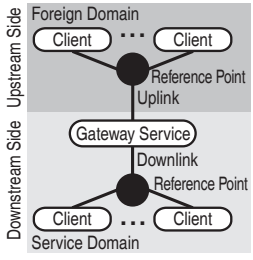

Fig. 1. Logical View on Abstract Scenario

Figure 1 shows a logical view of our abstract scenario. It is divided into two domains. One provides the gateway service that is linked by a communication network both inside the own domain (**downstream side**) and to the foreign domain (**upstream side**). The particular network links ending at the **reference points** are referred as **uplink** and **downlink**. Systems accessing service functionality from either sides are called **clients**. A **gateway service** in our context now is defined as a service being linked to different domains by discrete network links, while being completely located inside a single domain. Here we furthermore assume that the communication protocol used to access the service is not genuinely capable of load balancing, otherwise fault–tolerance would already be present by service redundancy via the protocol. Examples for a scenario in our sense are IP edge routers, firewalls or application–proxies without a dynamic routing protocol.

Now that the system hosting the service is identified as a classical single–point–of–failure, adding extra systems introduces fault–tolerance in terms of high availability (Fig. 2). The system which currently provides the service is called **master system**. Additional hosts, which are ready to take over provisioning are called **backup systems**.

Having backup systems raises the question how failure detection and handover in the case of a failure could be realized. For specifying a universally applicable answer, we have identified specific **requirements** by performing an in–depth analysis of concrete scenarios:

Separation of logic and actions: In designing failover solutions two major layers of abstraction can be differentiated. The logic is responsible of steering and coordinating the processes during monitoring and handover. In contrast, actions describe the execution of concrete activities. For gaining a generic solution, it is necessary to separate the logic (which is identical for any service the solution is designed for) from service specific actions.

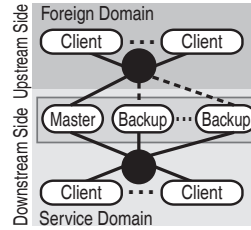

Fig. 2. Log. View with Redundant Systems

Distinction of monitoring services and communication links: To design a flexible solution applicable to a wide range of scenarios in a modular way, monitoring of service functionality must be strictly separated from checking of communication links. This enables partial reuse of realizations, when only the service type changes but the communication technology remains the same, and vice versa.

Service monitoring from client's perspective: In the sense of customer orientation, service monitoring must not only be carried out locally [4]. From a client's perspective the service is provided properly when service functionality and communication links are available. Therefore, in addition to testing service functionality itself, monitoring also must include service accessibility through the communication links. In case of gateway services both links to the up– and downstream side must be taken into account.

Independence from specific services and communication technology: A generic solution must neither make any assumption on the specific service nor the communication technology being used. Particularly in smaller scenarios, where off–the–shelf communication services (like IP mass–services) are deployed, special management and communication protocols are neither available by default nor can be ordered optionally.

Independence from technology–specific communication primitives: For gaining wide applicability, special communication primitives like multicasting must not be used for the purpose of coordination the takeover. Only unicast communication can be assumed to be available in general.

Minimal active links to foreign domain: Regarding security, connections to foreign domains should considered to be harmful. Therefore the number of active links to the foreign domain has to be kept to a minimum. This implies that upstream links should be kept down unless they are absolutely needed.

No need for extra hardware: For gaining flexibility in the usage of machines and achieving a short setup time of backup hosts, no hardware should be needed in addition to the one required to deliver the service anyway.

The next section will explain why existing solutions to the given problem need to be rethought and afterwards our own solution will be presented.

3 Related Work

In the field of fault–tolerance and high availability a great amount of valuable work has been published. A whole class of solutions deals with specific services and is mostly aimed for large–scale scenarios. In addition, they use specific properties and address special requirements of the particular service type, especially dealing with questions of service replication. High availability for database management systems is a typical example. As they are aiming for a specific scenario, they are not considered further.

Another class of fault tolerance solutions proposes special communication protocols, like Stream Control Transmission Protocol [15], to manage the takeover process. As these are specific protocols, the requirement of independence from specific communication architectures is violated.

