Duplex: A Reusable Fault Tolerance Extension Framework for Network Access Devices

Srikant Sharma   Jiawu Chen   Wei Li   Kartik Gopalan   Tzi-cker Chiueh
{srikant, jiawu, weili, kartik, chiueh}@cs.sunysb.edu

Rether Networks Inc.
99 Mark Tree Road, Suite 301
Centereach, NY 11720

Computer Science Department
Stony Brook University
Stony Brook, NY 11794

Abstract

A growing variety of edge network access devices appear on the marketplace that perform various functions which are meant to complement generic routers' capabilities, such as firewalling, intrusion detection, virus scanning, network address translation, traffic shaping and route optimization. Because these edge network access devices are deployed on the critical path between a user site and its Internet service provider, high availability is crucial to their design. This paper describes the design, construction and evaluation of a general implementation framework for supporting fault tolerance on edge network devices. This implementation framework, called Duplex, is designed to be independent of the functionality of the hosting edge network access device, such that only a minimal amount of programming is required to tailor this framework to a specific edge network access device implementation. Duplex can tolerate power failure, hardware failure, and software failure by supporting device mirroring and watchdog timer-based link bypassing. Empirical performance measurements of an instance of Duplex that is embedded in a commercial bandwidth management device show that the run-time overhead of its fault tolerance mechanisms is less than 1 msec 90% of the time, and the failure detection and recovery period is less than 1.3 sec when running at 100 Mbps.

1. Introduction

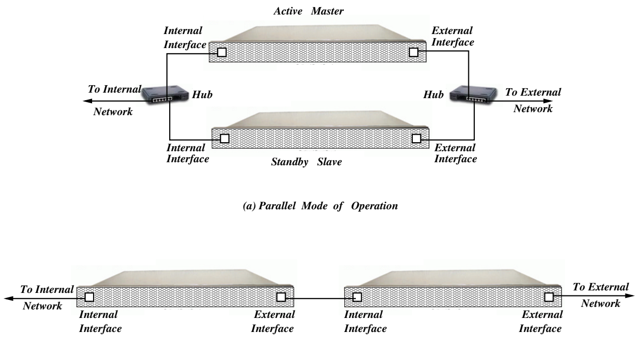
The network traffic coming into and getting out of a user site needs to be managed. Unfortunately, the packet management functions on standard routers are relatively limited. As a result, a growing variety of edge network access device (NAD) products have been developed to perform various packet filtering, scheduling, and transformation functions, such as firewalling, intrusion detection, virus scanning, network address translation, traffic shaping, route optimization, etc. These NADs are required to be deployed on the critical path between a user site and the rest of the Internet, because they need to see all the packets in and out of the user site. Accordingly, these NADs have to be highly available for undisrupted connectivity of the user sites to the Internet.

Although different NADs may perform different network packet management functions, their underlying hardware and software architectures are quite similar. NADs typically have two network interfaces. When packets arrive at one interface, they are buffered, inspected, modified, and queued for transmission on the other interface or simply dropped. From the standpoint of fault tolerance support, these NADs share a common set of requirements:

- The hard state that needs to be recovered after a failure is small and updated relatively infrequently; for example, the filter rules in a firewall, the set of public addresses available for a network address translator, the bandwidth reservations in a traffic shaping device, etc.
- It is acceptable to lose some packets during the failure recovery period. Therefore, the fail-over does not have to be instant.
- Connectivity to the Internet should be preserved at all costs. That is, if there is no other alternative but to let a NAD die, the NAD should behave like a passive cable after its death.

Given these common requirements for NADs, it is natural to consider a reusable fault tolerance implementation framework that could be tailored to individual NADs with a minimal amount of coding efforts in order to greatly reduce their development cost and the time to market. The focus of this paper is exactly about the design, implementation and evaluation of such a framework called *Duplex*.

The proposed fault tolerance implementation framework supports watchdog timer based link bypassing after unrecoverable failures, and device mirroring, which allows a slave device to take over a failed master device. The master and



(a) Parallel Mode of Operation

(b) Serial Mode of Operation

Figure 1. Duplex supports two dual-device configurations: (a) Parallel-Dual (b) Serial-Dual configuration.

slave devices could operate in a parallel or serial configuration as shown in Figure 1. In parallel mode, the internal interfaces of master and slave are connected to a common hub, as are their external interfaces. Consequently, both devices see the same set of incoming and outgoing packets, barring packets dropped at the network interfaces. In serial mode, the internal interface of one device is connected to the external interface of the other. This mode makes it possible to turn these two devices into a passive cable when they fail simultaneously. However, this flexibility comes at the expense of extra forwarding latency at each of the devices.

The services supported by Duplex include state logging, state synchronization between master and slave, failure detection, and state recovery after failure detection. In addition, Duplex includes a set of service programming interfaces for NAD developers to provide device-specific information, particularly those related to device state that needs to be recovered after a failure.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3, we present the individual building blocks of Duplex, namely, state logging and synchronization, failure detection, and failure recovery, and discuss the parallel and serial mode of operation. In Section 4, we describe how a commercial network access device called ISMD makes use of the fault tolerance mechanisms in Duplex. Section 5 presents the performance measurements of the failure detection and recovery mechanisms embedded in ISMD. Section 6 provides a summary of this work and an outline for future directions.

2. Related Work

To the best of our knowledge, Duplex is the first fault tolerance implementation framework designed specifically for network access devices. It integrates the well known fault tolerance concepts of process pair paradigm, state logging and replication, recursive restartability, and rollback recovery into a generic extensible framework tailored for high performance network access devices.

