

TRANSPARENT HIGH AVAILABILITY FRAMEWORK FOR CLIENT-SERVER
APPLICATIONS ON WIRELESS AND WIRELINE NETWORKS

BY

DHEERAJ AHUJA

B.E., University of Delhi, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

To Shweta and my parents.

ACKNOWLEDGEMENTS

It is a pleasure to thank the many people who made this thesis possible.

I would like to express my deepest gratitude to my advisor, Professor Ravi Iyer, for his enthusiastic supervision and guidance throughout the course of this research. I am especially grateful to Dr. Zbigniew Kalbarczyk for his encouragement, support, excellent ideas, and the numerous hours he spent reviewing the thesis. A million thanks go to Keith Whisnant for his patience in explaining to me the deepest mysteries of the world of Chameleon, and for his tremendous help in the design and implementation phase of this project.

I would also like to thank Dr. Pankaj Mehra, Compaq Computer Corporation, for his confidence in me, and for getting me started on this project during my internship in the summer of 2001. Finally, I am forever indebted to my parents and my fiancée for their understanding and encouragement when it was most required.

TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. RELATED WORK	5
2.1 Fine Grained Fail-over	5
2.2 Reliable Sockets (Rocks)	6
2.3 Mobile Sockets (MSOCKS).....	6
2.4 Fault Tolerant TCP (FT-TCP).....	7
2.5 Migratory TCP (MTCP).....	7
2.6 Cluster-based Web Servers	8
2.7 Mobile TCP	8
3. OVERVIEW OF CHAMELEON.....	10
3.1 Introduction.....	10
3.2 Chameleon Software Framework.....	11
3.2.1 ARMOR Process Architecture.....	11
3.2.2 Types of ARMORs	12
3.2.3 Execution Environment.....	13
3.3 Error Detection and Recovery.....	13
3.3.1 Node Failure.....	14
3.3.2 Application Failure.....	14
3.3.3 ARMOR Failure.....	14
4. SUPPORTING CLIENT-SERVER APPLICATIONS	15
4.1 Transparent Hook-up of Client and Server Applications.....	16
4.2 Design and Description of New Elements in Chameleon.....	19
4.2.1 APP_COMM_CLIENT	19
4.2.2 APP_COMM_SERVER	21
4.2.3 APP_SERVER_DB	23
4.3 Transparent Support: Socket Abstraction Layer	24
4.3.1 Server Socket Calls	26
4.3.2 Client Socket Calls.....	27

4.3.3	Common Socket Calls.....	28
4.4	End-to-End Message flow.....	29
4.5	Error Detection and Recovery.....	31
4.5.1	Server Process Failure.....	32
4.5.2	ARMOR Failure.....	33
4.5.3	Daemon Failure.....	34
4.5.4	Node Failure.....	35
4.6	Demo Application: Audio Streaming Server.....	35
5.	CHAMELEON FOR WIRELESS APPLICATIONS	37
5.1	Introduction.....	37
5.2	Robust connectivity over Wireless Networks.....	38
5.3	ARMOR-based solution to TCP over Wireless: Proxy ARMOR.....	39
5.4	Supporting TCP and UDP in Chameleon	41
5.5	Reliable UDP: Introducing Acknowledgements.....	44
6.	PERFORMANCE MEASUREMENTS.....	45
6.1	Experimental Setup.....	45
6.1.1	Wireline Network Testbed.....	45
6.1.2	Wireless Network Testbed.....	46
6.2	Performance Evaluation.....	47
6.2.1	Throughput.....	48
6.2.2	Initialization	50
6.2.3	Connection	50
6.2.4	Recovery Time.....	51
6.3	Execution ARMOR Proxy versus Daemon-based Proxy.....	53
7.	CONCLUSIONS AND FUTURE WORK.....	56
	LIST OF REFERENCES	58
	APPENDIX A: Socket Class Interface.....	60

1. INTRODUCTION

The client-server application paradigm has been popular for a long time and is becoming even more so with the spectacular growth of the Internet. Along with this growth, the basic client-server applications have seen a change in the way they are expected to operate. The client is no longer interested in the identity of the physical machine or the resource providing the service. As a result, there has been an increasing trend to decouple the service itself from a physical entity (specified by the IP address, for example) delivering that service.

In the traditional client-server model, the client is concerned about the remote entity it is connected to, whereas in the emerging model, the client is interested only in the service, and wants the service to be always available. The client might desire to switch between multiple servers offering the service during the lifetime of the connection as soon as it perceives a drop in the quality of service from the current server. This drop may be a result of overloading the server or a processor crash for example.

A useful metric in defining the quality of a service is *availability*. Availability is formally defined as the probability that a system is functional at a given point in time in a specified environment. To keep the service up and running continuously requires efficient and robust protection against various failure scenarios including application failures, traffic congestion, and node crashes. A majority of the services conforming to the client-server paradigm are built over the transport layer protocol TCP (Transmission Control Protocol [24]). TCP is a connection-oriented reliable transport protocol built over the unreliable, best-effort connectionless network protocol IP (Internet Protocol). TCP provides flow control between the sender and the receiver, thereby enabling entities with different speeds to communicate.

Figure 1.1 illustrates a typical TCP connection established between two processes running on networked hosts.

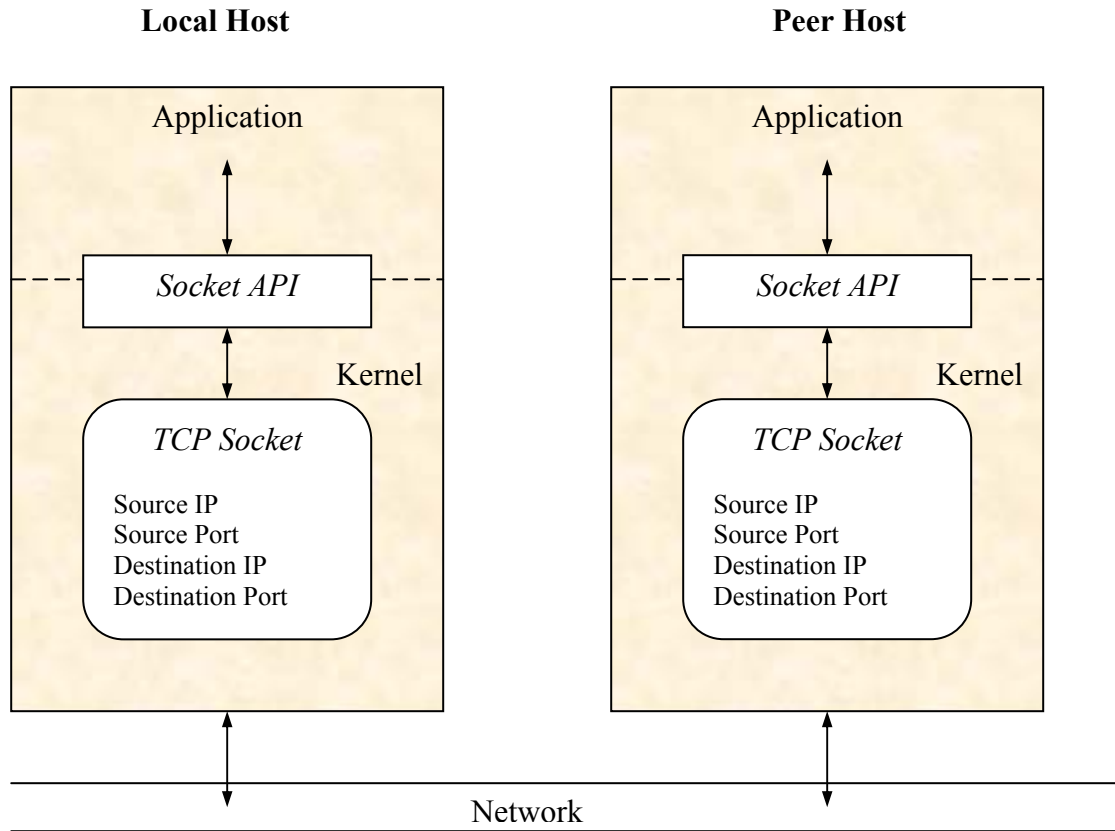


Figure 1.1: Established TCP connection between two hosts

The basic feature of a TCP connection is that it is associated with a pair of IP addresses, one for each endpoint of the connection. This feature leads to several shortcomings given the changes in the client-server model:

- TCP creates an implicit binding between the service and the IP address of the physical machine running the service. This model poses constraints in an environment where the client is primarily interested in the service rather than the exact location and identity of the server. If the server fails or becomes overloaded, it becomes impossible for the client to switch to another server offering the same service without terminating the existing connection and re-establishing a new one.