The IETF Working Group for Virtual Router Redundancy Protocol (VRRP) [9] has developed a protocol [11,7] delivering hot–standby redundancy for routing devices. Major limitations regarding our requirements result from being specific in communication technologies: On layer 2 it assumes IEEE 802, on layer 3 IPv4 is required. This limitation in generality is underlined by the fact that even for IPv6 a new revision [6] had to be prepared. In specifying a state machine consisting of only three states, specification is imprecise. Nevertheless VRRP was a good source of inspiration in general. Furthermore in [10] it addresses management explicitely. Closely related to VRRP is the Cisco Hot Standby Router Protocol (HSRP) [12]. It is also designed specifically for routing services. In addition to the prerequisites of VRRP, it requires a dynamic routing protocol on the upstream side. Although not being directly applicable as a solution, it served as a valuable inspiration, namely its in–detailed specified state machine.

Many solutions try to overcome availability limitations by adding load balancers, whose purpose is to spread service requests to a cluster of hosts. While this approach remedies shortcomings on service hosting machines, it simply shifts the single–point–of–failure problem to the load balancer itself. Consequently they don't solve the problem, but the load balancing service as a gateway service itself is subject for redundancy and considerations made so far apply for it.

The IETF Working Group for Reliable Server Pooling [8] proposes an architecture [17] which can be used to provide high availability for various services. In fact two solutions are suggested. The one introduces a special protocol [16] for servers and clients and thus violates the requirement of being independent from specific communication architectures. The other solution suggests to install a proxy which is equivalent to a load balancer. Hence, this solution also must be disregarded. Anyway, other documents of the working group helped in developing our own solution. For identifying requirements [18] made suggestions and [2] contributes by reviewing further related work.

Finally two prominent projects of the open source community should also be reviewed: High-Availability Linux (HA-Linux) Project [3] and Linux Virtual Server (LVS) Project [13,19]. Both projects' goal is to provide high availability to Linux by redundancy in specifying clustering solutions. HA-Linux provides the takeover daemon "heartbeat" [14]. It fully meets the requirement of independence from concrete communication technologies and does not require extra communication channels for message and heartbeat signaling. Unfortunately it is strictly limited to single backup host and does not include any options for service monitoring facilities. As LVS's failover daemon "keepalived"

[1] implements VRRP, it suffers directly from VRRP's limitations. Furthermore, LVS focuses in general on services solely connected to the domain it is hosted in and therefore does not address problems related to gateway services.

4 Generic State Machine

This chapter presents our solution for designing generic redundancy for gateway services. It starts with a design overview, followed by an in–detail explanation of our generic state machine (Sec. 4.2). Section 4.3 explains messages and timers used in the state machine. Afterwards customizable procedures for applying the generic state machine to concrete scenarios are presented in Section 4.4. Finally, Section 4.5 summarizes our design.

4.1 Design Overview

The overall goal of our solution is to keep the gateway service operational in case of a failure. This is realized by setting up a redundancy cluster, where one single host acts as the master and an arbitrary number of hosts are serving as hot–standby backups, ready to take over service provisioning in case of a failure (see Fig.2). Hereby first failures of **service functionality** of any cause on the current master (e.g. internal service errors, hardware failures, ...) are considered. Second, failures in **communication links** on any machine in the redundancy cluster, either on up– and downlinks, are taken into account.

Our design incorporates three main components: a **local state machine** executed on each host in the redundancy cluster, **message exchange** between hosts and a **status table** which lists all hosts in the redundancy cluster:

The state machine is responsible for monitoring service operation and coordinates service takeover in case of a failure. Customizable procedures are used to describe the necessary activities. They are noted as actions inside the states and are executed sequentially on entry into a state.

Message exchange: For coordinating a service takeover, messages are exchanged between hosts in the redundancy cluster and thereby our design assures that exactly one master host is present in the cluster. As the following explanations will show, the state machine's design is robust against loss of messages and assures that communication is operational before a backup becomes the new master host.

Status table: To decide which machine is to become the new master, the status table is used. It priorizes all hosts in the redundancy cluster in the manner of a total order, with the current master owning the highest priority. All hosts in the cluster are listed in this status table together with their current priority. Every host keeps a local copy of the table. Maintenance and distribution of the table is managed by the state machine, including new hosts being dynamically added to the cluster and others removed from it.