Duplex is based on the *process pair* paradigm for fault tolerance that was first pioneered in Tandem's NonStop kernel [2]. A pair of processes - primary and backup - are used to implement a service that needs fault tolerance. The primary process is responsible for all requests to the service and periodically *checkpoints* the operations on the device's internal state to the backup process. The backup process acts as a hot standby and assumes the responsibility of servicing incoming requests in the event of the failure of the primary process. The idea of using hot-standby backup servers has also been applied in web server systems [6, 1] where standby servers take over the web server request processing from a failed server. Process System Support (PSS) [8] provides a meta-programming environment that applies the process-pair approach to ensure high availability for distributed applications. Isis [3] is a distributed middleware that offers primitives for managing process groups, broadcast, failure detection and recovery, and synchronization using which primary/back and server replication mechanisms can be constructed.

The hot-standby backup device approach has also been applied to making routers resilient to fail-overs. The Virtual Router Redundancy Protocol (VRRP) [10] specifies an election protocol by which responsibility for routing is assigned to another standby router in the LAN in case of failure of the master router. The motivation behind VRRP is similar to Duplex in that it aims to eliminate a single point of failure in network access that can result in loss of connectivity. Similar proprietary protocols have been proposed by Cisco [12] and DEC [9].

Rollback recovery has been studied extensively in literature. Elnozahy [7] provides an excellent survey of rollback recovery techniques. There are two kinds of rollback recovery techniques: *checkpoint based* and *logging based*. Checkpoint based techniques periodically save the state of an executing process to a disk file from which it can be recovered after a failure. Examples of work on checkpoint based techniques include Libckpt [13] and Libckp [14]. Checkpointing of process state is an expensive operation in the context of high performance network access devices. Duplex provides a logging based mechanism that keeps a persistent record of nondeterministic events, such as changes made to the device configuration. In the event of a failure, the logged events are replayed in their original order to recreate device's pre-failure state. Duplex's logging mechanism achieves the best of both *pessimistic* logging [2, 4] or *optimistic* logging [11]. Pessimistic logging requires the service to block till the log is written to stable storage. Optimistic logging allows the log message to be flushed asynchronously to stable storage while the process goes ahead with other operations.

The concept of recursive restartability has recently gained popularity [5], in which a fault-tolerant system allows restarting components at multiple levels depending upon severity of failure. In a similar spirit, Duplex API permits

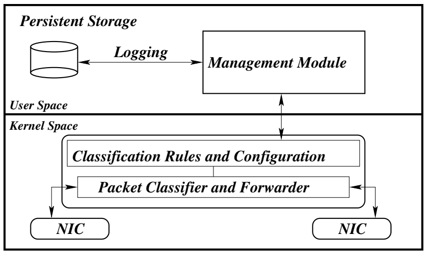


Figure 2. A network access device typically consists of a kernel-space packet processing engine, a user-space management daemon, and an administrative interface running on a remote desktop. The kernel-space packet processing engine acts on incoming packets according to pre-defined rules maintained by user-space management daemons. The processing rules constitute the hard state that the NAD needs to log in the normal mode and recover upon a failure.

specification of various failure events and the corresponding NAD components that need to be restarted in case of a failure.

3. The Duplex Framework

3.1. Overview

The software structure of a typical network access device is shown in Figure 2. In this paper, we focus specifically on NADs that are built from customized software running on standard hardware such as an industrial PC. The software of such network access devices is typically comprised of two major components: a kernel-space packet processing engine, and a user-space management daemon for configuration and statistics reporting. For instance, a firewall device maintains packet filtering rules that specify which packets to drop and which to forward. Similarly, a network bandwidth management device maintains reservation rules to prioritize the transmission order of incoming and outgoing packets. These “control information” are dynamically changing because they are subject to modification either manually or through a programming interface. The control information could be communicated between a NAD and the external world through a standard protocol such as SNMP. Because a NAD’s control information determines its run-time behavior, updates to the control information are logged to persistent storage for recovery after failures.

The design goal of the Duplex framework is to allow a NAD developer to convert a given NAD implementation into a fault-tolerant version with a minimal amount of coding effort. Toward this end, Duplex supports a set of general mechanisms for both single-device and fully redundant dual-device configurations and a service programming interface (SPI) for device-specific customization. These mechanisms include – (1) Liveness check of the hardware platform, the network interfaces, the kernel-space packet processing engine, and the user-space management daemon, (2) Control information logging and synchronization between the master and slave devices of a mirrored device pair, and (3) Transparent and seamless switch-over after a failure, and fast recovery of the failed device.

An important consideration in the design of a fully redundant dual-device NAD is to hide the complexity of fail-over from the rest of the network. This means that all the network interfaces of the two devices should be bound to one unique IP address (called *global IP*). However, because the two devices need to communicate with each other, additional private IP addresses should be assigned to these interfaces as well. Although both devices are bound to the global IP, only one of them should interact with external world through messages like ARP queries, TCP or UDP traffic, etc. The other device should silently discard the packets addressed to the global IP address. In the event of a fail-over, the device that takes over should start processing packets addressed to the global IP address. However, private communication between the two devices remain unaffected.

3.2. Service Programming Interface

Duplex provides a set of programming interfaces that a NAD developer can use to customize Duplex to a specific NAD implementation along the following three dimensions: a) user-level process liveness check, b) kernel module liveness check, and c) state logging and recovery.

