2ND QUARTER 2008, VOLUME 10, NO. 2

IEEE COMMUNICATIONS SURVEYS

www.comsoc.org/pubs/surveys

FAULT TOLERANCE FOR HIGHLY AVAILABLE INTERNET SERVICES: CONCEPTS, APPROACHES, AND ISSUES

NARJESS AYARI AND DENIS BARBARON, FRANCE TELECOM R&D
LAURENT LEFEVRE AND PASCALE PRIMET, INRIA/LIP

ABSTRACT

Fault-tolerant frameworks provide highly available services by means of fault detection and fault recovery mechanisms. These frameworks need to meet different constraints related to the fault model strength, performance, and resource consumption. One of the factors that led to this work is the observation that current fault-tolerant frameworks are not always adapted to existing Internet services. In fact, most of the proposed frameworks are not transport-level- or session-level-aware, although the concerned services range from regular services like HTTP and FTP to more recent Internet services such as multimodal conferencing and voice over IP. In this work we give a comprehensive overview of fault tolerance concepts, approaches, and issues. We show how the redundancy of application servers can be invested to ensure efficient failover of Internet services when the legitimate processing server goes down.

High availability and fault tolerance are key issues that have been considered in different areas. Indeed, failures can happen due to core network congestion, end server overload, server hardware or software fault, denial of service attacks, and so on.

In Internet routing, link as well as router availability were addressed through failure-aware routing mechanisms. In Internet servers, data, node, and service availability were invested in greatly. This survey focuses on fault tolerance for Internet servers.

Fault-tolerant frameworks take advantage of resource redundancy to provide reliable execution of a service when its legitimate processing server goes down. They are built on two key concepts: fault detection on one hand and fault recovery on the other. Moreover, they need to meet different challenges related to the robustness as well as the performance of the fault handling procedures. The first challenge is illustrated in Fig. 1 by the resources axis. It measures the resource consumption inherent in the granularity of the failure detection and recovery mechanisms in terms of CPU, memory, bandwidth, I/O, and so on. The second challenge is illustrated in Fig. 1 by the fault model axis. It measures the strength of the fault model in terms of fine-grained failover, client transparency, failure detection protocol robustness, and so on. For instance, the model should ensure that only one instance of the service is processing the client requests at a given time. It may be more or less compliant with off-the-shelf commercial or legacy applications and hardware. The third challenge is illustrated in Fig. 1 by the performance axis. It deals with the impact of the failure recovery procedure on the end-to-end quality of service (QoS) of the highly available service during both failure and failure-free periods. This measure is all the more relevant when the handled applications are QoS-sensitive. Representative parameters that would help quantify this impact include fault detection latency, replica launch latency, and fault recovery latency. Other application-dependent metrics can be considered as well such as bandwidth, latency, and loss rate. These metrics vary greatly in terms of their dimension and acceptable ranges for each class of applications.

ADDITIONAL HIGH AVAILABILITY REQUIREMENTS FOR CLUSTER-BASED ARCHITECTURES

Internet server clustering has been widely used to improve the scalability of the rendered services under heavy load. A cluster typically consists of a set of networked off-the-shelf servers that offer a single system image while providing additional processing capabilities. Cluster-based architectures take

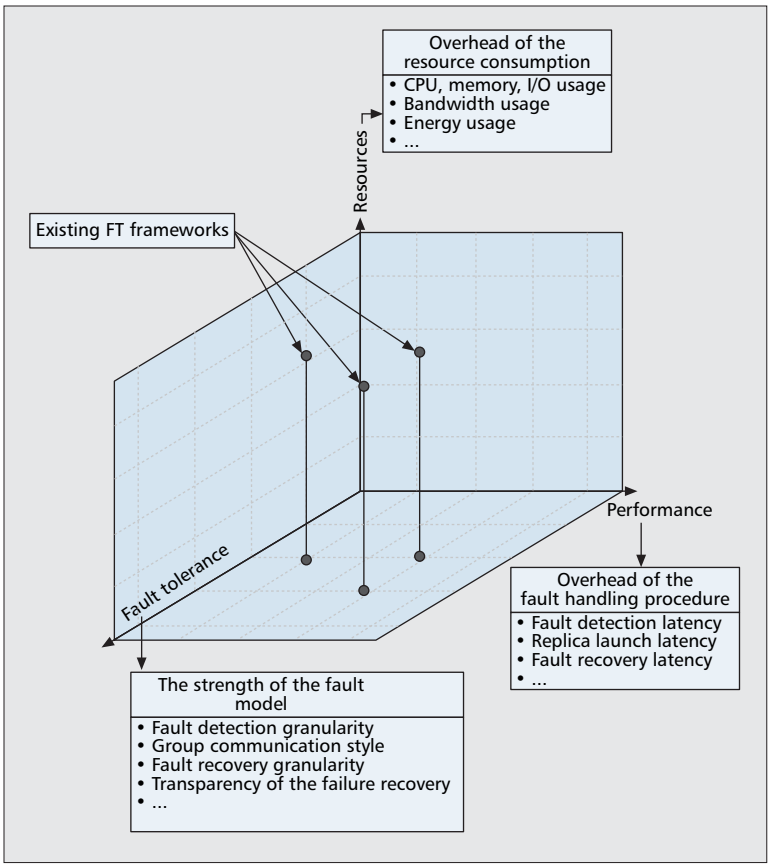

Figure 1. *Constraints of fault tolerance frameworks.*

advantage of resource redundancy to meet both scalability and high availability requirements. They create fault recovery models to provide highly available service frameworks having no single point of failure (SPoF) either at the entry or inside the cluster. They also provide a means to recover from the failure of a legitimate cluster resource on an available replica.

Redundancy can follow two different scenarios. First, in the passive scenario (Fig. 2a), only one node operates as the master server at a given time and processes the offered network traffic. All the other nodes are the master's replicas. Should the master node fail, one replica is elected to recover the service.

The second scenario is the active scenario (Fig. 2b). where all the replicas operate concurrently, sharing the offered network traffic. When a processing node fails, one replica is elected to take over the traffic already assigned to the failed node.

In a typical cluster the network traffic is first offered to the cluster head(s), where a load balancer is instantiated to divert incoming requests to an appropriate processing server inside the cluster. Load balancers can be designed as stateful or stateless devices. Stateful devices maintain an in-memory mapping between incoming flows and the associated available servers. In the event of failure of the legitimate entry point to the cluster, fault-tolerant frameworks require means to recover the legitimate load balancer's association table so as to ensure the survivability of both already established flows and lately offered ones.

On the other hand, since deliberate or unplanned failures can occur inside the cluster, fault-tolerant frameworks need to provide a means to keep on processing already accepted sessions until they normally terminate in the event of a planned failure at the legitimate processing server.

However, recovering a service when the entry point to the cluster or processing node crashes is not a simple procedure. Indeed, depending on the end-to-end session properties, different network-level operations may be considered.

Let us recall that an application typically manages several client sessions. Each client session can span over a single flow or over multiple flows. The involved flows can be used for either signaling or data exchange between the client and the server throughout the session lifespan (Fig. 3).

Typical examples include regular services such as file transfer, interactive Web sessions, e-commerce services, remote login, and email, as well as most recent Internet services such as video streaming, multimodal conferencing, live broadcast of events, video-on-demand services, and voice over IP. These Internet services have different high availability requirements and constraints in terms of allowed packet loss ratio, delay sensitivity, and transport- and session-level awareness (Table 1).

Network layer failover is not enough to transparently recover the already established sessions for most of these services. In particular, critical TCP-based services require robust transport-level failover as avoid interruption of the already established TCP flows in case of failure of the legitimate processing server. A first example can be a simple connection to a database server. When the connection terminates, all the uncommitted transactions handled over this connection are aborted, requiring the user to explicitly initiate them over a new established TCP flow. A second example references the Border Gateway Protocol (BGP) routing process. BGP is a TCP-based protocol deployed on border routers to connect between different administrative Internet domains. When a connection is broken with a peer, the router first floods the network with messages to propagate the failure of the peer BGP router. A second flood is required to propagate its recovery. A third example concerns voice over IP applications using H.323. H.323 signaling uses TCP at its transport protocol. Hence, when the flow is interrupted, the whole end-to-end signaling is broken.