Furthermore, we make the reasonable assumption that the state machine is implemented as specified and executed correctly as long as hardware is operational. The second assumption is that monitoring and testing procedures are positive–definit. This means, whereas an erroneous negative result is acceptable (claiming that the object under test has failed, while it is still operational), a positive result must be reliably valid.

The next section explains our generic hot–failover state machine in detail.

4.2 Hot–Failover State Machine

The UML statechart diagram in Fig. 3 shows the Main state machine. It gives an overview over the state machine as it is executed on each host participating in the redundancy cluster for a certain service.

The process starts with an initialization phase (init). Next, based on the initial priority received during init main state, a decision on the initial state of the host is taken (decide). When the host's priority is lower than the one of the master, backup main state is entered. In case of the own priority being equal to the current master priority this host is the active master, consequently service main state is the next one. If the master's priority is lower than the one of this host, a service takeover is necessary, accomplished by moving to handover main state.

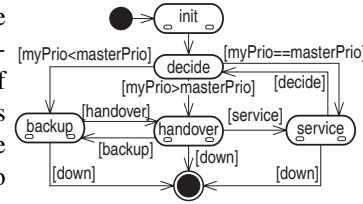

Fig. 3. Main State Machine

When a service takeover is necessary, handover main state is entered. If the takeover succeeds, service main state is reached, else the machine falls back to backup. When a host gives up service main state it falls back to the initial decision.

In the following sections all mentioned main states are explained in detail by showing sub–states and depicting conditions triggering transitions between the main states.

Init Main State is passed only once in the life–cycle of the state machine and detects the host's initial priority.

Entering check service present state (Fig. 4) the presence of a master host is detected by the customizable procedure `remoteSvcChk()` (see also backup main state). When it succeeds a *SvcCheck* message is sent. When receiving a *SvcChkReply* message, the status table is initialized by the message's contents (see Sec. 4.3). When no *SvcChkReply* message arrives the cycle is restarted. In case of an initial bootstrap of the cluster, no master service will be detected. Then the status table has to be preset from a statically predefined table.

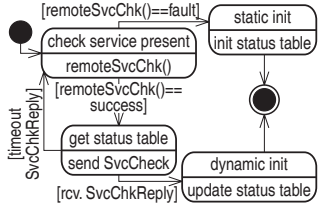

Fig. 4. Init Main State

Backup Main State describes the behavior of a host acting as a backup for the service, remaining in this state until a service failure is detected. To discover such a failure an active service monitoring from the client's perspective is performed here.

Entering in backup idle state the *SvcCheckTimer* is started. Being started on each entry, this timer is responsible for triggering periodical remote checks of the service. In case there is a *SvcCheckTimer* alarm, the service functionally is first checked from the client's point of view (`remoteSvcChk()`). In case of failure, backup main state is left and handover main state is entered (Fig. 3). On success, a local service check on the master host is triggered by sending a *SvcCheck*

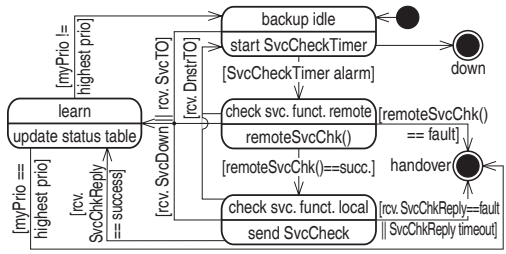

Fig. 5. Backup Main State

message. If a *SvcChkReply* message with fault status is received or does not arrive at all, the service is considered to have failed and a takeover is started. When *SvcChkReply* indicates success, in learn state the local status table is updated from the message's data. Depending whether the own priority in the status table is the highest, learn state is left. If it is the highest, this host is designated to become the new master and therefore a handover is necessary. Otherwise the system returns to backup idle state and the so far described process restarts.

Both checking processes are interrupted immediately when either a *DnstrTO*, *SvcDown* or *SvcTO* message (see handover and service main states) is received. As they all indicate a change of the master host, further service checking would be useless.

Handover Main State. When the service monitoring procedures in backup main state indicate a failure, a backup host needs to become the new master. As the failure indication might not only be provoked by a service failure, but also by a failure in the backup host's network connectivity, these cases have to be separated before the service takeover finally can be completed. This distinction is the main purpose of the handover main state (Fig. 6).