When a Duplex network access device is started, a special start-up script bootstraps and registers its associated user-level management processes according to a configuration file. This configuration file specifies the set of management processes that need to be started, how to start them, and what to do when these processes fail at run time. After the start-up script spawns these management processes, it informs Duplex’s liveness check module of their process ID for run-time monitoring. When a user-level management process dies, possible actions include restarting the failed process, restarting all user-level management processes, or rebooting the entire NAD. The configuration file also includes information about kernel modules whose liveness needs to be checked at run time. Duplex provides a kernel-level interface called `iam_alive()` that needs to be periodically invoked by every kernel module of a NAD. Invocation of `iam_alive()` indicates that the calling kernel module is alive and is functioning correctly. Failure to invoke `iam_alive()` for a period of time signals a failure of a kernel module and triggers the failure recovery procedure that is associated with the failed module as specified in the configuration file.

Control state on a NAD needs to be stored in a persistent log for failure recovery, and in the case of fully redundant device mirroring, should be synchronized between the master and slave devices. For this purpose, Duplex provides certain logging interfaces, such as, `log_async()`, log_notify(), and log_recover(). The logging functionality is explained in detail in Section 3.6.

3.3. Failure Detection

The Duplex framework includes failure detection mechanisms for both non-redundant and fully-redundant configurations. In the *single* (non-redundant) configuration, detection of hardware and software failures is based on a periodic liveness check mechanism embedded into the timer interrupt service routine. Whenever a hardware timer interrupt occurs, the liveness check code is invoked to verify if the user-level management processes and kernel-level modules are still alive, and to take appropriate actions when some of these checks fail.

In the case that the timer interrupt is incorrectly disabled due to a software failure, the failure detection module will not have a chance to be triggered. Duplex relies on watchdog timer hardware to address this problem. Duplex assumes the existence of a dual-channel network interface card that has a bypass circuit controlled by a watchdog timer. The on-card watchdog timer needs to be refreshed periodically to continue in normal operating mode. Duplex implements this refresh operation also in the timer interrupt service routine. Upon a hardware/power failure or a software failure that disables the timer interrupt, the timer refresh code cannot get triggered and the on-board watchdog timer times out, which in turn causes the on-board hardware relays to short-circuit the two channels. As a result, the inbound and outbound network links of a NAD get physically connected when the hardware or software on the NAD fails. In this scenario, the NAD is effectively bypassed and becomes a passive cable.

During initial bootup, the dual-channel NIC operates in Bypass mode by default. For proper operation, Duplex first switches it into the Normal mode and enables the watchdog timer. From this point onwards the watchdog timer is refreshed periodically. Any failure in software or hardware liveness check disrupts the periodic refresh activity, and results in the bypass circuit being triggered. Thus, for the single configuration, Duplex can recover from failures of a NAD's user-level management processes, kernel-level modules, software hangups, and hardware/power failures. In some cases, the recovery is partial in that the NAD is converted into a passive link.

In the *dual* (fully-redundant) configuration, the master and slave need to constantly monitor the liveness of each other. Duplex supports an *adaptive heartbeat mechanism* for this purpose. The master and slave send a heartbeat packet to each other periodically, and declare a failure when each fails to receive a heartbeat packet from the other over a period of time. Because heartbeat packets are sent over the same network interface used by user traffic, there is a possibility that heartbeat packets may be lost due to collision or congestion. To distinguish between heartbeat packet loss and device failure, Duplex uses a two-level time-out mechanism. When a *soft* timer on a device expires, Duplex begins to suspect that the other device is dead, and increases the heartbeat packet frequency to probe the other's liveness. When the *hard* timer on a device expires, Duplex concludes that the other device is dead, and initiates the switch-over procedure if it is the slave device, or converts itself into the single configuration if it is the master device. To eliminate the influence of link congestion, both soft and hard timers are dynamically adjusted according to the current user traffic load, based on the following equation:

$$hard\_timeout = soft\_timeout + \frac{current\_traffic\_load}{2^{load\_factor}} \quad (1)$$

In practice, the load_factor is empirically set to 14 for 100 Mbps networks and to 11 for 10 Mbps networks after observing system behavior. The soft_timeout is set to 100 msec. In summary, the adaptive heartbeat mechanism reliably distinguishes among device failure, network interface failure, link failure or heartbeat loss due to congestion.

3.4. Parallel-Dual Configuration

In *Parallel-Dual* configuration, two NAD devices with identical hardware and software are connected in parallel as shown in Figure 1. During normal operating mode, one of the devices serves as the *master device* while the other as the *slave device*. When the master device fails, the slave device detects the failure and takes over as the new master device. The communication between master and slave could be through a dedicated path such as a serial line, or through the standard network interfaces. Duplex assumes master-slave communication goes through the standard network interfaces to remove special communication line requirement and to make it possible to detect network interface failure.

Duplex uses a finite state machine to model the transition of the two devices of a fully redundant NAD during the failure detection and recovery period. There are six possible states that a device could be in: *Probe, Master, QSMaster, SubMaster, Slave*, and *Suspended*. The heartbeat packets exchanged between two devices include the current state of the sending device. Transitions among these states are triggered by heartbeat packets or their absence within a certain period of time, as shown in Figure 3. Using this state transition mechanism, Duplex is able to tolerate device hardware/software failures and network interface failures associated with a device.