One of the critical challenges that led to this work is the observation that the current fault tolerance frameworks are neither transport-level- nor session-level-aware. This article

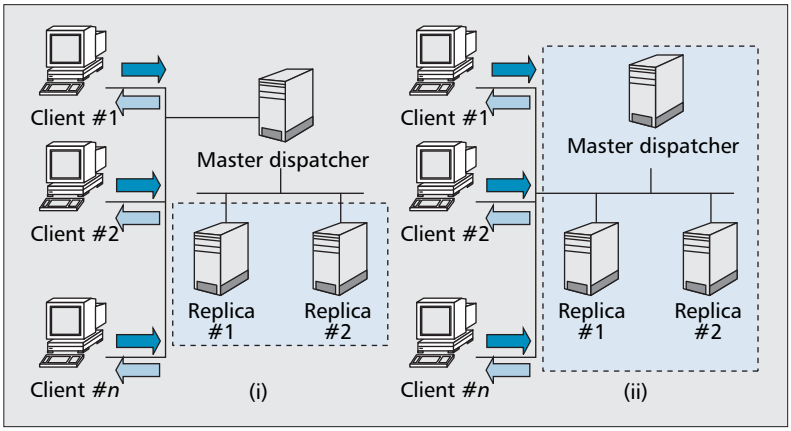

Figure 2. *a) Passive redundancy; b) active redundancy.*

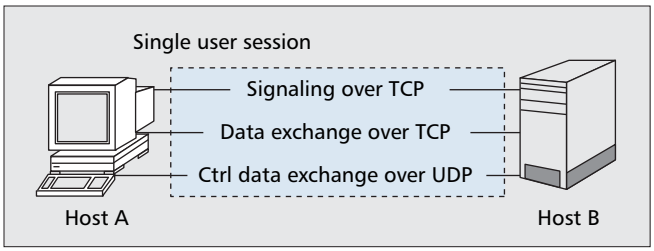

Figure 3. *A session abstraction.*

gives a comprehensive overview of fault tolerance concepts, approaches, and issues. We show how the redundancy of application servers can be used to ensure efficient failover of Internet services in case of failure of the legitimate processing servers.

The Organization of the Survey

The rest of this survey is organized as follows. We give an overview of the fault models and types. We describe the major failure detection protocols and mechanisms, and outline their applicability in both synchronous and asynchronous systems. We revisit the states that must be replicated during failure-free periods. We address the methods that can be used to replicate these states, and discuss their limitations and performance during failure-free and failure periods. We deal with the failure recovery mechanisms and provide a comprehensive overview of the different operations required to provide fault tolerance support at the network, transport, session, and application levels. Finally, we conclude by summarizing the lessons learned and positioning the already discussed mechanisms against large-scale distributed replication and failure recovery approaches such as those advocated by the domain name system (DNS) and peer-to-peer (P2P) networks.

Internet Server Fault Models and Failure Detection Approaches

This section deals with failure concepts. First, we recall the fault models and distinguish between the possible types of faults for Internet services. Second, we give an overview of failure detection approaches. Finally, we outline the applicability of these approaches to synchronous and asynchronous systems.

Fault Types and Models

Failures can occur due to software or hardware malfunctioning, at the client or server side, or in the network path separating the sender from the receiver. Client-side faults concern the client device. Network-side faults include the corruption, delay, reordering, duplication, and loss of packets crossing the network due to possible link failures or persistent network congestion. Server-side faults result in the silence or malfunctioning of the processing server, for instance, due to the failure of one or many of its processes, persistent overload, or failure of some of its components (processor damage due to radiation or electrical noise, memory buffer overflow, power supply interruption, storage device corrupted blocks, network interface buffer overflow, etc.).

A system may exhibit arbitrary faults [1–3]. Faults can occur according to two different models. The first model references the worst case and identifies the *Byzantine* fault class [4]. A fault is said to be Byzantine when it occurs arbitrarily and maliciously, causing the system to behave incorrectly. A typical example of a Byzantine server fault consists of transmitting incorrect messages from a faulty server. For instance, a TCP-based faulty server may send a TCP RST or FIN datagram, causing the remote client to abandon or close the connection. When the client-side connection state is lost, the TCP connection can no longer be transparently restored. The second model references the best case and defines the *fail-stop* fault class. Fail-stop faults or *perfect failures* have a deterministic impact on a subsystem component and cause it to die silently. The faulty component behavior is inactivity during failure. Hence, fail-stop faults are easy to detect.

In general, fault detection is accomplished by means of some simple heartbeat-based protocols. Some fault injection research [5, 6] has shown that in practice, most faults obey to the fail-stop model, meaning that if any component of the node fails, the entire node is assumed to crash.

In the rest of this article we focus on the server side failures and use the terms fault, failure, and crash interchangeably.

Application	Application protocol	Transport protocol	Multiple-flow-based	Loss-sensitive	Delay-sensitive
File transfer	FTP	TCP	Yes	No	No
Email	SMTP	TCP	Yes	No	No
Web	HTTP	TCP	No	No	No
Remote terminal access	TELNET	TCP	No	No	~ ms
Remote file server	NFS	TCP or UDP	Yes	No	No
Real-time audio-video	HTTP/FTP RTP/RTCP SIP/H.323	TCP UDP TCP or UDP	Yes	Yes	~150 ms
Voice over IP	SIP/H.323	TCP OR UDP	Yes	Yes	~150 ms
Instant messaging	SIP/H.323	TCP	No	No	Yes/no

Table 1. *High availability constraints for representative applications.*

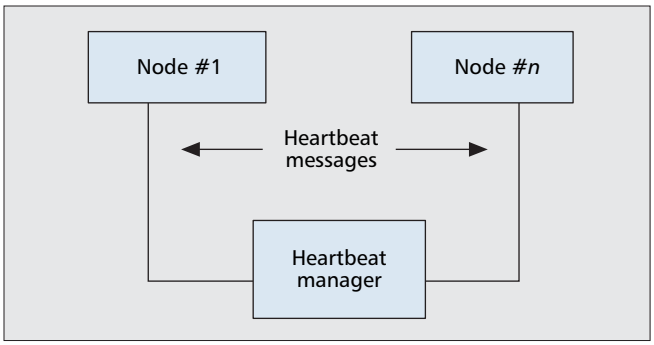

Figure 4. *Heartbeat monitoring abstraction.*

Failure Detection Approaches

Failure detection is the first building block in a fault-tolerant framework. Two properties are desirable for this component. First, it should detect failures as soon as they occur so that the framework can quickly trigger the failure recovery procedure. Second, it must be robust enough to ensure that only one error-free instance of the service is running at once.

In the following we first assume fail-stop failures where the absence of proof of aliveness is taken as evidence of failure. Then we give an overview of the software and hardware failure detection approaches. Second, we discuss the applicability of the perfect failure detection approaches to asynchronous systems.

Perfect failure detection [2] is based on the explicit and periodic exchange of heartbeat messages between replicas during error-free periods (Fig. 4).

Heartbeat-message-based monitoring obeys either the pull model or the push model. In the pull model (Fig. 5) the monitor process periodically asks the monitored host or process for availability information.

In practice, the monitor process arms a fixed or an adaptive timeout on receipt of an availability message from the monitored host. Assuming a very small network packet loss rate, consecutive losses of heartbeat replies mean the failure of the monitored host. The exchange of ICMP ping messages is a simple pull-based protocol used for host monitoring purposes.

In the push model (Fig. 6) the monitored process periodically sends availability information to the peer listening monitor process.

δ is chosen such that it approximates the sum of the heartbeat request and reply transmission and processing times, while n is a configurable parameter denoting the number of retries prior to declaring the suspected node as in failure.

Heartbeat messages can be exchanged over one-to-one unicast channels, or one-to-many broadcast or multicast channels. Nonetheless, for the sake of less network bandwidth occupancy as well as better scalability, the exchange of multicast-based heartbeat messages is strongly recommended [7], and heartbeat messages are typically small (~150 bytes).

Cluster membership protocols are often based on heartbeat message exchange. A new host announces its membership to the cluster by periodically sending a heartbeat message to the monitor process. The absence of heartbeat messages from a given node causes its removal from the list of the active nodes within the cluster.

The above described protocols are generally used to detect a node or link failure. However, an error may occur at a smaller granularity than hosts, causing, for instance, one process to crash on the partially unavailable host while all the other processes still operate correctly. However, the implementation of error detection protocols at smaller granularity such as at the process level seems more complex and more costly. Watchdog timers are an inexpensive method of process and node error detection. They can be used to monitor user space processes such as Web servers, database servers, swap memory, or network interfaces. Their basic idea is that the process being monitored must reset a timer before it expires. Otherwise, it is assumed to have failed. Since error detection is based entirely on the time between timer resets, only processes with relatively deterministic runtimes can be monitored. The major limitation of this approach is that it only provides an indication of possible process failure. Indeed, a partially failed process may still be able to reset the timer. Moreover, the fault type coverage is limited, as neither the data nor the results handled by the monitored process are checked.