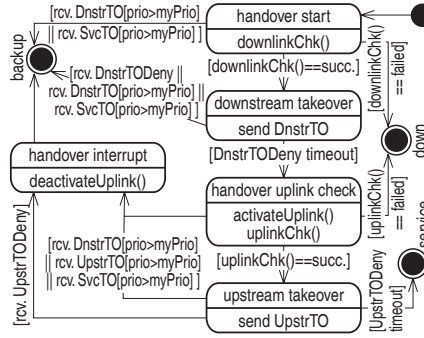

Fig. 6. Handover Main State

In handover start state the downlink is checked by `downlinkChk()` to ensure communication to the other hosts is functional. On success, in downstream takeover state a *DnstrTO* message is sent to indicate the beginning takeover to other hosts in the redundancy cluster, especially to the master host. The master may now reply a *DnstrTODeny* message (see service main state), leading to fall back to backup main state. If none is received, in handover uplink check state, first the uplink is activated by `activateUplink()` and gets tested by `uplinkChk()`. If it succeeds, both links are proven to be functional. Therefore taking over the upstream is signaled to the other hosts by sending a *UpstrTO* message in uplink takeover state. If the current master host raises its veto by sending a *UpstrTODeny* message (see service main state), the uplink is shut down by `deactivateUplink()` and the host remains in backup main state. Otherwise the takeover is completed and service main state is reached.

Takeover is interrupted if either a *DnstrTO*, *UpstrTO* or *SvcTO* message with a higher priority than the own is received, since this indicates that another host is going to take over the service. When any of the link checks fail, the host is considered to be inoperable for backup use and the state machine quits.

Service Main State. A host being in service main state designates it as the master host which currently executes the service. Furthermore it is responsible for maintaining the status table. To avoid inconsistencies it is changed only on the master host. Besides, a local monitoring of the service is accomplished and on–demand checks are carried out.

When entering service main state, first the status table is updated to reflect the new situation which is announced to other hosts in the redundancy cluster by sending a *SvcTO* message. Afterwards the service is launched (`activateSvc()`) and then the takeover is made public to clients on both interfaces (`takeoverDnstr()`, `takeoverUpstr()`). Before moving to service active state the *SelfChkTimer* is started. It is responsible for periodically triggering the self monitoring process similar to *SvcCheckTimer* in backup main state. When a *SelfChkTimer* alarm occurs, a full service check performed in service full check A state.

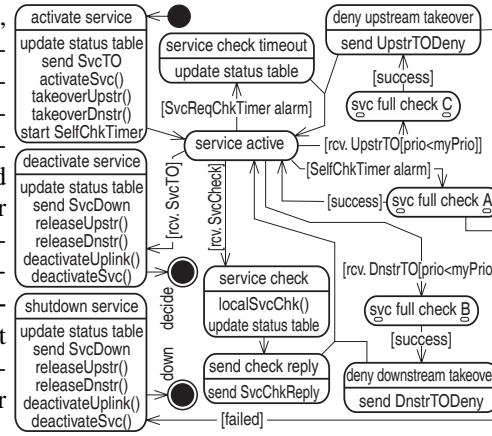

Fig. 7. Service Main State

Service full check B state is entered on arrival of a *DnstrTO* message from a host with a lower priority than the own. As this indicates a takeover attempt on the downstream side by a backup host, a complete check is performed. In case all steps were successful, the takeover is considered to be unnecessary and consequently is vetoed by replying a *DnstrTODeny* message. This process is only executed in case of a takeover attempt by a lower priorized host, as this leaves the possibility of direct takeovers by new higher priorized candidates entering the redundancy cluster. A similar process starts for the upstream side in case of a *UpstrTO* message arriving.

As the master host is responsible for maintaining the status table, for each host in the redundancy cluster a *SvcReqCheckTimer* is present. An alarm of such a timer indicates that no heartbeat was recognized from the associated host. Therefore it is considered to have left the redundancy cluster and the status table is updated accordingly.

A simple service check is carried out by `localSvcChk()` on arrival of a *SvcCheck* message from a backup host. As this also indicates that this host is alive, it serves as a heartbeat signal. In consequence, the associated *SvcReqCheckTimer* is reset. When the incoming *SvcCheck* message has been sent by a new host in init main state, the new host is added to the status table. The result of the test is sent back by a *SvcChkReply* message.