Once started up, a NAD device enters Probe state for five seconds to decide into which state it should evolve. If the peer device is already in Master state, then it enters Slave state. If it fails to detect an active Master, the device assumes the role of Master. If both devices are in Probe state, the contention is resolved based on the MAC address of the

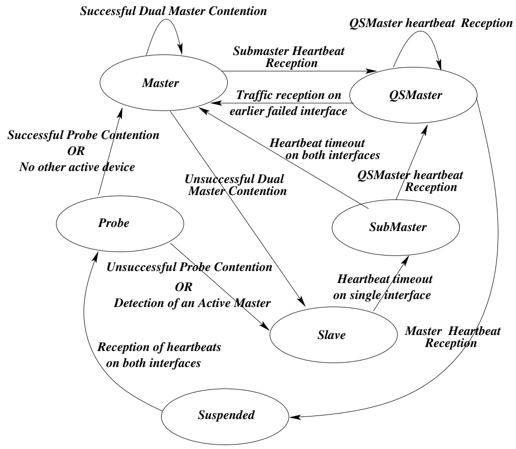

Successful Dual Master Contention
QSMaster heartbeat Reception
Submaster Heartbeat Reception
Master
QSMaster
Traffic reception on earlier failed interface
Successful Probe Contention OR No other active device
Heartbeat timeout on both interfaces
QSMaster heartbeat Reception
Unsuccessful Dual Master Contention
Probe
SubMaster
Unsuccessful Probe Contention OR Detection of an Active Master
Heartbeat timeout on single interface
Slave
Master Heartbeat Reception
Reception of heartbeats on both interfaces
Suspended

Figure 3. The state transition diagram for the Parallel-Dual mode. Each device starts in Probe state. State transitions on a device are triggered based on heartbeat messages from the pairing device. In normal operating mode one device is in Master state and the other in Slave state. When a failure is detected but not yet fully confirmed, the devices evolve through QSMaster or SubMaster states. After a failure is confirmed the failed device enters Suspended state.

devices' network interfaces. The provision of Probe state makes it possible to use the same probing mechanism for the single and dual configurations. In the normal operating condition, one device is in Master state and the other is in Slave state. The master device provides the services of the NAD to the outside world, and interacts with any remote administrative interface.

In the Parallel-Dual configuration, master and slave device are connected through both of their network interfaces, and the heartbeat packets are also exchanged between the two devices through both network interfaces. Because the time to detect Master device failure plays an important role in the end-to-end fail-over latency, the heartbeat frequency from Master to Slave is set to be significantly higher than the heartbeat frequency from Slave to Master. When the slave device fails to receive any heartbeat messages over a pre-defined interval through one of the network interfaces, there are three possibilities[1] : (1) the master device is dead, (2) the corresponding network interface of the master device is dead, and (3) the corresponding network interface of the slave device is dead. To distinguish among these cases, Duplex introduces an additional state called Quasi-Master (or QSMaster) state, in which the devices determine whether the network interface in question is working properly. When a device in QSMaster state decides that its interface is alive, it enters Master state; the other device, upon learning that its peer is entering Master state, enters Suspended state and eventually Slave state.

The QSMaster is a transient state which is used for identifying the failed device. When a device in QSMaster state receives a heartbeat packet which indicates that its peer is in Master state, it enters the Suspended state. If after a failure, the Slave node immediately transitions to the QSMaster state there is a high probability that it will receive heartbeats from current Master which has not yet switched to the QSMaster state. As a result, it may wrongly switch to Suspended state without verifying its own liveness. Thus, upon the detection of missing heartbeat packets, the master device should enter QSMaster state before the slave device. This ensures that when the slave device enters QSMaster state, the only reason that it could receive a heartbeat packet indicating its peer is in Master state is because its peer has gone through the transition of Master, QSMaster, and Master. Duplex introduces one more state called SubMaster to guarantee this ordering. Assume the master and slave devices originally work properly, and when the slave device fails to receive any heartbeat messages over a pre-defined interval through one of its network interfaces, it enters SubMaster state. After the master device receives a heartbeat packet through another network interface that indicates the slave device is in SubMaster state, the master device enters QSMaster state. After the slave device learns, via heartbeat messages, that the master device is in QSMaster state, it enters QSMaster state as well. Now that both devices are in QSMaster state, the one that detects its network interface is working earlier enters Master state, and the other enters Suspended state. If the master device dies, the slave device takes over as Master after detecting the loss of heartbeats on both the interfaces. Note that it is OK for the slave device to enter Master state even when the reason that it cannot receive heartbeat messages from the master device is because both of its interfaces are dead. In this case, since the slave device is cut off the network, it does not matter which state it is in.

In QSMaster state, devices check the liveness of their interfaces by sending out ARP queries targeted at hosts that are known to be reachable through these interfaces. If there is *any* traffic received on both interfaces the device elevates itself to Master state. It is also possible that loss of consecutive heartbeat packets is due to transient errors. In this case, both devices conclude that their interfaces are working properly and attempt to enter Master state. The device that has interacted with the remote administrative interface most recently will be the winner of this Dual Master contention.

Logically there are three types of high-level activities: (1) NAD function, (2) communication with remote administrative interface, and (3) network interface liveness check. Devices in Master state perform (1) and (2), devices in SubMaster state perform (1), and devices in QSMaster state perform (1) and (3). Although both devices may perform the NAD function simultaneously, this will not cause problems as only one device is truly connected to the network. When a device is in Suspended state, it tries to first determine whether its hardware and software are working properly, and moves on to Probe state only when that is the case.

[1] The possibility of packets lost due to congestion is eliminated through the dynamic two-level time-out mechanism described earlier.