- Mobile computer users routinely perform actions that can lead to TCP connection failures. Examples leading to connection failures include: (1) link unavailable (when the user moves out of wireless range), (2) link failure (e.g., modem dropping a connection), (3) host suspended (e.g., mobile host goes to standby mode), (4) mobile host moves to a new subnet; this requires a new IP address to be assigned, which leads to the failure of the existing TCP connections, and (5) recovery from a host or an application crash using process migration to restart the application on a new host.

Approach and Contributions of this Research

This thesis proposes a transparent, end-to-end, high-availability framework for client-server applications operating over wireline and/or wireless networks. The framework provides robust connectivity to client-server applications, and protects the applications and the service against various failure modes by supporting transparent detection and recovery from errors. The recovery procedure includes re-establishing the lost connections between the client and server, besides recovering the application processes.

While techniques have been proposed for providing a robust TCP connectivity over mobile networks, none of those schemes deals with the issue of improving the availability of the application. Our scheme emphasizes availability, while preserving mobility and persistent connectivity over wireless networks.

One of the distinguishing features of our framework is the transparency with which it offers the services. Both ends of the connection (server and client) are completely unchanged. The solution is implemented in user-level code and can be used by any TCP-based client-server application without reprogramming, recompilation, or static re-linking. A new socket library¹ to be dynamically linked is provided to the client and server applications to support use of the framework.

¹ The new socket library preserves the standard interface of a C++ library provided by the operating system kernel.

The proposed high availability framework builds upon the ARMOR-based Chameleon infrastructure [1]. New concepts and ideas for providing high availability to networked client-server class of applications are incorporated into the system as services provided by ARMORs (Adaptive Reconfigurable Mobile Objects for Reliability). The scheme is proxy-based and introduces two redirection proxies between the client and server processes. These proxies are a special service provided by the ARMORs to help maintain track of the open TCP connections between the clients and servers. A failure at either end of the connection is detected by ARMORs. The recovery phase uses the information stored in the redirection proxy elements to restore the connections that existed before the failure. For an application demanding persistent connectivity in the wireless environment where connections might drop due to mobility of the client as explained earlier, this scheme keeps the connections alive during periods of unstable connectivity of the client.²

Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 discusses the related work in the area of dependable systems and robust connectivity to client-server applications. Chapter 3 provides a brief overview of Chameleon and the various error detection and recovery capabilities it offers. A detailed design of the proposed end-to-end, high-availability infrastructure for client-server applications is presented in Chapter 4. Chapter 5 explains the issues of using TCP over wireless networks and discusses the improvements made to the Chameleon infrastructure to make it more wireless-friendly. Chapter 6 presents the experiments and performance measurements done to evaluate the feasibility of the proposed design. Chapter 7 concludes the thesis and suggests possible future work.

² In the traditional case, if TCP detects loss of wireless connectivity through retransmission timeouts, it terminates the connection and this response is visible as an error returned by the socket calls to the application using the socket API.

2. RELATED WORK

The issues related to providing highly available services to clients have been approached in many ways. Incorporating high availability in services through transport layer support has been very common. These methods include incorporating fault-tolerance in TCP [6], proxy-based schemes to protect TCP connections for mobile networks [15], cluster-based schemes [14,16], and some application specific solutions [13]. TCP/IP connection handoff protocols have also been used in the context of mobility extensions to TCP [10,11,12]. We describe each of these approaches in the following sections and compare them with our solution wherever applicable.

2.1 *Fine Grained Fail-over*

This work describes a an architecture that enables migration of HTTP connection endpoints within a pool of support servers that replicate per-connection application state [13]. The new server initiates connection migration on events such as the primary server's failure. This is achieved by adding a new HTTP-aware kernel module in the transport layer to extract information from the application data stream to be used for connection resumption. This solution does not require any changes to the server application. However, the approach is not generic, as the transport layer must have knowledge of the upper-layer protocols. The design requires that the information in the application data stream is (a) sufficient to describe the connection state of the application, and (b) compatible with respect to the in-kernel module. This scheme fails if the application-level data is not accessible to the transport layer, because of encryption for example.

In our scheme, we do not need to make any changes to the kernel, or to the applications. The instantiation of the recovered server triggers the event to re-establish the connections that existed between the original server and the clients. In addition, our approach works for all classes of applications based on the client-server paradigm and is not restricted to HTTP-based applications.

2.2 *Reliable Sockets (Rocks)*

Reliable Sockets [8] are a portable, user-level replacement to the sockets interface that transparently protects applications from network connection failures. Reliable sockets preserve TCP connections across failures that commonly arise in mobile computing, including host movement, network reconfiguration, link failures, and extended periods of disconnection. *Rocks* is a transparent scheme, and like our solution, does not require any modifications to application binaries or to the operating system kernel. Reliable sockets can automatically recover a TCP connection if one of the two endpoints has moved. In the event of a process crash or a node crash, this scheme, like most other schemes described in this section, depends on an external entity to detect the crash, recover the application, and migrate the application to a new node. The scheme is modeled to provide persistent connectivity but does not provide for a complete solution to provide high availability for client-server applications.

2.3 *Mobile Sockets (MSOCKS)*

The emphasis of the MSOCKS project is on providing continuity in service sessions to mobile clients equipped with multiple network interfaces [15]. MSOCKS is a proxy-based system to protect the TCP connections between two processes. The mobile client is connected to the proxy with more than one network link. The proxy has a kernel modification called *TCP splice*. The TCP connections are spliced at the proxy by means of sequence number mapping, which enables a client to close its end of a connection and to establish a new one without disrupting the server. Though the project addresses the problem of managing a service session by more than one TCP connection, it is not concerned with providing high availability to services and the client processes. Unlike our work, this project does not provide for detection of abnormal behavior or errors in the application end-points, nor does it deal with recovery and migration of those processes. Hence, the problems of error detection and recovery of connection end-points do not appear in the MSOCKS work. In contrast, these are primary issues in our work, in addition to providing for persistent connections to the client and server applications. Additionally, our scheme is entirely application-level based, eliminating the need to modify the operating system kernel.

2.4 *Fault Tolerant TCP (FT-TCP)*

FT-TCP [6] is a scheme for transparent recovery of a crashed process with open TCP connections. A wrapper around the TCP layer intercepts and logs the read requests by a process for replay during recovery and shields the remote endpoint from failure. The work is very similar to our approach, but we would describe the problems we address to be complementary to the one on which this project focuses. Like the MSOCKS project, the FT-TCP project does not deal with the error detection and recovery of the crashed process endpoint of the connection. It only provides for a technique to recover the lost connections of the recovered process. FT-TCP does not address the issue of providing high availability to services and clients in the emerging client-service model. The emphasis is on fault tolerant TCP connections and not on a high availability infrastructure for client-server class of applications.

2.5 *Migratory TCP (MTCP)*

The MTCP project [7] proposes a transport-layer protocol designed for building highly available Internet network services. The protocol provides a mechanism to transparently migrate the server endpoint of a live TCP connection. The connection migration is dynamic and can occur multiple times during the client-server session. The client process initiates the migration when it perceives a degradation of service. The solution aims to resume the server process by migrating the connection to a new host. The migration of the connection is transparent to the application attached to the connection at the client endpoint. However, this solution requires the server application to be rewritten to recognize and respond to the connection migration requests. The server application must use the specialized API provided and be programmed according to a particular programming model. In contrast, one of the main achievements of our proposed solution is the transparency it provides; neither end of the connection needs to be modified. This high level of transparency is achieved by dynamically re-linking the applications to a new socket library that redirects traffic through the framework and helps maintain the state of all the connections that exist. This information is used during recovery of either endpoint to re-establish the connections that existed before the failure

happened. The MTCP scheme protects only the service endpoint of the connection from failures. Our scheme protects the client as well as the server process with equal ease and transparency.