Service main state may be left for several reasons. First, in case of failure detection (leaving to shutdown of the system) and second, when an explicit service shutdown is requested (returning to decide state in Fig. 3). In both cases service main state is left carrying out the same actions but with different succeeding states. A shutdown might result from one of the full service check states being left in failed state or a manual shutdown of the service. Explicit service takeover is signaled by another host from entering service main state, therefore it is detected by receiving a *SvcTO* message. Before leaving service main state, first the status table is updated and a *SvcDown* message is sent to all hosts. Clients are informed on both interfaces by `releaseDnstr()` and `releaseUpstr()`. Finally the uplink and the service itself are shut down by `deactivateUplink()` and `deactivateSvc()`.

Service Full Check A|B|C Sub–States. Although the three service full check states have different preceding / succeeding states, they have the same internal behavior in common. Therefore they are all specified in common in Fig. 8. First, in downlink check state the *SelfChkTimer* is stopped and the downlink is tested via the customizable procedure `downlinkChk()`. On success the uplink check state is entered, where the uplink is tested by `uplinkChk()`. When the test was successful, too, the service is checked locally using `localSvcChk()`. If this test succeeds the whole subprocess was successful. In case any of the checks failed, the subprocess is left immediately in failed state.

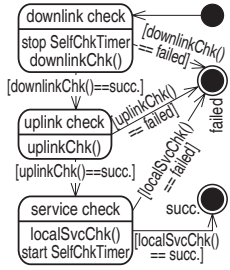

Fig. 8. Service Full Check A|B|C Sub–States

4.3 Messages and Timers

While semantics of messages exchanged between hosts in the redundancy cluster already became clear in the previous sections, Fig. 9 specifies data carried with the messages, whether they are uni– or multicast, the communication side they are exchanged on, and their principle source and destination. Note that although some messages are multicast ones in principle, this does not imply that communication services must be capable of true multicasting. As every host explicitly knows the other participants in the redundancy cluster through its local status table, true multicasting can be replaced by multiple unicast messages. Therefore the requirement of only having unicast communication primitives is not violated.

message	data	type	side	source	destination
SvcCheck	own prio.	UC	DS	BH	MH
SvcChkReply	status; status table	UC	DS	MH	host which prior sent *SvcCheck*
SvcDown	status table	MC	DS	MH	all hosts in table
DnstrTO	own prio.	MC	DS	BH	all hosts in table
DnstrTODeny	-	UC	DS	MH	host which prior sent *DnstrTO*
UpstrTO	own prio.	MC	US	BH	all hosts in table
UpstrTODeny	-	UC	US	MH	host which prior sent *UpstrTO*
SvcTO	status table	UC	DS+ US	BH	all hosts in table

UC=unicast; MC=multicast; DS=downstream
US=upstream; BH=backup host; MH=master host

Fig. 9. Messages exchanged

To ensure proper operations regarding the logic of the distributed system, queuing of asynchronously arriving messages is necessary to preserve them for processing. E. g. in service main state, while processing a *SvcCheck* from one backup host, another *SvcCheck* message arriving from a different host must not be lost. Otherwise the second host might be removed from the redundancy cluster as its associated *SvcReqCheckTimer* might alarm, indicating that this backup host is not available anymore. Figure 10 shows a list containing the main state and the messages to queue there. It should be stressed that this does not include messages which are awaited synchronously as an answer to a prior request message, like the *SvcCheck* / *SvcChkReply* pair in backup main state.

main state	queued messages
backup	DnstrTO, SvcDown , SvcTO
handover	DnstrTO, UpstrTP, SvcTO
service	DnstrTO, UpstrTO, SvcTO, SvcCheck

Fig. 10. Messages to be queued

For messages possibly triggering a direct reply, timeout values have to be specified. Assuming that the roundtrip delay is some magnitudes smaller than the execution time of monitoring procedures, the lower timeout bounds for *SvcChkReply*, *DnstrTODeny* and *UpstrTODeny* are determined by the maximum execution time of customizable procedure executed before sending these messages.