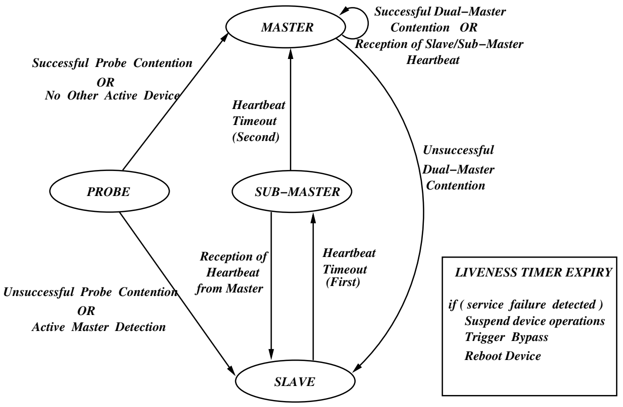

MASTER
Successful Dual–Master Contention OR Reception of Slave/Sub–Master Heartbeat
Successful Probe Contention OR No Other Active Device
Heartbeat Timeout (Second)
Unsuccessful Dual–Master Contention
PROBE
SUB–MASTER
Reception of Heartbeat from Master
Heartbeat Timeout (First)
Unsuccessful Probe Contention OR Active Master Detection
SLAVE
LIVENESS TIMER EXPIRY
if (service failure detected)
Suspend device operations
Trigger Bypass
Reboot Device

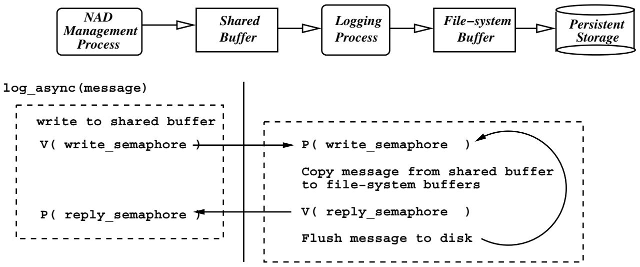
Figure 4. The state transition diagram for serial mode operation of Duplex framework.

3.5. Serial-Dual Configuration

Figure 4 shows another configuration supported by Duplex – the Serial-Dual configuration. When a NAD device comes with a dual-channel network interface card with bypass capability, the Parallel-Dual configuration is no longer appropriate because failure of a device may add a direct packet forwarding path that could impact normal network operation. In this case, it is better to connect the master and slave devices in Serial-Dual configuration. A key advantage of the Serial-Dual configuration over the Parallel-Dual configuration is that even when both devices die, the connectivity between a user site and the Internet remains available.

In the Serial-Dual configuration, the master device and the slave device are connected through one interface, over which the heartbeat messages are exchanged. Because any network interface failure of either device could break the Internet connectivity, it is essential to tolerate such failures. When a device fails to receive packets on an interface over a period of time, it induces traffic on that interface by sending ARP queries for known hosts and if still no traffic is observed, the device triggers the bypass circuit on the dual-channel network interface card. With this mechanism in place, Duplex can now assume that the network interfaces of both devices are alive. When the slave device fails to receive any heartbeat messages within a pre-defined interval, i.e., a heartbeat timeout, it enters SubMaster state and sends out a SubMaster heartbeat to inform the master device of the loss of heartbeat packets. If the master device dies, the slave device will move to Master state after another heartbeat timeout; otherwise, the master device will send back a Master heartbeat immediately to return the slave device back to Slave state. In this design, the old master device is given a priority to stay in Master state so that unnecessary switchover is avoided.

Compared with the Parallel-Dual configuration, the slave device in the Serial-Dual configuration is no longer a standby device but an active device that blindly forwards all traffic. In addition, it is necessary for each device to determine which interface should be used in communicating with its peer. Finally the state transition diagram is simpler, because there is only one interface for inter-device communication, and it is no longer possible for the two devices to communicate with each other after a heartbeat timeout.



```
log_async(message)
  write to shared buffer
  V( write_semaphore          P( write_semaphore  )
                              Copy message from shared buffer
                              to file-system buffers
  P( reply_semaphore          V( reply_semaphore  )
                              Flush message to disk
```

Figure 5. Asynchronous Logging Procedure.

3.6. Fast State Logging and Recovery

The state of a NAD consists of configuration information and operational rules, such as packet filter rules in firewalls and bandwidth reservations in traffic shapers, and should be put to a persistent log for post-failure recovery. Although the format of log records is NAD specific, it is possible to provide generic read and write interfaces for log access. In addition, for the dual configuration, it is essential to replicate the control state of the master device on the slave device, and keep them synchronized at all times. Therefore, logging a state update involves writing a log record to the master device's local log file and the slave device's log file.

Although it is semantically desirable to write to the log files synchronously, the associated latency is unacceptable for a NAD, especially at high link speeds. To address this problem, Duplex uses a dedicated logging process that shares a buffer with the NAD's user-level management process. Whenever the user-level management daemon needs to log a state update, it calls the `log_async()` interface, which writes the log message to the shared buffer and signals the logging process, which in turn completes the logging operation by performing a synchronous write to the log file. As the synchronous disk access request is being serviced, the logging process is blocked and the control of the CPU is transferred back to the NAD. In the dual configuration, the logging process on master further contacts the logging process on the slave device to replicate the log record. The user-level management daemon on the slave device can periodically check for new log updates or can register a callback routine using the `log_notify()` interface to handle new log updates.

Figure 5 illustrates the asynchronous logging procedure. Duplex uses two user-level semaphores to manage the control flow between the user-level management process and the logging process - a write semaphore and a reply semaphore.

The logging process just sleeps on the write semaphore when there is nothing to write to persistent storage. When there is some message to be logged, the user-level management process calls log_async(), which writes the message to the shared buffer and wakes up the logging process through a V operation on the write semaphore. Once the logging process takes control, it initiates a disk write operation, then allows the user-level management process to continue via a V operation on the reply semaphore, and finally performs a flush operation on the write to complete the real disk operation. This way, the visible logging latency for the NAD is just two extra memory copies with a small synchronization overhead, although each message is indeed synchronously written to the log file.

When a device boots up, it calls a log recovery function log_recover(). This function first checks whether there is a peering device. If no peer exists, the device restores its control state from the local log file. If a peer exists and the checksum of local log file matches with that of the peer then the device restores the control state from the local log file. If a peer exists but their logs' checksums do not match, the device restores its control state completely from the log file of its peer. This ensures the control states of the peering devices are synchronized.