2.6 *Cluster-based Web Servers*

A widely accepted solution to providing high availability to very crucial services in the Internet is to use cluster-based schemes. The Location Aware Request Distribution project [9] proposes TCP connection handoff in clustered HTTP servers for distributing incoming requests from the front-end machine to back-end server nodes. It is a single handoff scheme in which a connection endpoint can migrate during the connection setup phase, whereas in our solution connections can migrate dynamically at any time during the course of the connection.

2.7 *Mobile TCP*

Mobile TCP approaches do not consider the task of migrating the connection endpoints between physically distinct machines. Hence they do not address the needs of clients that are not concerned about receiving service from a particular server. Some Mobile TCP techniques enable the endpoints of a TCP connection to be re-assigned a different IP address; these include the TCP Migrate option [12], Mobile TCP sockets [19], and Persistent Connections [18]. The TCP migrate option is a kernel extension to TCP and is not widely deployed. It does not handle TCP failures, and it requires the disconnected peers to reconnect before the underlying TCP layer aborts the connection. TCP might abort a connection for reasons such as too many retransmissions (kernel might abort a connection after a preset limit on the number of retransmission is exceeded) or the reception of a reset message from the peer. However, the scheme depends on external support to detect and initiate reconnection with the disconnected peer.

Mobile TCP sockets and Persistent Connections interpose a library between the application and the sockets API, similar to our approach of an abstract socket library. This library preserves the illusion of a single, unbroken connection over successive connection instances.

Between connections, Mobile TCP sockets preserve the in-flight data, while Persistent Connections makes no such attempt. Both of these techniques depend on external support to reestablish contact with a disconnected peer, and neither interoperates safely with existing applications.

An alternative to TCP-specific techniques, *Mobile IP* [25] routes all IP packets, including those used by TCP, between a mobile host and ordinary peers by redirecting the packets through a *home agent*, a proxy on a fixed host with a specified kernel. As described earlier, the problem addressed in our approach is much more generic and does not deal only with wireless connections. It can be used by any client-server based application that needs the service to be highly available, irrespective of the type of network (wireless or wireline). For mobile applications it additionally provides a robust framework for maintaining persistent TCP connections despite the frequent loss of physical-layer connectivity due to loss of radio signals.

3. OVERVIEW OF CHAMELEON

3.1 Introduction

In this chapter, we introduce an adaptive fault tolerance infrastructure called *Chameleon*, which allows different levels of availability requirements to be supported concurrently in a networked environment. The Chameleon environment provides reliability and dependability to applications through the use of self-checking processes called ARMORs – Adaptive, Reconfigurable, Mobile Objects of Reliability. ARMORs control all operations in the Chameleon environment and provide error detection and recovery to the application and to the ARMOR processes themselves. The environment allows for the construction of distributed applications around the ARMOR processes executing on a network of heterogeneous, non-fault-tolerant nodes. Additional details about the Chameleon system can be obtained from [1,2,3].

The use of ARMORs to provide dependability to a user application allows for implementation flexibility. Since fault tolerance techniques are encapsulated in ARMORs, the runtime Chameleon environment need only use those ARMORs that provide the required level of error detection and recovery. In addition to providing a highly customizable environment, this also reduces the overhead introduced by Chameleon to a bare minimum, dependent on the level of fault tolerance desired by the application. The modular framework achieved by the use of ARMORs also provides a convenient mechanism whereby new functionality can be introduced into the system at a later time without disturbing the existing functionalities. The current work of providing an end-to-end high availability environment for client-server applications takes advantage of these architectural features.

The remainder of this chapter gives a brief description of the internals of Chameleon and the ARMOR process architecture. We present some relevant details of the messaging mechanism used in Chameleon and introduce the concept of Elements and Operations. Finally, we also describe some of the error detection and recovery capabilities of Chameleon.

3.2 *Chameleon Software Framework*

This section introduces the key components of the Chameleon framework and describes the execution environment.

3.2.1 **ARMOR Process Architecture**

An ARMOR is a multithreaded process internally structured around objects called *elements* that contain their own private data and provide elementary functions or services. Together, the elements constitute the functionality that defines the ARMOR's behavior. All ARMORs contain a basic set of elements that provide the core functionality to (1) implement reliable communication between ARMORs, (2) respond to heartbeats from the local daemon, and (3) capture the ARMOR state. Specific ARMORs extend this functionality by adding extra elements.

Elements and Operations

The state in an ARMOR process is organized in elements and controlled through operations. Elements are objects that contain (a) local data accessible only to the element and (b) functions that access the local data. These functions are the only means through which an element's data can be modified or accessed. Elements are passive entities and do not directly invoke functions of other elements in the distributed system. They provide primitive services in response to the operations delivered to them. An operation that is delivered to an element invokes one of the element's functions and brings about a change in the state of the element.

Messaging

The host daemons perform the actual network communication in the Chameleon environment. The existing implementation is built on TCP/IP, and this thesis work aims to provide a communication framework that can use both TCP and UDP based communication.

Each ARMOR is addressed by a unique identification number, allowing messages to be sent to it without prior knowledge of its physical location. To send a message, the ARMOR passes

the message to the local host daemon through an IPC channel. The daemon does a lookup and translates the ARMOR id into the location of the node on which the ARMOR resides. Before sending the message over the TCP socket, the daemon adds a header containing the source and destination identifiers and a checksum. When receiving messages, the daemon does a validity check on the message and forwards the message to the appropriate local ARMOR over another IPC channel.

Each incoming message results in the ARMOR process spawning a new thread to process the message. The thread delivers the message to the elements subscribing to that message within the ARMOR process. Structurally the Chameleon messages consist of two parts:

1. *Bundles of operations:* The operations to be performed on the destination ARMOR are grouped together to form message bundles. The destination ARMOR delivers each operation to the elements that have subscribed to the operation type, in the order they appear in the bundle.
2. *Payload fields:* The message contains a general payload area for storing data, which can be read from and written to by the elements while they process the operations in the message.

While processing an operation, an element can change the local state of the element or the payload fields, and it can change the control flow of the operations by pushing new operations on the current bundle. Adding operations and bundles in this manner results in a nested invocation of functionality in a distributed environment.

3.2.2 Types of ARMORs

The Chameleon environment used for executing client-server applications consists of the following kinds of ARMORs:

Fault Tolerance Manager (FTM). A single FTM executes on one of the nodes and is responsible for recovering from ARMOR and node failures by migrating the processes to another node. The FTM sends out heartbeat messages to the daemons in the network to detect node failures.

Daemons. Each node in the network executes a daemon process. Daemons are the gateways for inter-ARMOR communication, and they detect failures in the local ARMORs. The main functionalities of the daemon include: (1) installing other ARMOR process on the node, (2) caching the location of remote ARMORs, (3) routing messages to remote ARMORs, (4) sending heartbeat messages to local ARMORs to detect failures, and (5) notifying the FTM to initiate recovery of failed local ARMORs.

Execution ARMOR. A local Execution ARMOR directly oversees each application process. The main role of the Execution ARMOR is to: (1) launch the application process, (2) detect crash failures in the application process, and (3) notify the FTM if the application process fails.

The ARMOR architecture permits the functionality of several ARMORs to be merged into a single process. For example, the functionality of the daemon and the Execution ARMOR that execute on a node can be combined into a single ARMOR. This is sometimes useful (see Chapter 6), as it reduces the number of processes in the system as well as the number of indirections in the inter-ARMOR communication.

3.2.3 Execution Environment

Any network of unreliable nodes can be configured to participate in the Chameleon environment. Before executing any applications, the Fault Tolerance Manager (FTM) is installed on a node in the network. Daemons are installed on each node in the network and then registered with the FTM. Once the basic setup is complete, the TCL scripts launch the applications through the Chameleon environment, prompting the FTM to install Execution ARMORs on the appropriate nodes to support the application.