Regarding local alarm timers, the values of *SelfChkTimer* (see service main state) and *SvcCheckTimer* (see backup main state) directly influence how promptly a service failure is detected and how fast the whole system converges to a steady state. Their concrete values depend on complexity, load and duration of the respective checks and the local policy on acceptable service downtime. Therefore they must be determined individually for the concrete service in its particiular environment. To avoid unnecessary load, *SvcCheckTimer*, which triggers remote monitoring in backup main state, should be set indirect proportional to the backup hosts priority. Doing so, a lower ranking host will check service functionality less often than the master's direct successor. Additionally it should not be smaller than *SelfChkTimer* in any case.

4.4 Customizable Procedures

To adopt the generic state machine for a concrete service, several procedures are available for customization. While the redundancy logic remains untouched and is given by the generic state machine as of Sec. 4.2, only the customizable procedures are specific to a certain service.

Monitoring and Testing Procedures: The following procedures are all responsible for monitoring the various parts of the gateway service environment. They all return a boolean status, indicating success. All monitoring procedures must be implemented to deliver positive–definite results, meaning that positive answers must be valid, whereas erroneous negative results are acceptable. By specifying distinct procedures for service and communication monitoring an important requirement from Sec. 2 is fulfilled.

The `downlinkChk()` and `uplinkChk()` procedures are used in several states in the generic state machine. Their aim is to solely check network connectivity relative to respective reference points (Fig. 2).

In backup main state the `remoteSvcChk()` procedure checks the services functionality from the remote side. Checks should be implemented from a clients side of view. This means that tests should act like a client, checking service functionality and connectivity as a whole. In contrast `localSvcChk()` solely is used to test the services functionality on the host currently providing the service. Hence additional service–internal data can be taken into account leading to the possibility of proactive monitoring.

(De–)Activation Procedures: The two pairs of (`de–`)`activateUplink()` and (`de–`)`activateSvc()` are responsible for activating and deactivating the network interface on the uplink side and the service itself. No monitoring/checking actions are carried out within these procedures

Client Related Procedures: To announce a change in the host providing the service to clients two procedures, each for upstream and downstream, are available (`takeoverDnstr()`, `takeoverUpstr()`, `releaseDnstr()`, `releaseUpstr()`). The takeover procedures should implement actions necessary to inform clients of the fact that a new master host is starting service provisioning. The release procedures can be used for activities towards clients necessary on shutdown of the service functionality.

4.5 Summary of Our Solution

In the previous sections we presented a robust and sound solution for management of redundancy in terms of high availability. Hereby, the requirements identified in Sec. 2 are fulfilled as follows.

The separation of logic from actions is accomplished by specifying the logic as a generic state machine (Sec. 4.2), while actions are noted as customizable procedures (Sec. 4.4). Consequently the state machine design makes no assumption whether a later implementation is done as an integrated part of a (new) service application or a standalone addition to an already existing application. Leaving concrete actions to be carried out in customizable procedures, the requirement for independence from specific services and communication techniques is also fulfilled.

Messages used to coordinate takeover — to ensure that only one single master host is present in the redundancy cluster and to define a total priorization order of hosts — are exchanged on communication links present already for service provisioning. Thus no extra heartbeat link is presumed.

Furthermore the requirement of having only a minimal number of active upstream links is fulfilled. During normal service operation, all messages between hosts are exchanged via the downstream side (Sec. 4.3) and uplinks of a backup host are just activated shortly before becoming the master host (see handover main state).

To gain service monitoring from the client's perspective, a backup system actively tests the status of the service (see backup main state), achieving monitoring of connectivity and service functionality as a whole. To enable proactive actions, the active master host additionally does local service monitoring (see service main state), taking internal conditions of the service into account. Especially in case of an integrated implementation, where internal data are more easily observable, monitoring capabilities are substantially extended compared to external monitoring only.

By specifying different procedures for monitoring services and communication links, the distinction of monitoring types is realized. Furthermore independence from technology–specific communication primitives are addressed in Sec. 4.3 and 4.4.

Regarding the work described in Sec. 3 our solution offers some advantages. Compared to VRRP (and "keepalived" as an implementation) our solution is specified in depth and can be applied to a wider range of communication technologies as it makes no specific assumptions here. In case of HSRP it removes the need of a dynamic routing protocol and so our solution can be used in much wider range of scenarios. Also monitoring capabilites are significantly extended as local monitoring as well as monitoring from the client's perspective is provided by our solution. This is also an advantage in comparison to "heartbeat", further the restriction to a two host setup is removed here.