4. Case Study: Internet Service Management Device

The Internet Service Management Device (ISMD) system from Rether Networks Inc. (www.rether.com) is a network traffic management system that controls the access link bandwidth usage based on user-defined policies. It is comprised of three major components: the core network access device ISMD, an Archive Server (AS) and a Network Management Interface (NMI). The AS and NMI provide a web-based management and analysis interface for such functions as flow reservations, and real-time statistics reporting. The availability of an ISMD system and hence the access link it controls is relatively unaffected by AS and NMI status as these components are primarily meant for user interaction and statistics collection. Moreover, to maximize the user access flexibility, there are no state information stored either on AS or on NMI.

The core ISMD itself has a user-level manager that is responsible for communicating with the AS and for starting the ISMD engine inside the kernel. The ISMD kernel module performs such functions as packet scheduling, forwarding, load balancing, and network address translation (NAT), and is derived from Linux. Like other network access devices, ISMD is also prone to hardware failure, single or multiple link failures, network interface failures, and service failures (in kernel engine or user instance). To improve the availability of the core ISMD, we apply the Duplex framework to add logging and recovery to user-level manager, state transition mechanism to kernel module, and liveness checks in the interrupt service routines.

4.1. User-Level Manager Customization

Since the user-space manager is an essential part for the health of the core ISMD, its liveness needs to be constantly monitored by the liveness check module. Toward this end, the user-level manager is started by Duplex's start-up script, so that Duplex's liveness check module is made aware of the process ID of the ISMD user-level manager and is able to monitor its health from that point on. In addition, the fallback operations when the user-level manager dies are also specified, so that the liveness check module could take corresponding actions, in this case, it will reboot the machine and restart from the latest consistent control state.

ISMD's user-level manager maintains various control state, including reservation list (for bandwidth management), router list (for load balancing), etc. These control states need to be stored in persistent storage so that the ISMD can recover them after a failure. For the single configuration, the user-level manager is modified to invoke log_async() whenever there is a state change, and after a failure, the user-level manager invokes log_recover() immediately after it restarts. For the dual configuration, the user-level manager needs to additionally register a call-back function through the log_recover() interface. This call-back function allows the user-level manager to update its internal state in response to new control state changes when the ISMD is operating as a slave device.

4.2. Kernel Module Customization

The kernel engine of the core ISMD keeps scheduling and forwarding packets in an infinite loop until one of the following events arises: reception of a local packet, or expiration of the software timer for giving control to the user-level manager. In each loop iteration, it polls network interfaces to receive packets, performs packet classification, queues packets, and schedules packets transmission. To enable liveness check on the ISMD kernel engine, a call to iam_live(&ismd) is included in the main loop. In some cases, the liveness check on the ISMD kernel engine fails because the CPU control stays at the user level for too long and thus the kernel engine is unable to invoke the iam_live (&ismd) call. To address this problem, a separate iam_live (&syscall) call is included at the entry point of all system calls to check the liveness of the entire kernel. The core ISMD is considered dead with when Duplex detects (1) failure of invocation of iam_live (&ismd), (2) failure of invocation of iam_live (&syscall), and (3) failure of the user-level

Traffic	**Fail-over time (ms)**		**Lost Packets**	
(Mbps)	Parallel	Serial	Parallel	Serial
0	101	106	0	0
2	129	126	58	1594
5	159	160	189	3721
10	231	220	514	7246
20	346	247	1632	13899
30	457	460	3370	17424
40	588	591	6270	28511
50	717	713	8705	29477
60	833	836	12239	36385
70	952	955	26365	42221
80	1073	1089	32494	48579
90	1174	1197	51225	58902
100	1287	1267	68705	73728

Table 1. Fail-over time and lost packet count for Parallel-Dual and Serial-Dual ISMD. The timings are against the increasing traffic load. The packet size is fixed at 512 bytes. The packet loss is more in Serial-Dual compared to the Parallel-Dual configuration because in this configuration the Slave cannot act as a warm replica of the Master during normal operation.

manager. The fall-back operation for all these failure scenarios is to reboot the ISMD.

Duplex is responsible for detecting failures and managing the state transitioning during the failure detection period. A NAD should perform different operations when it is in a different state. For example, when an ISMD device in SubMaster mode, it should perform the core ISMD function, but should not communicate with the AS. What a NAD should do in a particular state is completely up to the developers of the NAD. Duplex does not offer any help except providing a global state variable called `devicestate`, which indicates the current state of the NAD. Changing the hosting NAD's implementation to properly interact with the state transition mechanism in Duplex is expected to take most of the customization efforts. In the case of the ISMD, the original polling loop is modified to include a series of state variable checks to determine what subset of operations should be performed in each of possible states in different configurations (Single, Parallel-Dual, and Serial-Dual).

5. Performance Evaluation

After tailoring the Duplex framework to ISMD, we evaluated the resulting system's fail-over performance for both Parallel-Dual and Serial-Dual configurations. The hardware platform of the ISMD being evaluated is a 1U industrial PC with a Pentium Celeron 500 MHz CPU, 64 MBytes of PC100 RAM, and two types of persistent storage: a regular hard disk, and a Disk On Module (DOM) unit based on Flash memory technology. The disk drive used was a WD200 5400 RPM IDE drive with 20GB capacity. The DOM units were DOM2000 modules from Adlink with a typical media transfer rate of 1.2 MBytes/s for write and 2.5 MBytes/s for read. The network interfaces used in Parallel-Dual configuration were Intel EEpro-100 fast Ethernet adapters and that in Serial-Dual configuration were dual-channel PCI-8124 adapters from Adlink. The heartbeat period is 20 msec and each heartbeat message is 64 bytes in size. This amounts to around 3 Kbps or heartbeat traffic which is very trivial compared to the overall 100 Mbps link bandwidth.