3.3 Error Detection and Recovery

Chameleon is designed to recover from transient and permanent faults. Such faults can result in the failure of the hardware or operating system, the application (abnormal termination), or even the Chameleon ARMORs that provide the high reliability and fault tolerance techniques.

Thus the ARMOR processes need to be protected against faults, a concept popularly known as *checking the checker*. This subsection gives a brief description of the hierarchy of error detection and recovery strategies in Chameleon. For details, refer to [3].

3.3.1 Node Failure

To detect a node failure, the FTM periodically sends a heartbeat message to each daemon in the network. When the FTM fails to receive an acknowledgement from the daemon, it raises a node failure indication. The FTM initiates recovery by migrating the ARMORs and the application processes that were executing on the failed node to other working nodes in the environment.

3.3.2 Application Failure

The user application may experience an *abnormal termination error* (due to an unhandled exception), a *value-domain error* (executing correctly by producing incorrect values due to data corruption), or a *time-domain error* (no forward progress). On detecting application failure, the Execution ARMOR notifies the FTM to initiate recovery. After an error in the application has been detected, the Execution ARMOR overlooking that application is primarily responsible for restarting the user application either from the start or from a previous checkpoint (if checkpointing is enabled).

3.3.3 ARMOR Failure

To ensure the correct execution of the Chameleon ARMORs, the local daemon monitors all the ARMORs it installs on the local node. Once a failed ARMOR is detected, the daemon notifies the FTM to initiate recovery from the most recent checkpoint. The FTM may reinstall the ARMOR on the same node, on a different node, or on a different platform.

4. SUPPORTING CLIENT-SERVER APPLICATIONS

This chapter gives a detailed design description of the tunneling support for application messages through the ARMOR-based Chameleon infrastructure. Most client-server applications use socket-based communication protocols. The two most commonly used socket protocols are TCP and UDP. Of particular interest in relation to providing a high-availability to client-server applications is the use of TCP sockets. Since TCP (Transmission Control Protocol) is a connection-oriented protocol, the transfer of messages begins only after a connection has been established between the client and the server via a 3-way handshake mechanism. In contrast, UDP (User Datagram Protocol) is connectionless protocol in which the message exchange can commence without any synchronization phases. Since the notion of a state is associated with every TCP connection, recovering from a crash is more complex for the TCP server than for a UDP server. TCP connections between the server and its clients are lost in the crash and need to be re-established as a part of the recovery phase.

Providing support to recover from such crashes requires in-depth understanding of the issues involved in transparently re-establishing client-server connections. The necessary level of support can be achieved by making the client and server intelligent enough to detect and recover from connection losses. In this scenario, however, the application programmer needs to consider these issues during design phase and add the necessary support to the application code. There are numerous widely used client-server applications that do not provide the ability to transparently recover from server crashes. Very often, the client and server need to be restarted, which is not an acceptable solution if the service provided is to be highly available.

Using the framework proposed here, the application programmer can concentrate on the application-level issues and the problem at hand without having to worry about incorporating the error detection and recovery features into the applications. Often the complexity of providing for these features might well exceed the complexity of the original problem, which

again emphasizes for the need of such a flexible middleware solution. The following sections highlight the design details of the proposed framework.

4.1 Transparent Hook-up of Client and Server Applications

A brief overview of the architecture of Chameleon high-availability framework was provided in the previous chapter. As mentioned earlier, the ARMOR-based Chameleon infrastructure is used as a foundation for building the framework to provide fault-tolerance support to socket-based client-server applications. Specifically, new concepts and ideas are incorporated into the system in the form of new elements, which make the ARMOR a transparent gateway for client-server applications to communicate with each other.

Figure 4.1 illustrates the support available to black-box applications in Chameleon. The fault tolerance can be achieved using application replication, voting on the computation results (fault masking), or detection of an abnormal termination and restart of the application. Although this support works well for stand-alone applications, it does not work very well in case of socket-based applications. The ARMOR-based infrastructure has no control over the application-level message communication; the client and the server communicate via the socket library provided by the operating system itself. Thus in case of an abnormal termination (crash) of either end of the connection, ARMORs cannot make a clean restart, which would require re-establishing the lost connections between server and the client. The reason why Chameleon fails in such a scenario is because ARMORs overlooking the application have no information about the external connections between the client and the server and thus cannot transparently recover from the crash.

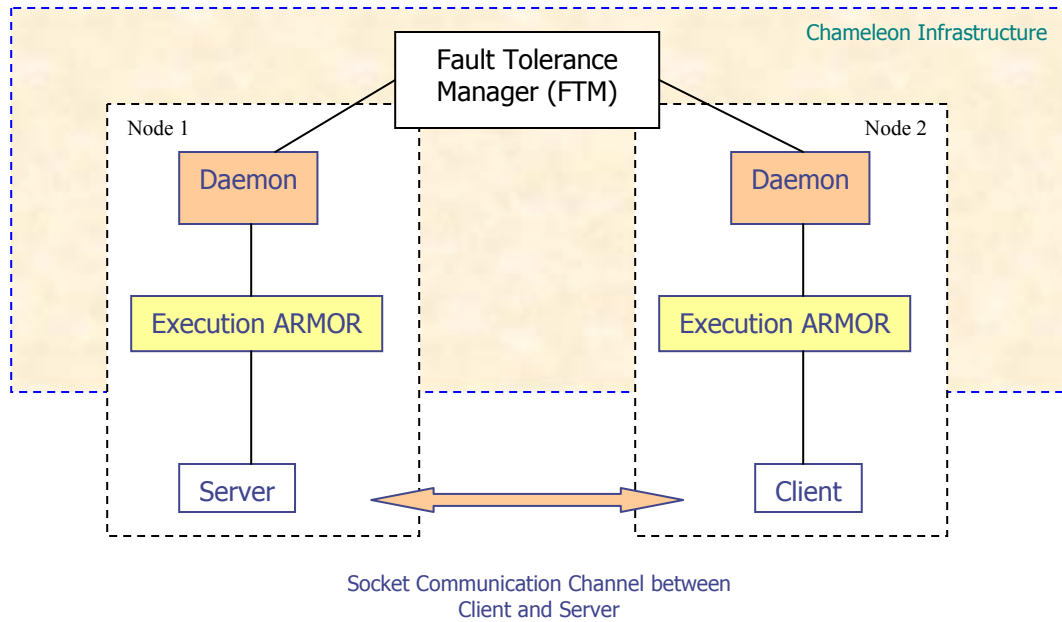


Figure 4.1: Existing Scenario of Communication between Client and Server

For the framework to be able to transparently recover the application from a crash in a client-server setup, it needs to be made aware of the connections the application has established with the external world. This is precisely the job of Daemons in the Chameleon infrastructure. Daemons are entities resident on each of the participating nodes and serve as a primary gateway for all communication between ARMORs in the Chameleon environment.

Our main goal in this work is to be able to provide for this functionality in a completely transparent way, i.e., the application requires no modifications. We achieve this goal by providing special ARMORs (or elements which could become a part of the Execution ARMOR or Daemon) that allow the client and server side of the application to transparently hook themselves to the infrastructure. The process is transparent to the user level, and the applications still act as though the communication is being done through the operating system level sockets. However, in reality, the reconfigured gateway ARMORs intercept the messages exchanged between the client and the server and reroute them through the Chameleon infrastructure. The ARMOR behaves like a redirection proxy for the other end of the connection. In effect, the end-to-end connection is now broken into a three-level connection: from server to the server-proxy, from the server-proxy to the client-proxy (this connection is

provided by Chameleon), and from the client-proxy to the client. The new scheme for the message communication between the client and server applications is illustrated in Figure 4.2.