Our state machine pays attention to a number of failures, especially including malfunctions in communication links on backups as well as the master host. But special care must be taken in implementing customizable procedures to avoid some faults still possible. Where an erroneous takeover attempt is prevented by the possibility of vetoing in handover main state, the procedures for (de-)activating services and links must work reliably, because our state machine cannot remedy byzantine faults in these procedures.

5 Application to a Linux–Based Firewall

Applying the generic state machine to solve the single point of failure problem induced by the Linux–based packet filtering firewall of our institute's infrastructure resulted in the implementation described in this section.

Implementing the generic state machine for use in a concrete service consists of three main steps. First, the state machine has to be implemented. Second, the message communication mechanism must be realized according to the scenario's specific circumstances. The third step consists of implementing the customizable procedures.

Universal IP–Service Daemon. Performing the first two steps leads to a universal daemon for IP–based services. It was implemented in C on Linux. For message implementation simply UDP packets were used with an additional symmetric encryption. All hosts in the redundancy cluster share the same key. Configuration is done on file basis. The configuration files need to be distributed manually among the hosts.

Knowing the fact that IP–communications is used, parts of step 3 also can be realized already. Transparency to clients is solved by using a roving pair of IP–addresses, each one for up– and downstream side. Therefore all hosts in the redundancy cluster own a host–specific pair of IP–addresses for message exchange and testing in backup and handover state. In service state the roving pair of IP–addresses is additionally bound to the interfaces (this capability is often referred as "single link multihoming" or "IP–aliasing") by `takeoverDnstr()` and `takeoverUpstr()`. Additionally, to announce the change of the host which owns the roving IP–addresses, a broadcast ping on both sides is sent, giving clients an opportunity to notice the change and re–adapt their IP / Layer-2 translation mechanism (e.g. change ARP table entries in case of Ethernet). One of the main advantages of the roving IP–addresses scheme is that IP–related service configuration remains independent of the actual host, permitting simple configuration replication among backup hosts.

The `downlinkChk()` is implemented as a broadcast ICMP ping on the downstream IP–network, claiming the connection to be healthy when at least one foreign reply arrives. The upstream side is checked in `uplinkChk()` by sending an ICMP ping on the upstream interface to the next hop router.

Having only made the assumption of an IP–based infrastructure the pair of `activateUplink()` / `deactivateUplink()` cannot be determined as they are specific for the underlying layer 2 technology. Their implementation is delegated to external programs (e.g. shell scripts) configurable in the configuration file. The same applies for service related checking procedures (`remoteSvcChk()` and `localSvcChk()`) and service (de–)activation ((de–)`activateSvc()`).

Tailoring for Firewall. For use with our institute's firewall the above described daemon was enhanced by some shell scripts. As Ethernet is used on the upstream side, the scripts for (de–)`activateUplink()` (un–)load the kernel driver modules for the Ethernet–card, realizing a total link deadness on deactivation of the link. Additionally IP configuration of the interface is done here. Service activation is done by enabling IP forwarding, adapting the routing table and setting up the firewall rules via `iptables`. Deactivation is done by the respective inverse actions. Remote service checking in `remoteSvcChk()` is done by an ICMP ping from the downlink interface to the next hop router on the current

firewall's upstream side. Procedure `localSvcChk()` is realized by comparing currently active filter rules to the expected ones.

6 Conclusion and Further Work

Services which are provided and connected to one domain, but also are linked to a foreign domain, so called gateway services, demand special requirements to fault tolerance in terms of high availability. As current solutions are restricted to specific services and/or violate one of the requirements for gateway services, no generic solution for high availability is available.

The introduced generic state machine gives a solution for any gateway service by separating monitoring and failover logic from individual actions specific for concrete services. Its applicability has been proved by implementing a universal daemon for IP–based gateway services and its deployment for a Linux–based firewall.

Moreover this paper could serve as a first starting point for standardization. By reusing the logic of the generic state machine in conjunction with an in–depth specification of the customizable procedures for a concrete service type, a vendor–independent and interoperable failover mechanism for this service type could be specified.