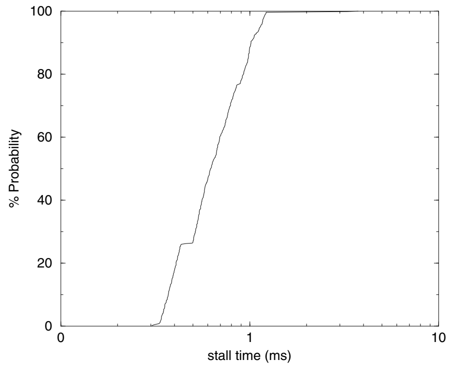


Figure 6. Service stall time when the persistent media is a hard disk drive. With more than 90% probability the logging operations incur less that 1 msec of stall time.

The performance evaluation metrics used were *fail-over latency* at different traffic conditions and the associated *data loss* in terms of bytes and packets. The main run-time overhead associated with fault tolerance was due to state logging, which causes the ISMD's service to be stalled whenever there is an update to the control state. These were measured by instrumenting the ISMD kernel to timestamp various relevant system events.

5.1. Parallel-Dual Configuration

The fail-over latency and packet loss counts at various traffic loads for a Parallel-Dual ISMD are shown in Table 1. The failure is introduced by inducing a power failure at the master device. The fail-over period represents the time between when the master device dies and when the slave device becomes the new master, This period can be decomposed into two parts – (1) Time between when the master device fails and when the slave device moves to the SubMaster state from the Slave state, and (2) Time for the slave device to evolve from the SubMaster state to the Master state.

It was observed that the transition time from the SubMaster to Master state was a constant 10 msec. The majority of the fail-over latency is due to the heartbeat timeout, which increases with the traffic load (Equation (1)). Without any traffic load, the shortest observed fail-over latency was about 100 msec. This time increases with the network traffic load. The fail-over latency was still less than 1 second for a traffic load of 70 Mbps. At the peak possible load of 100 Mbps the worst-case fail-over latency observed was around 1.2 seconds.

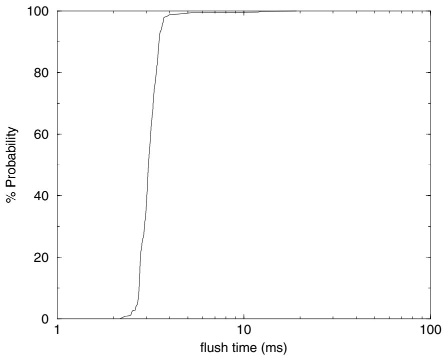


Figure 7. Time required to flush the log messages to the disk. With 90% probability the write operations incur less than 3 msec and almost always the operations take less than 4 msec.

The test traffic comprised of fixed packets of size 512 bytes each. The number of lost packets increases with the increase in load as expected. As the slave device in the Parallel-Dual configuration buffers the same set of packets as the master device, essentially the slave device acts as a warm replica and can take over the responsibility of traffic forwarding immediately after the failure of the master device is detected, i.e., when the slave device moves to the SubMaster state. The maximum measured packet loss count is around 70 Kpkts at the peak traffic load of 100 Mbps. For a more typical traffic load of 50 to 70 Mbps, the packet loss count ranges from 8 Kpkts to 26 Kpkts.

5.2. Serial-Dual Configuration

The fail-over latency and packet loss counts at various traffic loads for a Serial-Dual ISMD are also shown in Table 1. The failure is again introduced by a power failure at the master device. The failure detection and fail-over times were similar to the parallel configuration. This is understandable because the fault detection mechanism and the fail-over procedure are largely the same for these two configurations. However, the number of packets lost in the Serial-Dual configuration is higher than that for the Parallel-Dual configuration for the same traffic load. This is because the slave device in the Serial-Dual configuration blindly forwards the traffic without any additional processing in order to minimize the additional packet latency it introduces. As a result, the slave device in this configuration is no longer a warm replica of the master device, and all the packets buffered at the master device at the time of its failure are lost. Fortunately, this additional packet loss count becomes relatively less significant as the input traffic load increases, as shown in Table 1. The peak packet loss count is around 73 Kpkts at the maximum load of 100Mbps and from 30 Kpkts to 42 Kpkts for a traffic load of 50 to 70 Mbps.

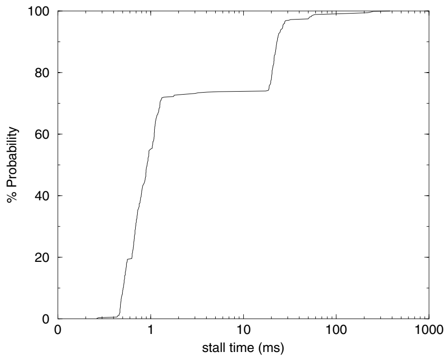


Figure 8. Service stall time when the persistent media is a Disk On Module unit. With more than 70% probability, the write operations incur less than 1 msec of stall time.

5.3. State Logging Overhead

The main run-time overhead of Duplex is the performance cost associated with state logging, since it takes some time to flush a log record to the disk. The design goal of Duplex is to mask this delay as much as possible so that its impact on the NAD service is minimized. Accordingly, we measure the disk flush cost of every logging operation, and the service stall time due to state logging that is visible to the NAD service. The difference between the two represents the effectiveness of Duplex's state logging implementation. Each logging operation involves a record of size 2048 bytes, which requires one disk block write. We measured 500 logging operations and plotted the probability distributions for the service stall time and the disk flush time.