Different implementations of watchdog timers exist covering both hardware and software timers. The Linux operating system provides full support for watchdog circuits. A watchdog device can be connected to a system to allow the kernel to determine whether a given process hangs. Softdogs are on the other hand internal timers updated as soon as a process writes to `/dev/watchdog`. A possible high availability oriented use of softdogs is the monitoring of the heartbeat-based monitoring process itself.

Perfect failure detection is an important concept necessary to build reliable and highly available frameworks. In particular, when the replicas share access to a storage device such as a database or file system, *read* and *write* operations on the stored data must be committed once by the available legitimate processing node to keep the whole framework in a consistent state. Several approaches have been proposed to provide a processing node with exclusive access to a shared resource. These approaches fall into two categories: the quorate software-based class [8] and the fencing hardware-based class [9].

The quorate-based class [8] involves the concept of quorum, which means voting sufficiency. Its basic idea is to assign a quorum to at most one single elected replica in the cluster. This approach applies well during failure-free periods, but

```
The monitor process
function failure_detector(Host h)
  Send {Heartbeat_Hello} to the receiver
  Wait δ
  On receive {Heartbeat_Reply}
        return up;
  After n*δ
        return crashed;
The monitored process
procedure Availability_announce()
  Forever
    On receive {Heartbeat_Hello}
      Send{Heartbeat_Reply} to the monitor
```

Figure 5. *Pull-based heartbeat monitoring.*

```
The monitor process
function failure_detector(Host h)
  On receive {Heartbeat_Hello} from h
        return up;
  After n*δ
        return crashed;
The monitored process
procedure Availability_announce()
  Forever
      Send{Heartbeat_Hello} to the monitor
  Wait δ
```

Figure 6. *Push-based heartbeat monitoring.*

performs less well when a non-perfect failure happens. Indeed, if the failed node resumes its activities while an elected node has taken over the service, the accessed shared data is no longer in a consistent state.

The fencing based class [9] provides fencing functions following two different models. The first model is system-reset-based [10]. It achieves perfect failures by killing the suspicious node. By the way, Shoot The Other Node In The Head (STONITH) [11] uses a network power switch to remotely shut down or reboot the failed node. Watchdog timers can also be used to reliably initialize a hardware reset on the suspicious node in a STONITH-like manner.

The second model addresses the limitations of system-reset-based solutions by providing resource-based fencing [12] where the resource reservation process is embedded within the shared resource itself. Hence, resource-driven clusters assume that the storage is qualified to perform the reservations. For instance, SCSI reservations [12] are often used to ensure that SCSI storage is exclusively owned by a single node.

In asynchronous systems perfect failure detection is hard to achieve [13]. Indeed, purely asynchronous systems are characterized by having no finite upper bound on the transmission delay of messages or the processing delay. Thus, the use of time-based failure detectors is not suited to asynchronous environments.

Service Replication Concepts, Approaches, and Issues

The replication concept aims to allow the recovery of a service by replicating each of its related states on a number of replicas. Should the primary processing server fail, the traffic is taken over by an elected backup node.

Lot of approaches exists to coordinate between replicas during failure-free periods. Among these approaches, we name the leader/follower [14], active replication [15], checkpointing [16], message logging [17], and hybrid approaches [18]. Challenges to state replication include transparency, low overhead, and consistency.

Transparency means that no changes are required at the client or server side to provide state replication. Overhead refers to the cost of the replication process during failure-free periods (e.g., in terms of additional delay and resource usage). Consistency means that replicas maintain the same view of the replicated state. However, for highly nondeterministic applications such as multithreaded, random-number-based, or time-based applications, keeping replicas in a consistent state is not a trivial problem. The choice of a given replication policy should achieve the best trade-off between these constraints and the characteristics of the highly available service.

Before dealing with the replication mechanisms, let us recall the states that must be properly preserved among redundant nodes. Servers typically maintain a service state for each handled client session. This state is required to sustain communication with the client. A single client session can involve one or multiple flows for the signaling and data exchange between the client and server.

Client-transparent fault tolerance means that service is recovered without interrupting already established sessions and avoiding the explicit participation of the client side in the failover operation.

In order to achieve a transparent failover, a consistent server side state must be available at the backup nodes for each already established session. In case of failure of the primary processing node, a replica is elected to recover the already established sessions.

For a given client session, the server side state includes all the information required to identify the service. Hence, it includes the flow- as well as application-level states associated with the handled session.

The flow state is the network abstraction defining the communication channel used by the application to communicate with the remote hosts. It is defined at the transport level as the address known and used by the client to access the service. For services built on top of IP such as ICMP, the service identity is simply the destination IP address used by the clients to reach the server. For services built on top of the transport layer, the service identity also includes the TCP or UDP destination port number identifying the service running on the target host.

In addition, servers typically maintain some state while providing the service for both reliable and unreliable communications. For connection-oriented communications such as those based on TCP, the server maintains a transport-level state at each endpoint. This state is used to implement the functionalities provided by the protocol such as reliable packet delivery, flow, and congestion controls. Transport-level states are updated by the kernel upon the arrival or departure of a datagram. For instance, a TCP state includes the sequence and acknowledgment numbers, advertisement window value, and so on. More important, it includes all the client packets that have been acknowledged by the server and are not yet processed by the application layer as well as all the data to be sent to the client. TCP stores the client packets received by the server in its receive queues. Since the client is no longer able to retransmit the acknowledged data, it is important not to consequently lose this data as a result of a server fault. On the other hand, connectionless communication protocols such as UDP do not provide any functionality guaranteeing the reliable execution of a service. The flow state is restricted to the socket structure, which basically describes the end-to-end flow identifiers.

Several flows may be associated with the same user session. These flows may be required for signaling and data exchange throughout the session lifespan. A session state is the abstraction provided for the application layer to describe the association between the communicating endpoints. This association includes the flow states associated with the concerned session as well as the associated application level state, if defined. In contrast to the open systems interconnection (OSI) communication model, which provides functions for setting up, tearing down, interrupting, or resuming a session from an agreed synchronization point, the TCP/IP stack provides no explicit support for session handling. In order to provide session-aware fault tolerance, a fault tolerant framework must be able to properly identify and replicate all the states associated with a highly available session.

While a connection state is restricted to issues including sequence, error, and congestion control for a particular end-to-end connection, an application state provides the framework for additional application-level conversations. An application is either *stateful* or *stateless*. A stateful application maintains an explicit user-level application state. This state includes the information needed to perform application-level services such as compression, authentification, and multiparty conferencing. Typically, the application-level server side maintains specific protocol states required to correctly handle the incoming requests for the entire lifetime of the application, across multiple client requests and connections. Three-tier architectures keep most of the application state on a separate server host such as a remote database server (Fig. 7). When the time required for reading the application state from the storage device is less than the time needed to transmit the

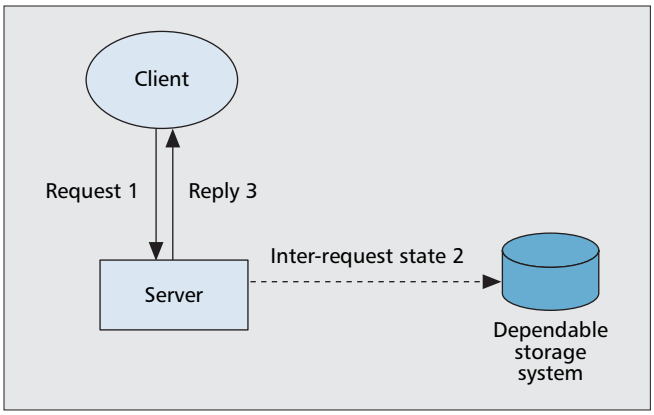

Figure 7. *Application state on a dependable storage system.*

request on the wide network, this approach incurs a small latency in the end-to-end delay.

Once the primary server fails, the service is recovered on an elected replica (Fig. 8). In practice, the network identity takeover at the replica is not enough to prevent breaking the already established flows. For instance, the replica needs to re-establish the connection-oriented flow with the remote database server before resuming read and write operations on consistently stored application states. For stateless applications such as telnet or HTTP, each application-level request looks to the server like a distinct self-contained request. This means that it is processed independent of its predecessors, and there is no application-level client state to be preserved at the server side. Therefore, session-aware failover requires only to properly recovering the states of each flow involved in the session. However, when the stateless application server is extended to provide extra-services such as authentification or user connection tracking, the user session spans over a *virtual* session. At the application level, the virtual session is implemented using cookies, SSL session identifiers, and so on. More often, these values are used at the server side to index the stored data and preferences on a per-user basis. Well-known means to store these states include in-memory processes such as Java Servlets and Active Server Pages or flat files on shared file systems (CODA, NFS, etc.), as well as more sophisticated database storage systems such as Oracle or MySQL systems.