Although an applicable solution is provided, there is room for further improvements. First, a formal verification using model checking techniques is being done right now. Next, security considerations regarding authorization and authentication in message delivery between redundant hosts will be specified at the level of the generic state machine. For this purpose [11,7] and [14] will provide a valuable basis. By now message security topics are left to the responsibility of a concrete implementation. Furthermore, to setup redundancy clusters where hosts serve as a backup for multiple services, the generic state machine will be extended to incorporate the notion of a service type, enabling to distinct between different services. Included are questions regarding load balancing and active feedback of backup hosts to influence their priority ranking in the cluster.

References

[1] A. Cassen. Linux virtual server high availability using vrrpv2. Technical report, November 2001. `http://www.keepalived.org/pdf/LVS-HA-using-VRRPv2.pdf`.

[2] J. Loughney (ed.), M. Stillman, Q. Xie, and R. Stewart. Comparison of protocols for reliable server pooling. Internet Draft, work in progress, IETF, March 2002. `http://www.ietf.org/internet-drafts/draft-ietf-rserpool-comp-03.txt`.

[3] High-Availability Linux Project. `http://linux-ha.org/`.

[4] R. Hauck and I. Radisic. Service Oriented Application Management — Do Current Techniques Meet the Requirements? In *New Developments in Distributed Applications and Interoperable Systems, Proceedings of the 3rd IFIP Int. Working Conference (DAIS 2001)*, Krakow, Poland, September 2001. Kluwer Academic Publishers. `http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/hara01/hara01.shtml`.

[5] H.-G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999.

[6] R. Hinden. Virtual router redundancy protocol for ipv6. Internet Draft, work in progress, IETF, February 2002. http://www.ietf.org/internet-drafts/draft-ietf-vrrp-ipv6-spec-02.txt.
[7] R. Hinden, D. Mitzel, P. Hunt, P. Higginson, et al. Virtual router redundancy protocol. Internet Draft, work in progress, IETF, February 2002. http://www.ietf.org/internet-drafts/draft-ietf-vrrp-spec-v2-06.txt
[8] Reliable Server Pooling (rserpool). IETF Working Group Charter. http://www.ietf.org/html.charters/rserpool-charter.html.
[9] Virtual Router Redundancy Protocol (vrrp). IETF Working Group Charter. http://www.ietf.org/html.charters/vrrp-charter.html.
[10] B. Jewell and D. Chuang. RFC 2787: Definitions of managed objects for the virtual router redundancy protocol. RFC, IETF, March 2000.
[11] S. Knight, D. Weaver, D. Whipple, R. Hinden, et al. RFC 2338: Virtual router redundancy protocol. RFC, IETF, April 1998.
[12] T. Li, B. Cole, P. Morton, and D. Li. RFC 2281: Cisco hot standby router protocol (hsrp). RFC, IETF, March 1998.
[13] Linux Virtual Server Project. http://www.linuxvirtualserver.org/.
[14] A. Robertson. Linux-ha heartbeat system design. In *4th Annual Linux Showcase & Conference (ALS 2000)*, Atlanta, USA, October 2000. http://www.linuxshowcase.org/2000/2000papers/.
[15] R. Stewart, Q. Xie, K. Morneault, C. Sharp, et al. RFC 2960: Stream control transmission protocol. RFC, IETF, October 2000.
[16] R. R. Stewart, Q. Xie, and M. Stillman. Aggregate server access protocol (asap). Internet Draft, work in progress, IETF, May 2002. http://www.ietf.org/internet-drafts/draft-ietf-rserpool-asap-03.txt.
[17] M. Tuexen, Q. Xie, R. Stewart, M. Shore, et al. Architecture for reliable server pooling. Internet Draft, work in progress, IETF, April 2002. http://www.ietf.org/internet-drafts/draft-ietf-rserpool-arch-02.txt.
[18] M. Tuexen, Q. Xie, R. Stewart, M. Shore, et al. RFC 3237: Requirements for reliable server pooling. RFC, IETF, January 2002.
[19] W. Zhang. Linux virtual server for scalable network services. In *Proceedings of Ottawa Linux Symposium 2000*, July 2000. http://www.linuxvirtualserver.org/ols/lvs.ps.gz.