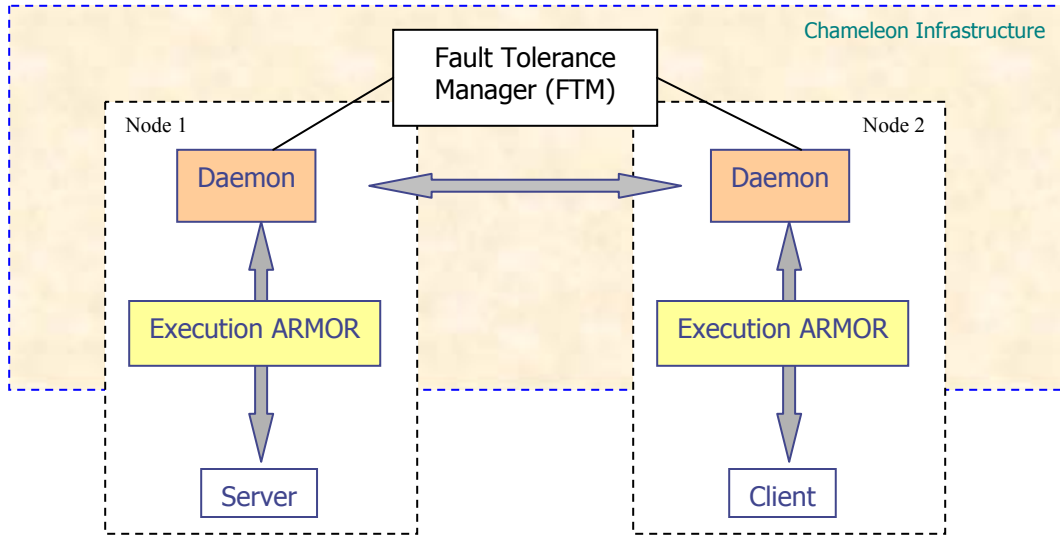


Figure 4.2: Proposed Mechanism for Chameleon-aware Communication

The following three steps establish the connection between the client and server:

1. When the server is launched, it links itself to the Execution ARMOR on its host. The server registers its IP address and port number with the FTM along with the ARMOR identification number. During this registration phase, the IP address and port number of the server are mapped to the ARMOR id and stored in a runtime table maintained by the FTM.
2. When the client application is launched and it issues a Connect system call, the Execution ARMOR intercepts the call. This ARMOR then inquires about the actual location (id of Execution ARMOR overlooking the server) of the server process and then reroutes the connection request to the server.
3. Once the connection establishment phase is complete, all send and receive socket calls are intercepted by the Socket Abstraction Layer (described in Section 4.3) and the application-level messages are tunneled through the Chameleon infrastructure. In

effect, there is a local connection between the server and its Execution ARMOR and another connection between the client and its Execution ARMOR, with the ARMORs communicating with each other using the Chameleon messages.

The ARMORs behave like proxies for the other end of the connection. Both the client and server applications are unaware of this redirection and thus continue to behave as they would in case of a traditional socket connection.

4.2 Design and Description of New Elements in Chameleon

To provide transparent support to the client and server application, several new elements were added to the library of elements in Chameleon environment. These elements, when integrated with the existing Daemons, become fully functional proxies for the client and server applications. The application can transparently attach itself to the hook-up points provided in the architecture and communicate with the peer entity via messages tunneled through the base architecture. We now describe in detail the design and implementation details of the new elements, simultaneously explaining how they provide for the transparency in the framework.

4.2.1 APP_COMM_CLIENT

[Communication Hook-up Interface for the Client Application]

This element provides the primary hook-up interface to the client-side applications. The APP_COMM_CLIENT element can be a constituent of either the Execution Armor or the Daemon on the client node. Figure 4.3 illustrates the detailed design of the element. During the initialization phase, the element creates a message queue to receive the incoming connection requests from the client applications running on its node. It then spawns a new thread to monitor this message queue for new messages from the clients.

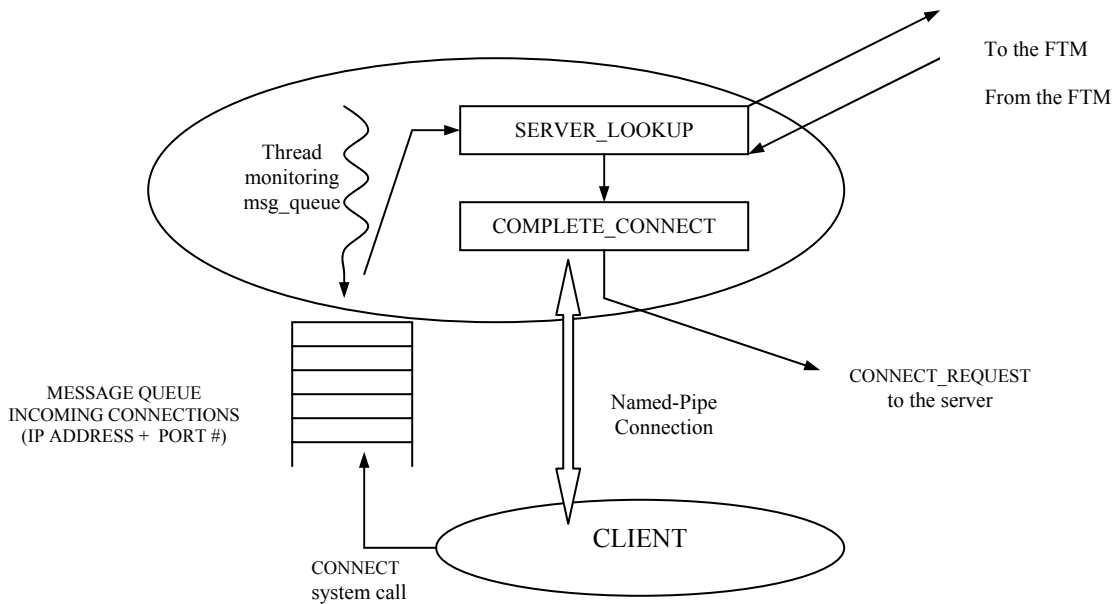


Figure 4.3: Design and Implementation of the APP_COMM_CLIENT Element

The client application is launched on the node where an ARMOR containing the APP_COMM_CLIENT element has already been initialized. The socket *connect* system call issued by the client process is captured by the Socket Abstraction Layer (described next), which sends a message to the APP_COMM_CLIENT element’s message queue. The message contains the IP address and the port number of the server host with whom the client wants to connect. In addition to sending this message, the client opens up a named pipe connection and waits for the ARMOR to connect to it.

On arrival of a new message into the message queue, the thread blocked on the message queue is reactivated. This thread extracts the IP address and port number of the server process from the message and then dispatches a server lookup message (*MSG_APP_LOOKUP_SERVER_INFO*) destined to the FTM. The APP_SERVER_DB element in the FTM maintains a runtime table of all the available servers in the system and subscribes to the lookup message. It responds back to the APP_COMM_CLIENT element at the client node with the id of the ARMOR associated with the server process. Upon receiving the id of the ARMOR associated with the server, the element dispatches a connection request message to

the remote server, and completes the connection process on the client side by connecting to the open named pipe. As a first message, it notifies the client of the server's armor-id.

Once the connection is established on both ends, the element launches a proxy thread, which monitors the named pipe connection between the client and the element. The sole purpose of this thread is to act as a proxy of the server to the client. Whenever the client sends a new message to the server using the *write* socket call, the Socket Abstraction Layer redirects the message on to the named pipe connection. The proxy thread receives this message and simply forwards it to the server side ARMOR, which then delivers the message to the server process.

If the server is not present in the system, the armor-id returned by the FTM is -1 . When the client application receives the armor-id of the server, it is checked against this value. If the server is non-existent, the client closes the connection and notifies the user about the error condition.

4.2.2 APP_COMM_SERVER

[Communication Hook-up Interface for the Server Application]

This element provides the primary hook-up interface to the server-side applications. The APP_COMM_SERVER element can be a constituent of either the Execution Armor or the Daemon on the server node.

Figure 4.4 illustrates the detailed design of the element. During the initialization phase, this element creates a message queue to receive a message from the new servers coming up on this node in the system. It then spawns off a new thread to monitor this message queue for new messages from the upcoming servers.