The Semi-Active Replication Approach (Leader/Follower Replication)

The basic idea of the semi-active replication approach, also referred to in the literature as leader/follower replication [14], is to have a replica perform each nondeterministic action first. Then the leader notifies each follower with the results of the action that was performed successfully. Finally, each follower uses this information to keep itself in a consistent state compared to the leader (Fig. 9).

The semi-active replication approach performs well for read-only files but seems less appropriate when the service involves files to be concurrently modified by other processes, using, for instance, the *write()* system call. This approach also performs very poorly when keeping the replica states consistent requires transfers of large volumes of information. Finally, this approach is not suited to replicate the state of busy servers because their throughput might be severely affected.

The Active Replication Approach

The active replication approach [15] requires all the clients to receive and concurrently process the offered network traffic. Its main objective is to ensure that all replicas maintain the same consistent states while guaranteeing that only one server is replying to clients at a given time (Fig. 10).

While the leader does not need to forward any data to the follower(s), further processing is required to ensure that the follower(s) perform the same processing as the leader and produce the same data as well. For instance, an active replication system may require the leader to notify the follower(s) of how many bytes it has successfully read or modified. Such an operation seems necessary when the replicas have different processing capabilities or the upper layer service exhibits nondeterministic behavior.

A building block for each active replication architecture deals with the way each replica reliably receives the offered network traffic sent to the leader. Different techniques can be used for that purpose. The first technique is based on protocols originally considered for setting up group membership. In particular, we name the atomic multicast protocols [19] used to reliably deliver the offered network traffic to a group of replicas. However, this approach requires that both clients and servers be reliable-multicast-aware, that is, able to send and receive data via atomic broadcast/multicast-aware application programming interfaces (APIs).

An alternative solution consists of delivering the traffic exchanged between the clients and the replicas to an intermediate gateway or proxy that would reliably perform one-to-many message delivery to the replicas on one hand and many-to-one message delivery to the clients on the other hand. The major drawback of this approach is that it implies additional overhead due to the cost of translation of the offered unicast messages to outgoing multicast messages during failure-free periods. Moreover, this approach adds a potential single point of failure, which corresponds to the gateway or proxy.

An alternative approach suggests having each replica passively intercept and process the traffic originally offered to the

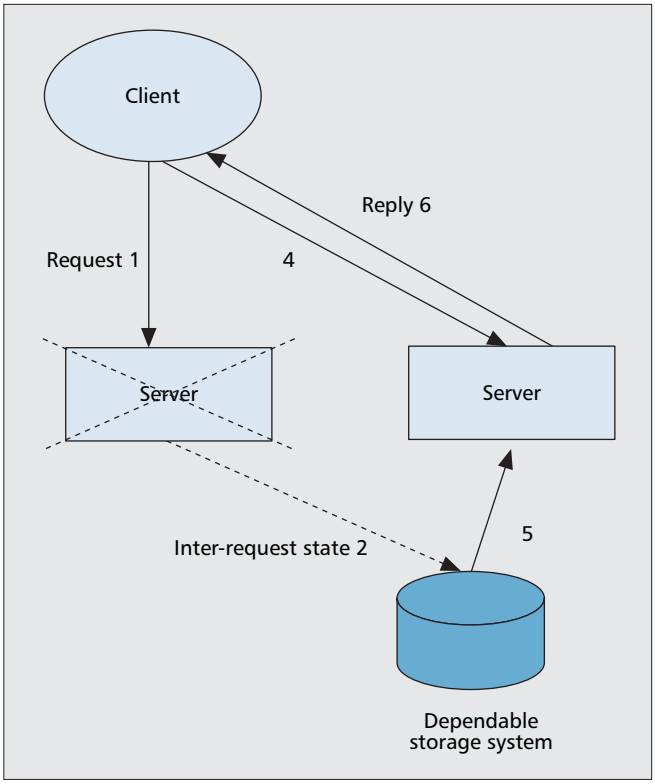

Figure 8. *Application state recovery from a storage system.*

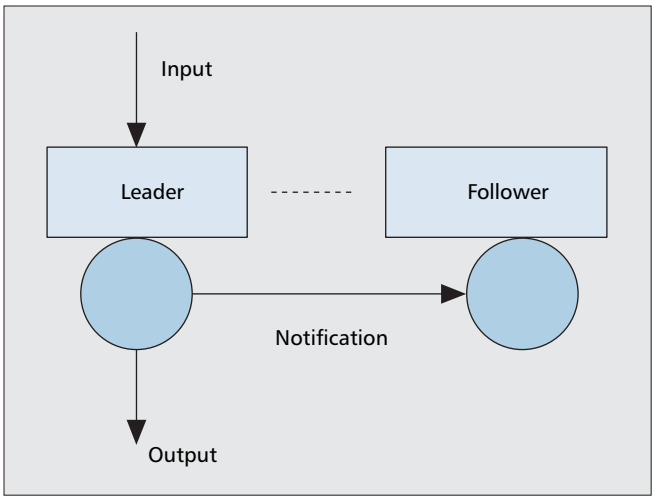

Figure 9. *The semi-active replication approach.*

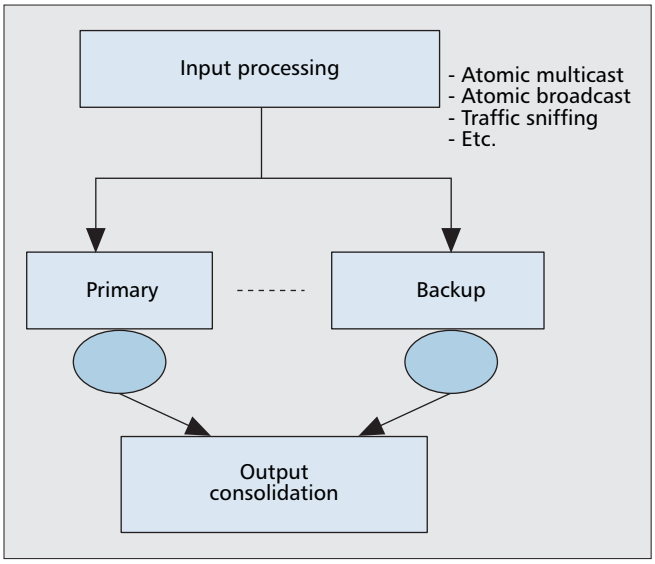

Figure 10. *The active replication approach.*

primary server. This approach requires the follower to be fast enough to keep up with the leader processing hardware and software capabilities to avoid penalizing the performance of end-to-end highly available communication during failsafe periods. Different topologies can be used to allow this configuration. These are the router topology, Address Resolution Protocol (ARP) proxy topology, and shared Ethernet topology.

In the router topology (Fig. 11) the follower is configured as the router for the leader host. The router has at least two interfaces; one of them listens to the Ethernet segment that the router shares with the leader.

The idea is to add a routing rule to the router's routing table that all packets having as destination IP address the leader's IP address should be first sent to the follower, which is set up to forward these packets to the leader. Similarly, the leader has a routing rule according to which all outgoing packets are first sent to the follower, which will forward them to the router. While this topology allows the follower to capture all traffic flowing between the client and the leader, its use remains uncommon in practice since it may be impossible to change the router's routing table for a given domain.

The proxy ARP topology (Fig. 12) assumes that the leader and follower are connected to the same LAN. When the gateway receives a packet having as destination IP address the leader's IP address, it issues an ARP request asking for the corresponding physical address. The follower, configured to perform ARP spoofing, replies with its own network interface hardware address. Once it receives the traffic, the follower forwards it to the leader through a second network interface.