The probability distributions for the service stall time and disk flush time when the media is a hard disk are shown in Figure 6 and Figure 7 respectively. When the persistent media is a hard disk drive the observed service stall time is less that 1 msec with 90% probability. In comparison, the actual disk flush time visible to the synchronous write process is around 3 msec with a probability of 90% . After eliminating extreme cases the maximum observed disk flush time is around 4 msec. The probability distribution for the service stall time and disk flush time when the media is a DOM unit are shown in Figure 8 and Figure 9 respectively. When the persistent media is a Disk On Module unit the observed service stall time is less than 1 msec with 70% probability, less than 10 msec with 90% probability and less than 100 msec for almost all of the operations. In contrast, the actual disk flush time is nearly constant at 100 msec for almost all of the operations. In both cases, the service stall time is significantly lower than the disk flush time, which shows Duplex's two-process state logging mechanism is indeed quite effective. The fact that the majority of logging operations take less than 1 msec for hard disks suggests that logging has absolutely no impact on the operation of the hosting NAD because its internal buffer memory can easily compensate a stall time of this magnitude.

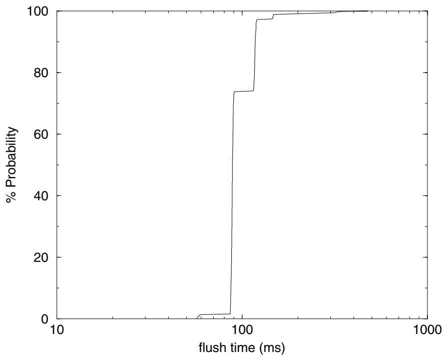


Figure 9. Time required to flush log messages to the Disk On Module unit. Almost all write operations require around 100 msec.

6. Conclusion

The network access devices on the access link of any enterprise to the Internet are critical resources that need to be carefully guarded. The goal of Duplex project is to develop a reusable fault tolerance implementation framework that can turn a generic NAD into its fault-tolerant counterpart with minimal customization. The Duplex framework provides NADs with protection against the following types of failures: system hardware, software, power, and network interface failure. In the Serial-Dual configuration, Duplex can tolerate up to two device failures without losing the access link's connectivity. To the best of our knowledge, Duplex is the first and perhaps only fault-tolerance implementation framework for network access devices.

We have successfully implemented an initial prototype of the Duplex framework and demonstrated its utility in a commercial traffic shaping system called ISMD from Rether Networks Inc. Performance measurements on the ISMD show that the fault-tolerance mechanism of Duplex framework indeed greatly improve the availability of the NADs and thus the access links. The worst-case fail-over time of 1 sec guarantees almost 100% access link availability even when the MTTF of single NAD units is as low as a few days. Moreover, this excellent fail-over performance comes with a negligible run-time overhead, less than 1 msec of service stall time due to state logging.

There are several directions we plan to pursue further based on the current Duplex framework. First, we plan to enhance the Duplex prototype to make it independent of the underlying operating system. Second, we plan to re-visit the state transition logic, and develop a generic API that can simplify the interaction between the hosting NAD and the state transition mechanisms in Duplex. Experiences from the ISMD project indicate that this part requires most of the customization efforts. Third, although the Duplex framework significantly reduces the implementation effort required to add fault tolerance to an existing NAD, it does not completely eliminate it. It is an open research question whether it is possible to build a program transformation tool that automatically inserts proper Duplex APIs at proper places based on a minimal amount of annotations provided by the application programmers. Finally, it will be interesting to explore whether it is possible to develop a similar framework for other types of systems than NADs.

References

[1] N. Aghdaie and Y. Tamir. Implementation and evaluation of transparent fault-tolerant web service with kernel-level support. In *Proc. of the IEEE Intl. Conf on Computer Commications and Networks, Miami, Florida*, October 2002.

[2] J. Bartlett. A nonstop kernel. In *Proc. 8th ACM SIGOPS SOSP*, 1981.

[3] K.J. Birman. Replication and fault tolerance in the Isis system. *ACM Operating Systems Review*, 19(5):79–86, 1985.

[4] A. Borg, W. Blau, W. Graetsch, F. Hermann, and W. Oberle. Fault tolerance under unix. *ACM Transactions on Computing Systems*, 7(1):1–24, Feb 1989.

[5] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 125–132, May 2001.

[6] D.M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available web server. In *Proc. of IEEE COMPCON'96, San Jose, CA*, pages 85–92, Feb. 1996.

[7] M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Oct. 1996.

[8] C. Hagen and G. Alonso. Highly available process support systems: Implementing backup mechanisms. In *Proc. of 18th IEEE Symposium on Reliable Distributed Systems (SRDS'99), Lausanne, Switzerland*, Oct. 1999.

[9] P. Higginson and M. Shand. Development of router clusters to provide fast failover in IP networks. *Digital Technical Journal*, 9(3), Winter 1997.

[10] R. Hinden, D. Mitzel, P. Hunt, P. Higginson, M. Shand, A. Lindem, S. Knight, D. Weaver, and D. Whipple. *Virtual Router Redundancy Protocol*. Internet Draft draft-ietf-vrrp-spec-v2-06.txt, Feb. 2002.

[11] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, Sep. 1990.

[12] T. Li, B. Cole, P. Morton, and D. Li. *Cisco Hot Standby Router Protocol (HSRP)*. RFC2281, March 1998.

[13] J.S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proc. of Usenix Technical Conference 1995, New Orleans, LA*, Jan. 1995.

[14] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C.M.R. Kintala. Checkpointing and its applications. In *Symposium on Fault-Tolerant Computing, Pasadena, CA*, pages 22–31, June 1995.