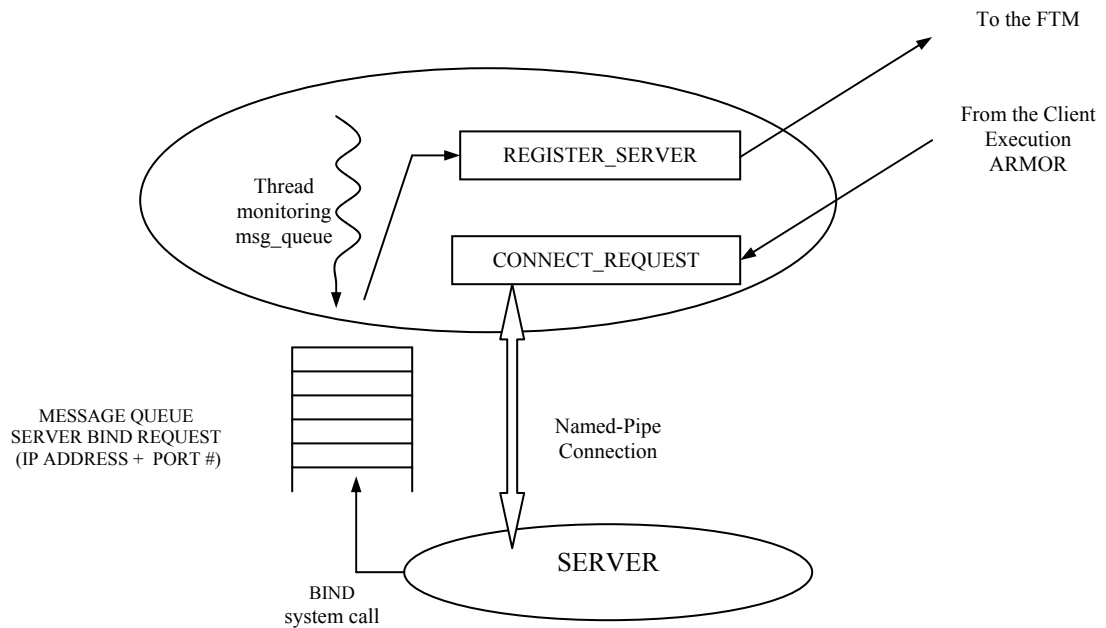


Figure 4.4: Design and Implementation Details of the APP_COMM_SERVER Element

The server application is launched on the node where an ARMOR containing the APP_COMM_SERVER element has already been initialized. The socket *bind* system call is intercepted by the Socket Abstraction Layer, which sends a message to the APP_COMM_SERVER element's message queue. The message contains the IP address and the port number on which the server process wishes to bind and wait for client connection requests. The blocked thread on the message queue in the APP_COMM_SERVER element is reactivated as a result of this new message. It extracts the IP address and the port number of the server process and sends a register server message (*MSG_APP_REGISTER_SERVER*) to the FTM, which adds the new server in its runtime table. It associates the armor-id of the Daemon, which sends this register message with the IP address and port of the server. Upon receiving a server lookup message from the client side, it responds with this armor-id.

When the server issues an *accept* system socket call, the socket abstraction library opens up a named pipe from the server process end and waits for a client to complete the connection. When the client-side armor sends a connection complete request to the server-side ARMOR,

the APP_COMM_SERVER element completes the named pipe connection that was initiated earlier by the server process.

Once the connection between the client and server is established, the element launches a proxy thread as in the case of the client, which monitors the named pipe connection between the server and the element. The purpose of this thread is to act as a proxy of the client to the server. Whenever the server sends a new message to the client using the *write* socket call, the Socket Abstraction Layer redirects the message on to the named pipe connection. The proxy thread receives this message and forwards it to the client-side armor, which then delivers the message to the client process.

4.2.3 APP_SERVER_DB

[Runtime Database of the Existing Servers in the Network]

This element is made part of the Fault Tolerance Manager (FTM). The main function of the element APP_SERVER_DB is to maintain a runtime table of all the existing server processes in the Chameleon environment.

The data structure used by this element is a hash table addressable on the IP address and port number of the server process. The table maps the IP address and port number to the armor-id of the corresponding ARMOR. It provides an interface to add new servers to the list, remove existing servers, and query about the existing servers in the environment. The response to such a query contains the armor-id of the ARMOR that is associated with the server process. All messages intended for the server are sent to this ARMOR, which in turn forwards the messages to the server process.

When the server process is launched and tries to bind itself to an IP address and port number on the system, the Socket Abstraction Layer captures this request and sends a message to the APP_COMM_SERVER element in the Daemon on that node. The APP_COMM_SERVER element in turn sends a *register new server* message to the FTM. This message contains the IP address and port number of the server as well as the armor-id of the Daemon. The

APP_SERVER_DB element in the FTM adds the mapping from IP address and port number to the corresponding armor-id that can be used to reach the server into its runtime hash-table.

When the client process is launched, it eventually calls the *connect* system call with the IP address and port number of the server it wants to establish a connection with. This message is intercepted by the Socket Abstraction Layer and sent to the APP_COMM_CLIENT element in the Daemon on that node. The APP_COMM_CLIENT element then sends a *server lookup* message to the FTM. The APP_SERVER_DB element in the FTM does a local lookup into the runtime hash-table it maintains and responds with the armor-id of the Execution ARMOR associated with the server process, or -1 if the server does not exist on the IP address and port number specified in the query.

4.3 *Transparent Support: Socket Abstraction Layer*

To provide support for existing client-server applications in a transparent manner, a Socket Abstraction Layer has been developed. This layer consists of wrapper routines written in C++ for the UNIX socket system calls and resides above the TCP layer in the TCP-IP protocol stack. The abstraction layer is provided as a library of the commonly used TCP socket calls. In the current implementation, the application needs to be statically re-linked with this new socket library instead of using the UNIX system socket library. Ideally this library could be provided as a dynamic link library, thus providing a fully transparent mechanism to reroute application-level data through the Chameleon infrastructure. No code changes or modifications are needed at the application level, assuming the application has been coded using the standard C++ socket APIs. The wrapper routines intercept the socket calls made by the application and reroute these requests to serve our purpose. In this section, we describe the details of the socket calls that are provided in the abstraction layer. The header file with the signatures of all the functions in the class *Socket* of the standard UNIX socket library is provided in Appendix A.

As an overview, Figure 4.5 shows a timeline of the typical scenario of a connection-oriented transfer [23]. The first step is to start the server process. Some time later, the client process is launched to connect to the server.