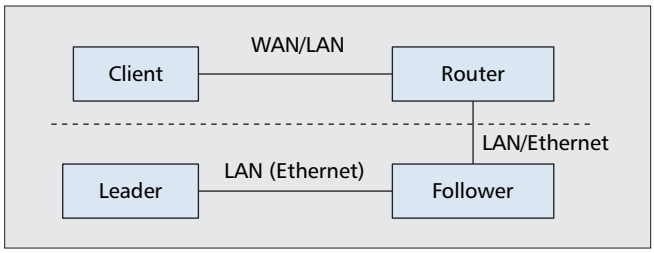

Figure 11. *The router topology.*

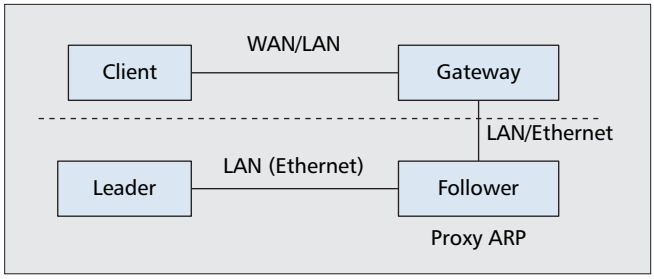

Figure 12. *The proxy ARP topology.*

The main drawback of this topology is that it incurs an additional delay to the end-to-end communication during failsafe periods. This delay corresponds to the time needed by the intermediate node to issue the forwarding decision.

The Ethernet topology (Fig. 13) is more flexible than the previous topologies and seems particularly interesting when the ARP proxy approach is prohibited in the network for security reasons. Its basic idea is to decouple the follower node from the proxy as well as from the router. However, it assumes that the follower is connected to the same shared Ethernet segment as the leader. The follower listens to the unicast, multicast, and broadcast traffic exchanged over the Ethernet medium. In particular, it listens to the traffic flowing between the client and the leader.

However, for both performance and traffic isolation, most modern Ethernet topologies are switch-based. The broadcast medium is replaced by a crossbar device that prevents one host from tapping the traffic destined to another host. Hence, when the primary and backup hosts are connected to the same Ethernet switch, the switch traffic isolation property implies that the follower will not be able to listen to the traffic destined to the other hosts in the network. Different solutions can be engineered to allow passive network tracing and interception. The first solution consists of using port mirroring capable switches, which provide the administrator with the capability of forwarding or mirroring the traffic flowing to or from one or many of its ports to some or all of the other ports of the switch. The second alternative consists of using multicast or broadcast enabled topologies to allow a backup node to listen to traffic originally destined to a primary processing server (Fig. 14).

The basic idea of the multicast-based topology is to statically associate both the primary and backup nodes to the same link layer group (by associating a single Ethernet multicast address to each replica). A standard switch would then automatically copy the incoming traffic destined to the primary node to all its outgoing ports. In particular, the backup node taps the full duplex traffic flowing between the primary

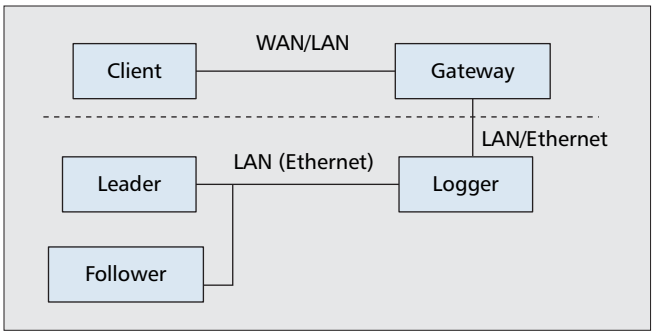

Figure 13. *The Ethernet-based topology.*

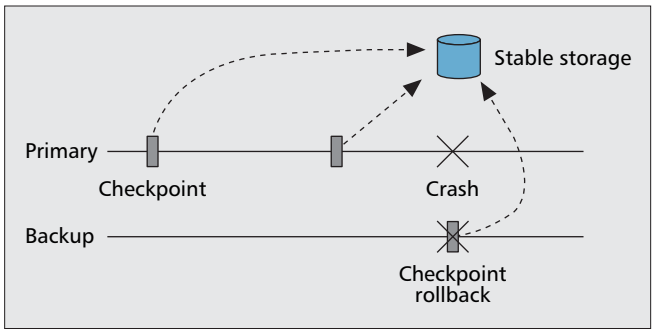

Figure 15. *Abstraction of checkpointing operations.*

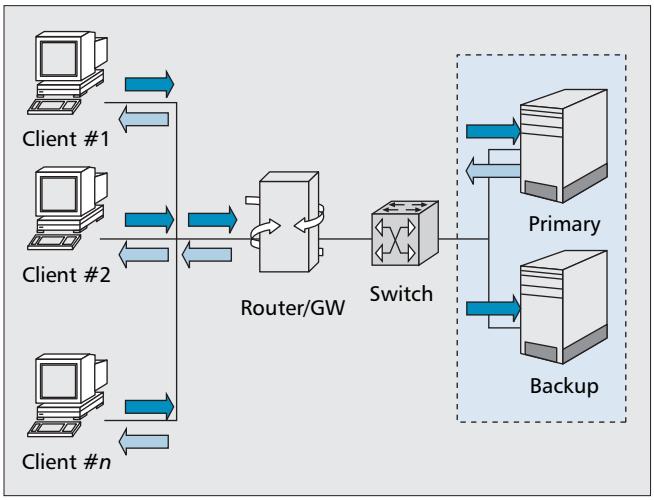

Figure 14. *Passive traffic interception in a multicast/broadcast topology.*

node and the clients. The major drawback of this solution is its high packet rate due to packet duplication, possibly leading to overload of both the backup's NIC and the LAN links. Moreover, since the backup is involved in more tasks, such as listening, filtering, and modifying the intercepted traffic, it is possible that it fails to intercept some packets exchanged between the primary and the clients. This issue should be considered by the failover mechanism as well. A first solution consists of recovering the lost packets from the primary node. A second alternative consists of recovering the lost packets from an alternate logger or gateway machine, which would keep in-memory windows of the packets flowing between the clients and the primary node for a finite period of time.

The Checkpointing Approach

In the checkpointing approach [16], the server state is periodically copied either to a standby server(s) or to a stable storage (Fig. 15).

In the event of failure of the primary node, the most recent checkpoint is recovered, and the processing resumes using the restored state. Different checkpointing approaches exist. They differ in terms of their frequency and completion time. The most aggressive checkpointing approach is *incremental* checkpointing. It aims to maximizing the consistency of the replicated states by performing checkpoints each time a critical state change occurs at the primary node. The major drawback of this approach is its cost in terms of CPU consumption at the primary node as well as in terms of added latency to the end-to-end communication. A second approach is *time-line based* checkpointing where a state is checkpointed each period of time. The time-to-checkpoint value depends on the measured failure frequency. A small value leads to very frequent state checkpoints and important overhead during failure-free periods. However, it offers fewer rollback operations in the event of failure. Other approaches suggest randomly checkpointing the replicated states or reducing the checkpointing overhead by guessing the optimal number of checkpoints that should be taken for a given system.

The Message Logging Approach

Message logging [17] can be applied at the flow or application level. Its main idea is to redundantly store or log all the messages delivered to the primary server on stable storage or a replica (Fig. 16).

For reliability purposes, a message would not be processed until an acknowledgment is received from the replica or storage device confirming that the message has been successfully stored.

During failsafe periods, replicas are idle. Once the primary server fails, the logged messages are replayed and reprocessed on the elected replica. Different approaches exist for message logging. *Pessimistic* message logging [20] consists of logging a message into a stable storage as soon as it is received. *Dependency-based* message logging [21] proposes to copy each received message into a volatile log space. This space will be flushed into stable storage once it becomes full. *Optimistic* message logging [21] also copies the incoming message into a volatile log space, but proposes to flush it on stable storage periodically or when the number of logged messages reaches a given threshold.

The Hybrid Approaches

Since frequent state checkpointing is costly, alternatives to it have been proposed. A first approach combines checkpointing with message logging [18]. The idea then consists of logging all the messages received since the last checkpoint. When a fault occurs, the most recent checkpointed state is recovered, followed by replaying all the messages logged since the last checkpoint to reach a consistent prefailure state. Compared to checkpointing each state change, the overhead during failure-free periods is reduced. However, recovery time is increased. A second approach [18] addresses the nondeterminism at the application level and suggests actively replicating the flow level state while logging the application level messages.

Discussion: A Comparison of Replication Approaches

The choice of one replication approach should take into account the constraints of the replicated service. In particular, it should consider the sensitivity of the service to packet loss and end-to-end delay as well as the required failover model characteristics in terms of transparency, performance, and consistency, during both failure-free and failure periods. In

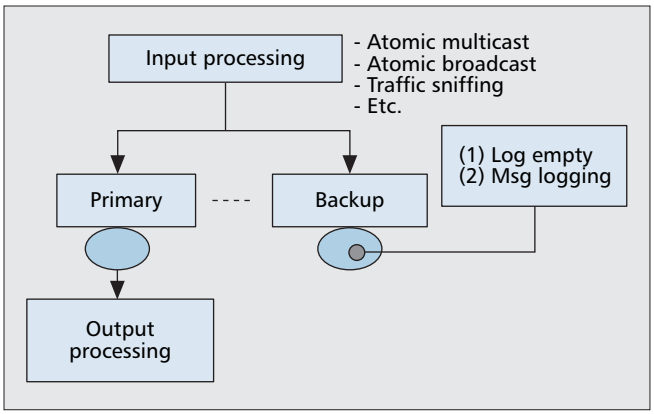

Figure 16. *Abstraction of the message logging operations.*

Table 2, we compare the above described replication approaches according to resources usage, provided level of consistency, overhead during failure--free periods, and performance during failures (e.g., in terms of recovery time).

Overhead during failure-free periods is defined as the delay incurred by the primary server during failsafe periods. It can result from waiting for a replica's acknowledgments before the primary sends back replies to clients. The recovery time is the time needed to bring the backup node to a consistent state, similar to the prefailure state of the primary node. During this period service is unavailable, meaning that the processing of already active flows is either frozen or interrupted while new offered traffic is rejected. Hence, this period should be as short as possible to efficiently mask failures when time-sensitive services are involved.

Both the active replication and message logging schemes require the application servers to be deterministic enough to result in the same output when provided with the same input. However, in practice, servers may exhibit nondeterministic behaviors due to multithreading or the use of time- or random-number-based variables. The active replication approach seems to offer the best recovery time value while involving more resources during failure-free periods. On the other hand, the message logging approach requires a large failover time since all the messages received prior to the failure must be replayed.

Failure Recovery Approaches and Issues

This step is required following failure detection. It consists of resumption of normal operations of the failed server on an available backup node. Its objective is to increase both the availability and reliability of the replicated service. Availability consists of allowing lately arriving requests to be processed transparently to the client side in the event of failure of the legitimate processing server. Reliability of the service requires avoiding interruption of already established sessions during the failover procedure.

The failure recovery procedure requires first electing a backup node among a set of available replicas. In order for the backup to transparently take over the offered network traffic, it should first take over the identity of the failed host. This step, known as *network identity takeover*, provides a basic level of service availability. However, it does not allow any improvement of service reliability because it does not prevent interruption of already established sessions if they involve connection-oriented flows. In order to avoid restarting already established connection-oriented flows from scratch, further steps are required at the upper layers. These steps consist of resuming the already replicated service from a consistent state, at either the transport, session, or application level. In the following we provide an overview of already known techniques used to achieve consistent client transparent failover at the network, transport, session, and application levels.

Network-Level Failover

Network-level failover consists of providing replicas the means to take over the network identity of the legitimate processing server if it fails. It provides an acceptable level of service availability for stateless services since it allows the elected replica to transparently process the traffic originally offered to the failed node.

Different approaches have been proposed to provide the network level redundancy. The IETF 802.3ad link aggregation group proposed the link aggregation protocol [22] which allows the use of multiple Ethernet network interfaces or links in parallel to provide both increased throughput and availability capabilities. The same concept is called *EtherChannel* by Cisco, *Link Trunking* and *IP multipathing* by Solaris, and Channel Bonding or Network Interface Card (NIC) Teaming in Linux. The basic idea is to aggregate redundant network interfaces at the link level to give the network layer the illusion of dealing with a single network interface. Once the pri-

	Active replication	Message logging	Checkpointing
Resource usage	–Requires a dedicated backup	–Requires an idle backup	–Frequent checkpoint is costly
State preservation frequency	–States are created on the fly	–Connection-level messages are logged –Application-level messages are logged	–With every state change, etc.
Recovery time	–Short	–Long (message log replay)	–Less than the time required in the logging scheme
Failure-free overhead	–Active replication scheme dependent	–Additional delay	–The commit delay overhead
Nondeterminism handling	–Must be handled by the active replication method	–Issue for the connection and application level	–Undefined
Need for message interception	–Depends on the primary/backup topology	–Depends on the primary/backup topology	–Depends on the primary/backup topology

Table 2. *A comparison of state replication mechanisms.*

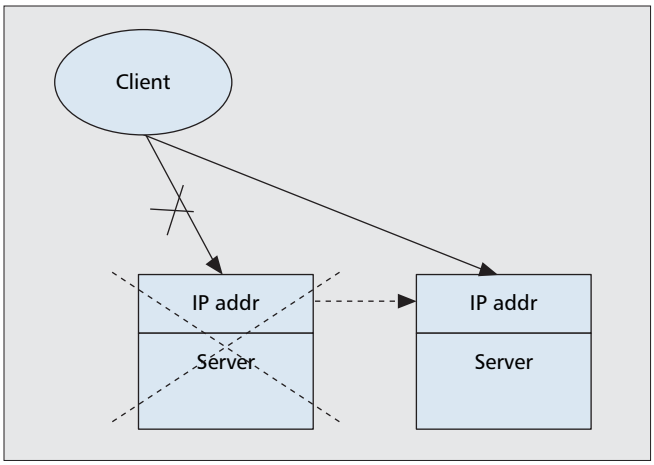

Figure 17. *ARP-spoofing-based IP takeover.*

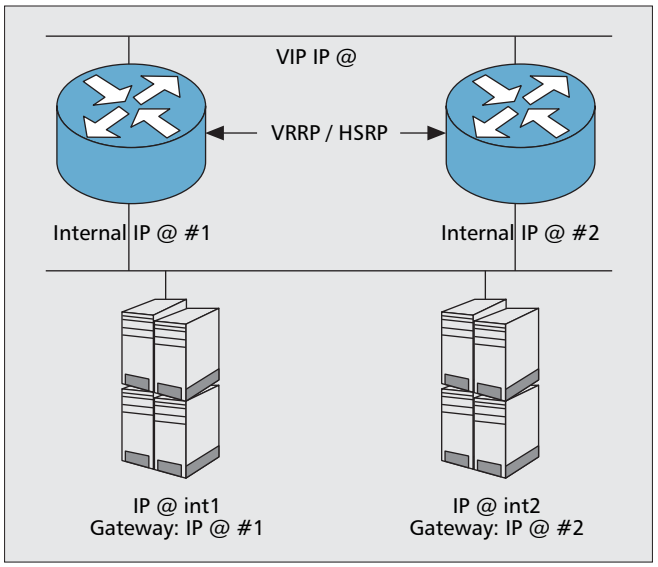

Figure 18. *VRRP/HSRP-based IP takeover.*

mary interface fails, an available NIC assumes its network-level address. Unlike channel bonding, IP multipathing provides the ability to failback to the primary network interface when the latter is restored to operational status. Aggregation also provides the capability of load balancing the inbound traffic across multiple network interfaces so as to achieve a higher throughput rate.

ARP-spoofing-based network identity takeover [23] provides the means to take over the network identity of a failed server using link layer ARP spoofing. Should the primary node fail, a backup node takes over the virtual IP address of the service. It starts then flooding the network with gratuitous ARP messages announcing the new association between the virtual service address and the link layer address of the lately elected primary (Fig. 17).

Fake [24] is an ARP spoofing based IP Takeover implementation for Linux. It is usually used conjunctly with Mon [25] and IpFail [26], which both provide failure detection capabilities.

The same idea has been used in the Virtual Router Redundancy Protocol (VRRP) [27], which standardized Cisco's Hot Standby Router Protocol [28]. VRRP is a nonproprietary redundancy protocol that provides routers with high availability capabilities. It builds the concept of a virtual router or virtual gateway, which abstracts a cluster of routers servicing hosts connected to the same network, instead of using a single physical router or gateway (Fig. 18).

VRRP covers both the active/standby and active/active scenarios. The active/active scenario uses load sharing techniques to improve replica utilization [29]. However, in the active/standby scheme, only the master node acts on behalf of the group at a given time by performing routing and replying to the ARP or ICMP packets. The master node periodically broadcasts gratuitous ARP messages to publish the association between the medium access control (MAC) address and the virtual IP address of the router or gateway. This MAC address has been fixed by the Internet Engineering Task Force (IETF) to 00-00-5E-00-01-XX, where the last byte corresponds to the configured virtual router identifier (VRID). Moreover, the master node periodically multicasts availability offers to all the replicas. An offer includes the priority of the node, virtual router identity, advertisement interval value, and so on at a given time; the master router has the highest priority value. Failure to receive a multicast packet from the master router for a period longer than three times the advertisement timer causes the replicas to assume master node failure. The virtual router then transitions to an unsteady state where an election process is initiated to select the new master router among the available replicas. On the other hand, planned failures force the priority of the master node to zero to speed up the takeover procedure.