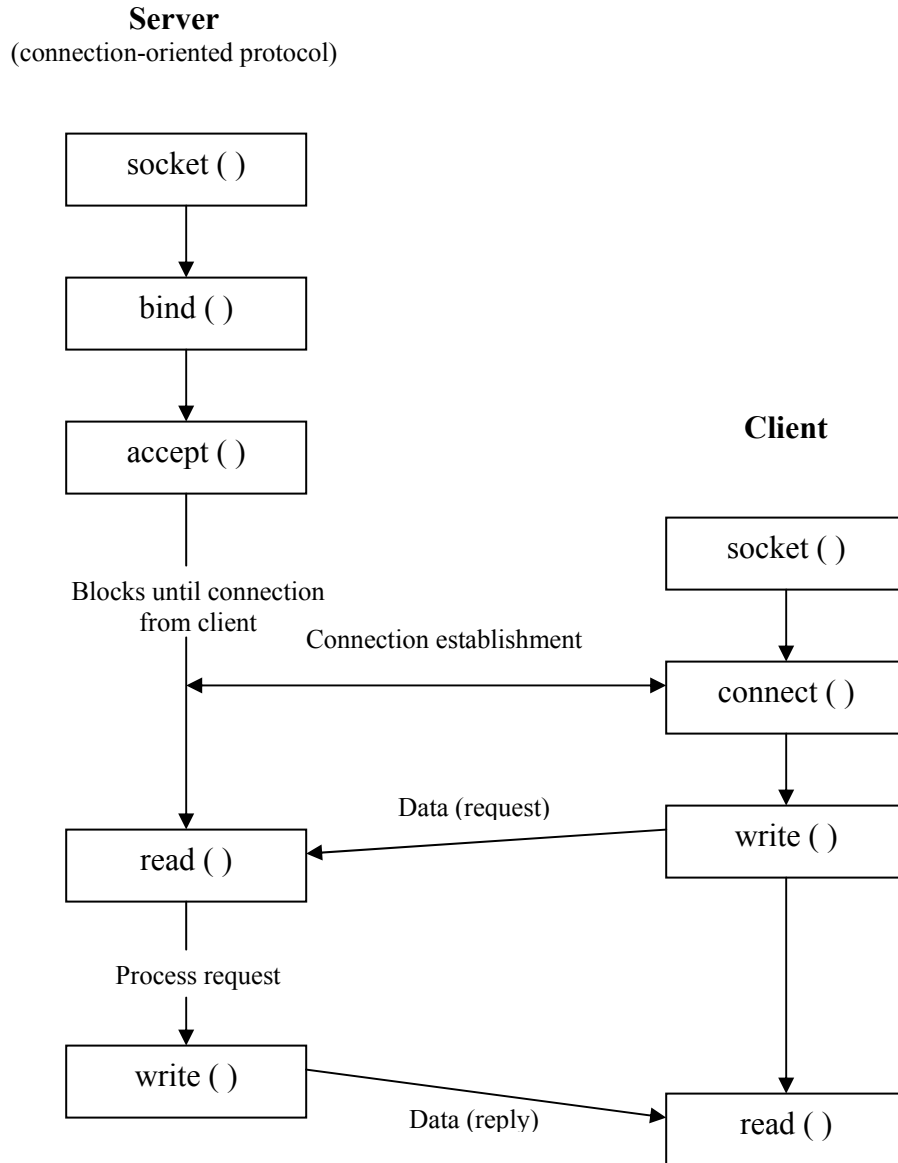


Figure 4.5: Socket System Calls for a Connection-oriented Protocol

4.3.1 Server Socket Calls

Socket Initialization – Constructor

socket_ct::socket_ct (u32 type)

The socket class constructor takes one argument as input. This argument specifies the type of the socket as a CLIENT socket or a SERVER socket. Based on the argument, the socket class instantiates itself and establishes a connection to the message queue that has been created by the APP_COMM_SERVER element installed on that node. In this step, no messages are exchanged, and only the message queue connection is established. The argument to specify the type is required to make the same socket class usable for both the client-side and server-side applications.

The default no-argument constructor returns with an error message asking the user to specify the type of the socket in the system call.

Bind

*socket_ct::bind (char *servAddr, u32 port)*

When the server process calls the *bind* system socket call with the IP address and port number to which it wishes to bind, the Socket Abstraction Layer captures this call with the wrapper function. It then sends a message to the APP_COMM_SERVER element in the Execution ARMOR or the Daemon on its node specifying the IP address and the port number. This message is sent over the message queue connection that was established during the socket initialization phase.

The APP_COMM_SERVER element extracts the IP address and port number from this message and sends a MSG_APP_REGISTER_SERVER message to the FTM. The APP_SERVER_DB element in the FTM adds a new entry for the server into the run-time hash table listing the existing servers in the Chameleon environment.

Accept

socket_ct::accept ()

After binding to a specific port on the host machine, the server process calls the *accept* system socket call. This call is blocking and waits for the client process to connect to the server by issuing a *connect* system call.

The Socket Abstraction Layer captures this system call and in response, opens a named pipe connection as a server. It then waits for the APP_COMM_SERVER element on its node to establish a connection with this named pipe, which it does upon receipt of a connect request from the APP_COMM_CLIENT element on the client node. Once connected, the APP_COMM_SERVER element sends the client-id to the server process over the named pipe. This named pipe is also used for any further communication between the client and the server as explained in the description of *read* and *write* socket calls.

4.3.2 Client Socket Calls

Socket Initialization – Constructor

socket_ct::socket_ct (u32 type)

The socket class constructor takes one argument as input. This argument specifies the type of the socket as a CLIENT socket. Based on the argument, the socket class instantiates itself and establishes connection to the message queue that has been created by the APP_COMM_CLIENT element installed on that node. The argument helps making a generic socket class, which can be used for both the client-side and server-side applications.

Connect

*socket_ct::connect (char *servAddr, u32 port)*

When the client process issues a *connect* socket system call, the Socket Abstraction Layer captures it using this wrapper routine in the *socket_ct* class. It sends a message to the APP_COMM_CLIENT element specifying the IP address and port number of

the server process to which the client wants to connect to. After sending the message, it opens a named pipe and waits for the APP_COMM_CLIENT element to complete the connection.

The APP_COMM_CLIENT element then sends a MSG_APP_SERVER_LOOKUP message to the APP_SERVER_DB element of the FTM, which returns the logical armor-id of the server (The armor-id of the Execution ARMOR associated with the server process). The APP_COMM_CLIENT element then establishes a connection with the client process over the open named pipe and forwards the armor-id over this connection. In the event that the requested server does not exist in the system, an armor-id (-1) is returned, and the connect call can exit using this as an error condition, as in the case of UNIX system call.

4.3.3 Common Socket Calls

Read

*socket_ct::read (char *msgBuffer, u32 nbytes)*

The *read* system call acts as a proxy for the TCP connection. When the user issues a *read* system call, the wrapper routine issues a *read* call on the named pipe it opened with the APP_COMM_CLIENT or APP_COMM_SERVER element during the connection-establishment phase. Thus a blocking read on the logical socket descriptor transforms into a blocking read on the named pipe connection.

Write

*socket_ct::write (char *msgBuffer, u32 nbytes)*

As with the *read* system call, when the user issues a call to the *write* system call, the wrapper routine issues a *write* call on its named pipe connection with the ARMOR. The application-level data for the peer entity is forwarded to the Chameleon environment. The thread listening on the named pipe for data encapsulates this user-level message into an ARMOR message and sends the encapsulated message to the destination ARMOR. At the destination, the application-level data packet is extracted

from the encapsulated message and sent to the peer entity over the named pipe connection. This data is then received by the application via the wrapper for the *read* system call.

Close

socket_ct::close ()

The *close* system call is responsible for doing the after-processing cleanup. When the user application calls the *close* system call, the wrapper routine captures it and closes its named pipe connection as well as the message queue connection with Execution ARMOR. In the case of the server, the Execution ARMOR sends an *unregister* message to the FTM. In response, the APP_SERVER_DB element in the FTM removes the entry for the server process from the runtime hash table.

4.4 End-to-End Message flow

This section gives a pictorial summary of the end-to end message flow from the launch of the server process to the teardown of the TCP connection between the client and the server. Figure 4.6 illustrates the end-to-end flow of messages during the connection establishment phase of the TCP connection. The accompanying description below gives details of the procedure. The messages are numbered according to the order in which they are issued in a typical scenario.

1. *Bind system call*: The server process binds itself to a particular IP address and port number on the system. This information is redirected to the APP_COMM_SERVER message queue. The thread listening on the message queue extracts the IP address and port number from the message and registers the server with the FTM.
2. *MSG_APP_REGISTER_SERVER*: The APP_COMM_SERVER element in the Daemon sends a message to the FTM to register the new server process with the IP address and port number it specified in the *bind* system call.