VRRP can be used over Ethernet networks as well as multiprotocol label switching (MPLS) and token ring networks. Most router vendors provide VRRP-enabled routers. VRRP implementations are also available for Linux and BSD. Keepalived [30] is a VRRPv2-based implementation for Linux that provides high availability capabilities to active/standby LVS-based clusters. The Common Address Redundancy Protocol (CARP) [31] is a nonproprietary and unrestricted alternative to HSRP and VRRP.

A third network identity takeover mechanism assumes the grouping of replicas in a cluster of nodes (Fig. 19). The incoming traffic is first offered to the entry point to the cluster where network address translation is performed on the incoming traffic before assigning it to a master server. When the legitimate processing server fails, the dispatcher statically assigns the incoming traffic to an available replica by translating the service virtual IP address to the replica's private IP address. The major drawback of this approach is that it introduces the entry point to the cluster (the dispatcher) as a potential single point of failure.

Transport-Level Failover

Transport layer protocols rely on an explicit association between a service and its physical location for the wired Internet. Hence, when a host fails, the corresponding end-to-end flow terminates. Transport-level failover approaches aim to ensure the survivability of the active flows even in case of failure of the legitimate processing node. The reliability of the already established flows is indeed increased by backing every related flow state on a number of replicas. Should the primary server fail, the flow is taken over by an elected backup while avoiding its interruption.

The currently proposed solutions for high availability at the transport level are divided into two families. First are the *client-aware* solutions, requiring changes to the client operating system (OS) as well as to the application server. Second are the *client-transparent* solutions, moving most of the fault tolerance work to either the target server or an intermediate proxy. In the following we focus on the active replication-based client-transparent approaches.

Connectionless protocols have no native support for reliability. For the sake of reduced latency, time-constrained services as well as most of the legitimate services running

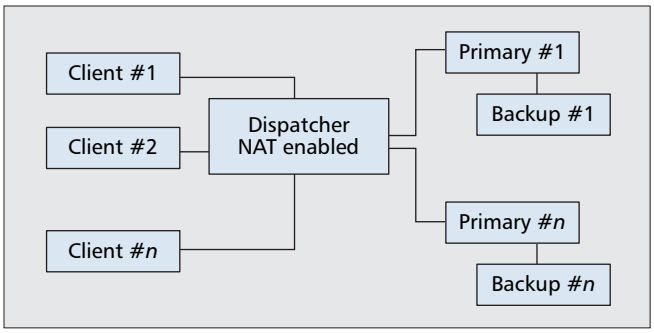

Figure 19. *Static NAT-based IP takeover.*

in the network such as DNS or NTP are UDP-based. Most of these services provide critical functionalities to the users. Hence, it is desirable to provide high availability features even for UDP-based communications. Connection-oriented services, however, involve more controls. In particular, TCP maintains a server state for each peer involved in a communication. TCP states store all the information required to perform error, sequence, and flow control. Different approaches have been proposed to provide TCP with high availability means. FT-TCP [32], transparent connection failover [33], and ST-TCP [34,35] are representative frameworks providing transport level failure recovery capabilities. They perform differently during failsafe and failure periods.

FT-TCP [32] is based on wrapping the TCP/IP stack into north and south modules operating between the TCP/IP layers and the network interface driver on one hand, and between the application server and the TCP layer on the other hand. Two modules are used on each replica to keep both the connection and the application states synchronized. The primary node first intercepts the incoming traffic as well as the system call results and writes them on an independent storage system. When *write()* operations are successfully acknowledged, the primary node sends back replies to the client. In parallel, the backup node reads the data on the stable buffer and uses it to synchronize its states. The major limitation of FT-TCP is that it significantly modifies both the primary and backup nodes. Moreover, changes on the primary node imply additional resources usage and delay to the end-to-end communication during failure-free periods.

In the transparent connection failover scheme [33], the backup node directly intercepts the incoming traffic. It produces data that is first sent to the primary node. The latter merges it with the data it already produced and then sends the resulting data back to the client. As we can see, the time saved by allowing the backup node to directly handle the legitimate traffic is wasted during the data merging process at the primary. Moreover, this approach incurs increasing usage of the primary's resources.

In ST-TCP [33, 34] the backup node passively intercepts the traffic flowing between the primary server and the clients. The TCP layer receive queue at the primary is modified such that it keeps a copy of any TCP segment already read by the application until a corresponding acknowledgment is sent back by the backup node. This aacknowledgment informs the primary that the data has been successfully processed at the backup. The main disadvantage of ST-TCP is its cost during failsafe periods. Indeed, when the backup server is not as fast as the legitimate processing server, the latter advertises reduced congestion windows to the clients, resulting in less end-to-end throughput. Moreover, in case of failure of the primary node jointly with datagram loss at the backup, ST-TCP leads to inconsistent replicated states at the backup server.

Session and Application Level Failover

Some regular services as well as most next-generation IP services involve multiple and heterogeneous flows for the same session. These flows are required for signaling and data exchange all along the session lifespan. Hence, failing over an already established session on an available replica requires to failback each associated state on the elected replica, be it a kernel- or an application-level state.

However, due to possible nondeterministic behavior at the application level, keeping replicas in a consistent state remains a nontrivial problem. Particularly, the active replication based approach doesn't apply natively to a wide range of applications. Other approaches have been proposed to handle the application-level inconsistency at the replicas during failure-free periods. A first approach consists of synchronizing the system calls generated at each replica by making the backup(s) intercept and use the primary's systems calls. While this approach has the advantage of being application internals unaware, it suffers from large overhead, particularly when the application executes a large number of system calls. An alternative solution consists of identifying the possible sources of any nondeterministic behavior at the application level. Replicas are then synchronized at those points (e.g., via messages). Content inspection can be used to identify the sources of nondeterminism at the application level. A third alternative consists of using checkpointing-based techniques such as those promoted by the Service Availability Forum (SA Forum) [36]. OpenAIS [37] and OpenClovis [38] are checkpointing-based frameworks requiring changes to the server side. Changes basically consist of placing checkpointing breakpoints into the server code, triggering the saving of the primary's application level states at the peer replica(s).

Conclusion and Lessons Learned

One of the factors that led to this work is the observation that current fault-tolerant frameworks are not adapted to next-generation Internet services. Indeed, these frameworks have to meet different challenges related to fault model robustness as well as fault model performance and resource consumption, during both failure-free and failure periods. In particular, currently proposed fault tolerance frameworks are neither transport- nor session/application-level-aware, even though the concerned services range from regular services such as file transfer to more recent services such as multimodal conferencing and voice over IP.

In this survey, we provide a comprehensive overview of the building blocks of fault tolerance frameworks. First, we focus on describing the different existing Internet server fault models. Then we outline the state-of-the-art failure detection approaches, and discuss their applicability to synchronous and asynchronous systems. Second, we deal with service replication concepts and approaches. We recall the different states required to be replicated on redundant replicas. These states belong to the network, flow, and service levels. In a second step we describe the state replication approaches. We outline their major limitations, and compare their performance during failure-free and failure periods. These approaches are the leader/follower, active replication, checkpointing-based, message-logging-based, and hybrid approaches. In particular, we show that the active replication approach offers the best recovery time value while being resource consuming during failure free periods. We also show that both the active replication and message logging approaches assume that servers are deterministic enough to result in the same output when provided with the same input. Finally, we focus on failure recov-

ery approaches for the network, flow, and session levels. Failure recovery consists of resuming the failed service starting from the last available consistent state rather than restarting the service from scratch. Network-level failure recovery provides a replica the means to take over the network-level identity of the failed server. Different mechanisms have been proposed such as the aggregation of network interfaces, the use of ARP spoofing, the use of the Virtual Router Redundancy Protocol as well as the use of network address translation. However, while providing an acceptable level of service availability for the stateless services, network-level failure recovery mechanisms provide no means to recover already established flows or sessions with the failed node. Transport- and session-level failure recovery approaches are required to avoid restarting already established flows or sessions from scratch. We focus on client transparent failover approaches, outlining in particular their performance during both failure-free and failure periods.

Other alternatives for high availability have been proposed. Distributed approaches provide service and data high availability capabilities. For instance, the DNS system allows the replication of sites to failback client requests on an available server in the event of failure of the legitimate processing server. The major disadvantage of the DNS system is the impact of its caching mechanism on the failure recovery period. Moreover, the DNS system provides no means to recover already established flows or sessions. Peer-to-peer networks provide a means to achieve improved availability of data and services by replicating the data over the Internet. However, some peer-to-peer topologies still involve particularly critical servers such as the bootstrap nodes. These nodes are involved during the bootstrap of given peer-to-peer services. According to some models, bootstrap servers may still communicate with peer nodes following a client/server communication. The state replication methods discussed here are valuable means to enhance the reliability of such critical components.

REFERENCES

[1] T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice*, Prentice Hall, 1981.
[2] T. Anderson and P. A. Lee, "Fault Tolerance Terminology Proposals," *Proc. 12th Int'l. Symp. Fault Tolerant Computing*, 1982.
[3] J. C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Dig. 15th Fault-Tolerant Computing Symp.*, 1985.
[4] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. 3rd Symp. Op. Sys. Design and Implementation*, Feb. 1999.
[5] W. Torres-Pomales, "Software Fault Tolerance: A Tutorial," NASA Langley Research Center, Hampton, Virginia, Oct. 2000.
[6] F. B. Schneider, "Byzantine Generals in Action: Implementing Failstop processors," *Proc. ACM Trans. Comp. Sys.*, May 1984.
[7] R. Aggarwal *et al.*, "Internet Draft: BFD For MPLS LSPs," Network Working Group, Mar. 2007.
[8] J. Bottomley, "Implementing Clusters for High Availability," *Proc. USENIX Conf.*, July 2004.
[9] L. Marowsky, "A New Cluster Resource Manager for Heartbeat," Jan. 2004.
[10] K. Kopper, *Linux Enterprise Cluster: Build a Highly Available Cluster with Commodity Hardware and Free Software*, Prentice Hall, May 2005.
[11] www.linux-ha.org/STONITH
[12] J. Bottomley, "Shared Storage Clusters," *SteelEye Tech. Tutorial*, 2000.
[13] C. Fetzer, "Perfect Failure Detection in Timed Asynchronous Systems," *IEEE Trans. Comp.*, 2003.
[14] D. C. Schmidt and C. O'Ryan, "Leaders/Followers, A Design Pattern for Efficient Multi-Threaded I/O Demultiplexing and Dispatching," Siemens res. rep., 2000.
[15] L. Wang, W. Zhou, and W. Jia, "The Design and Implementation of an Active Replication Scheme for Distributing Services in a Cluster of Workstations," *J. Sys. and Software*, 2001.
[16] O. Laadan, D. Phung, and J. Nieh, "Transparent Checkpoint-Restart of Distributed Applications on Commodity Clusters," *Proc. IEEE Int'l. Conf. CLUSTER 2005*.
[17] E. N. Elnozahy and W. Zwaenepoel, "On the Use and Implementation of Message Logging," *Dig.. 24th Fault-Tolerant Comp. Symp.*, 1994.
[18] N. Aghadie, "Transparent Fault Tolerant Network Services Using Off-the-Shelf Components," Ph.D. res. rep., 2005.
[19] S. Deering, "RFC 1112: Host Extensions for IP Multicasting," Network Working Group, Aug. 1989.
[20] Y. Huang and W. Yi-Min, "Why Optimistic Message Logging Has Not Been Used in Telecommunications Systems," *Proc. 25th Int'l. Symp. Fault-Tolerant Computing*, 1995.
[21] R. E. Strom and S. A. Yemini, "Optimistic Recovery in Distributed Systems," *Proc. ACM Trans. Computing Sys.*, 1985.
[22] SysKonnect white paper, "The Link Aggregation Protocol According to the IEEE 802.3ad," 2002.
[23] C. Fetzer and N. Suri, "Practical Aspects of IP Take-Over Mechanisms," *Proc. 9th IEEE Int'l. Wksp. Object-Oriented Real-Time Dependable Sys. 2003*.
[24] www.vergenet.linux.net/linux/fake/
[25] www.linuxvirtualserver.org/software/Mon/
[26] www.linuxvirtualserver.org/software/IpFail
[27] S. Knight *et al.*, " RFC 2338: Virtual Router Redundancy Protocol ," Network Working Group, Apr. 1998.
[28] T. Li *et al.*, "RFC 2281: Cisco Hot Standby Router Protocol (HSRP)," 1998.
[29] R. Hinden, "RFC 3768: the Virtual Router Redundancy Protocol," Network Working Group, Apr. 2004.
[30] http://www.keepalived.org
[31] OpenBSD Programmer's Manual pages, Section 4, "CARP: Common Address Redundancy Protocol," 0ct. 2003.
[32] D. Zagorodnov *et al.*, "Engineering Fault-Tolerant TCP/IP Servers using FT-TCP," *Proc. Int'l. Conf. Dependable Sys. and Networks*, 2003.
[33] R. Koch *et al.*, "Transparent TCP Connection Failover," *Proc. Int'l. Conf. Dependable Systems and Networks*, DSN 2003.
[34] M. Marwah, S. Mishra, and C. Fetzer, "A System Demonstration of ST-TCP," *Proc. Int'l. Conf. Dependable Sys. and Networks*, 2005.
[35] M. Marwah, S. Mishra, and C. Fetzer, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server," *Proc. Int'l. Conf. Dependable Sys. and Networks*, 2003.
[36] http://www.saforum.org
[37] http://www.openais.org
[38] http://www.openclovis.com

BIOGRAPHIES

NARJESS AYARI (narjess.ayari@orange-ftgroup.com) is a Ph.D. student and research engineer at France Telecom R&D Laboratories, Lannion, and a member of the RESO team at the LIP laboratory in the Ecole Normale Supérieure de Lyon (ENS-Lyon) in France. She received her engineering diploma in computer science from the National School of Computer Science of Tunis (ENSI) in 2001, and an M.Sc. in networking and distributed systems from the same school in 2003. Her current research interests include network traffic load balancing, high availability, QoS. and admission control in wired networks. Her Web site is http://perso.ens-lyon.fr/~narjess.ayari

DENIS BARBARON (denis.barbaron@orange-ft.com) graduated in computer science with a diploma of specialized higher studies from the University of Science of Nantes, France, in 1994. He has worked with several telecom companies in France since 1988. He joined FT R&D in 1997 as an R&D engineer. Since 2005 he has been a senior expert in cluster architectures at France Telecom R&D Laboratories, Lannion. His actual research interests focus on carrier grade telecom service platforms.

LAURENT LEFEVRE [M] (laurent.lefevre@inria.fr) obtained his Ph.D. in computer science in January 1997 at Laboratoire Informatique du Parallélism (LIP), Ecole Normale Supérieure de Lyon, France. From

1997 to 200 he was an assistant professor in computer science at Lyon 1 University. Since 2001 he is a permanent researcher in computer science at INRIA, the French Institute for Research in Computer Science and Control. He is member of the RESO team (High Performance Networks, Protocols and Services) at LIP. He has organized several conferences in high performance networking and computing, and is member of several program committees. He has co-authored more than 70 papers published in refereed journals and conference proceedings. He takes part in several research projects. His research interests include autonomic networking, high performance active networks, active services, high performance network protocols, grid and cluster computing network support, active grid, distributed shared memory systems, and data consistency. His Web site is http://perso.ens-lyon.fr/laurent.lefevre

PASCALE PRIMET (pascale.primet@ens-lyon.fr) is a senior scientist (directrice de recherche) at INRIA. Based at Ecole Normale Supérieure de Lyon (ENS-Lyon), within LIP, she currently leads the RESO project team at INRIA. This team is specialized in communication protocols and software optimization for high-speed networks and grid environments. From 1989 to 2001 she worked as a researcher and lecturer at Ecole Centrale de Lyon (ECL). In 2001 she joined INRIA as a researcher. In 2001–2002 she managed the Networking Work package of the European IST Data Grid project. In 2002–2003 she was the scientific coordinator of e-Toile, the first national RNTL grid platform project funded by the French Research Ministry. She is a member of the Steering Committee of the GRID5000 project, responsible for the networking area. She has been a co-chair of the GGF Data Transport Research Group. She is general chair of the GridNets conference, a member of the steering committee of the PFLDnet workshop series, was HPDC '06 workshop chair, and was session QoS and security chair of the 1st ITU-GGF workshop. She is a member of several international conferences' program committees (Vecpar, EuroPAR, AGMN, pfldnetELLIPSIS). She is an expert in networking for the National Research and Science Center and the National Research Agency, and a member of the GLIF community. She has published more than 60 research papers on networks, protocols, and grids.