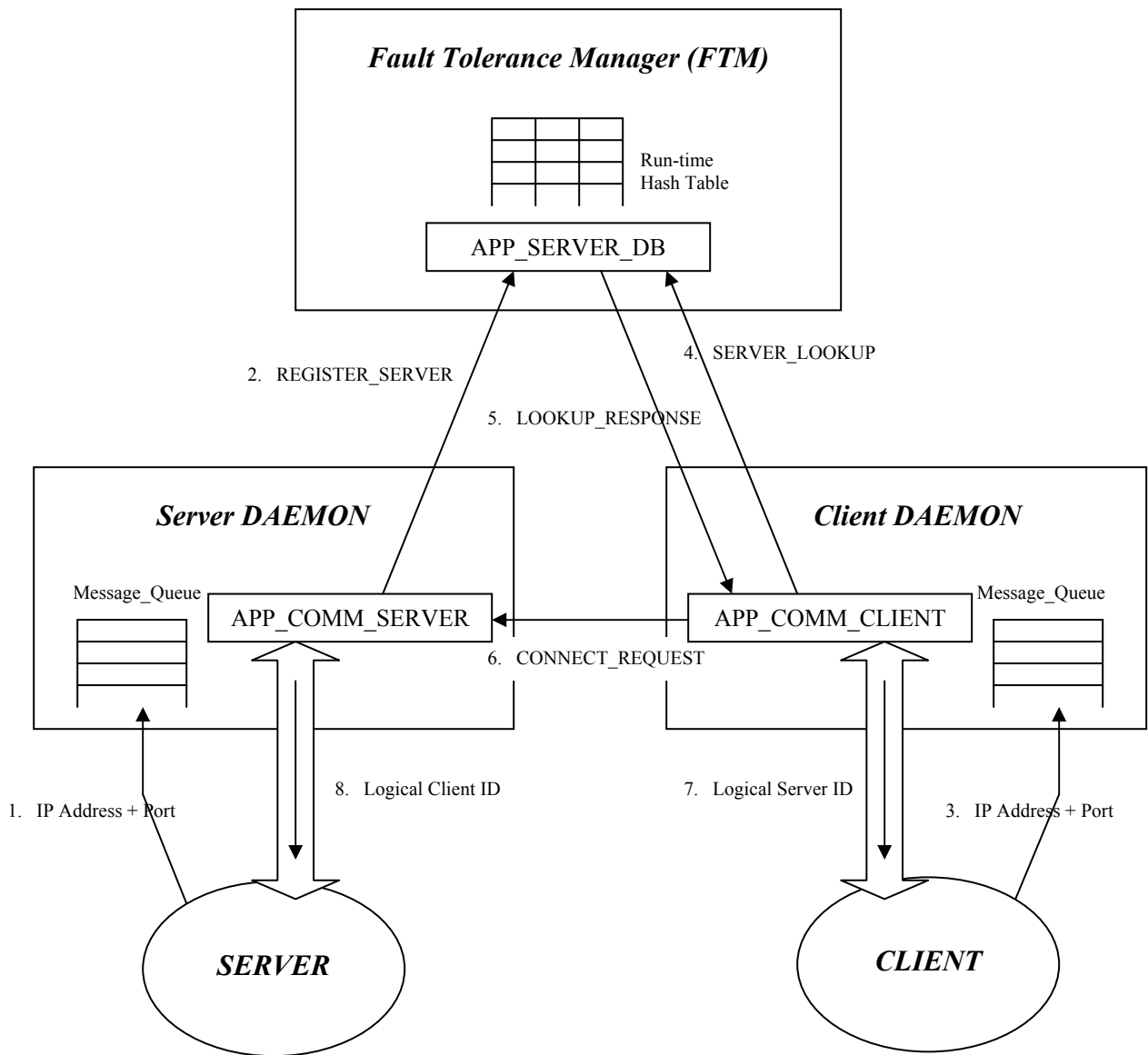


Figure 4.6: End-to-End Data Flow Diagram for Connection Establishment

3. *Connect system call*: The client process issues the *connect* system call with the IP address and port number of the server with which it wants to establish a connection. This request is redirected to the message queue of the APP_COMM_CLIENT element on that node.
4. *MSG_APP_LOOKUP_SERVER_INFO*: The APP_COMM_CLIENT element does a server lookup in response to the *connect* system call issued by the client process. It sends a message to the FTM to look up a server associated with the IP address and port number specified by the client.
5. *Server Lookup response*: The FTM responds back to the client Daemon with the armor-id of the Daemon associated with the server process. It returns -1 if no server is listed in its run-time table at the IP address and port number requested.
6. *MSG_APPCOMM_SERVER_CONNECT_REQUEST*: The client Daemon completes the named pipe connection with the client and sends a connection request message to the server Daemon with the armor-id received from the FTM.
7. *Logical Server ID*: The armor-id of the Daemon associated with the server is forwarded to the client process over the named pipe connection.
8. *Logical Client ID*: The server Daemon completes the named pipe connection with the server process when it receives the connection request from the client Daemon. It then sends the client's logical armor-id to the server process.

4.5 Error Detection and Recovery

The high-availability framework for client-server applications protects against various failure scenarios at different levels within the system hierarchy. The server process is often the most critical part of a client-server application and is required to be highly available. This means that failure of the server should be detected and the server reinstated without affecting the clients that have open connections with the server. After recovery, the server should be able to resume the ongoing transactions with the clients with whom it was connected before the crash, and this resumption needs to be transparent to the clients.

In addition to the server process, the ARMOR processes providing the fault tolerance services also need to be protected. The processes that have been added to provide high availability on behalf of the infrastructure should not become single points of failure. The most prominent of these new processes is the Daemon process that runs on each node participating in the Chameleon run-time environment. The elements that act as redirection proxies for the client and server applications are a part of the Daemons and thus need to be robust as well, in order for the proposed solution to provide the high availability.

The most challenging of all crash scenarios is to recover the system when the entire node running the server application and its associated Daemon crashes. In this case, the Chameleon environment detects the crash and migrates all the processes from the node that crashed to a new node. It is a challenging task to be able to recover the transactions between the server and the connected clients in such a failure scenario. The proposed framework can handle all such types of errors and can elegantly recover from the process crash in a manner that is fully transparent to the client processes connected with the server. The following subsections provide a detailed description of different failure scenarios and outline how our framework recovers from these errors.

4.5.1 Server Process Failure

The failure of the server process is detected by the Execution ARMOR (or Daemon) overlooking the server. When the server crash is detected, the ARMOR informs its manager of the failure and restarts the server application on the same node. The interface provided by the APP_COMM_SERVER element supports the reconnection of the server to the ARMOR. When the server application restarts, it reinitializes the TCP socket and re-issues the *bind* system call with the same IP address and port number as the first instantiation. When the APP_COMM_SERVER forwards a request to the manager ARMOR to re-register the server with the system on the same IP address and port number as before, the APP_SERVER_DB element recognizes this is as a request coming from a server in recovery phase. The element does a lookup in its run-time hash table and returns a list of the clients that were connected to the server before it crashed.

The APP_COMM_SERVER retrieves this list of clients and sends a message to the ARMORs overlooking each of these clients to replay the connection establishment request to the server. The APP_COMM_CLIENT elements subscribe to this message and resend the connection request message to the server. This message from the client-side ARMOR triggers the re-establishment of a connection between the server and its Execution ARMOR (or Daemon).

Once the named pipe connection between the server and its ARMOR is re-established, the data flow between the server and the client can resume. If the server application can store its state at regular intervals of time, the server process can resume its operation from the point at which it crashed, instead of restarting from scratch. This checkpointing feature for black-box applications is not supported in the current version; it is an exciting extension to our work, which is being looked into for future versions.

4.5.2 ARMOR Failure

The ARMOR failure scenario discussed here is the failure of the Execution ARMOR. This ARMOR overlooks the application process and is the gateway for all communication with the application. When the redirection proxy elements are made a part of the Execution ARMOR, it serves as the proxy for the remote end of the TCP connection. The failure of the Execution ARMOR is detected by the Daemon running on the node. The crash of the Execution ARMOR automatically crashes the application process connected to it through an open named pipe. Let us consider the case when the Execution ARMOR overlooking the server process encounters an error and crashes. This crash is detected by the Daemon using heartbeats and reported to the FTM. The FTM in turn re-instantiates the Execution ARMOR on the same node under the same Daemon. One point to be noted here is that the Execution ARMOR is restarted on the same node with the same ARMOR id it had before the crash. Thus, the logical id of the server process (which is essentially the armor-id of the Execution ARMOR) does not change as a result of the ARMOR failure. When the Execution ARMOR is up and running again, it restarts the server application. The server process recovers itself in the manner as described in the previous subsection. Thus the Execution ARMOR that guarantees the

availability of the server process, is in turn protected from errors by the Daemon, and thus it does not become a single point of failure in the system.

4.5.3 Daemon Failure

Daemons are installed on all nodes at the initialization of the Chameleon environment and the armor-id of the Daemon remains fixed throughout after that point of time. The procedure to recover a crashed Daemon when the Execution ARMOR behaves as a proxy is different from the recovery procedure when the Daemon is the proxy for the application. In the former case, the FTM migrates all the processes running on the node with the crashed Daemon (including the Execution ARMOR) to a new node with a Daemon already installed on it. Since the Execution ARMOR is restarted on the new node with the same id as before, the logical armor id of the application does not change.

However, in the latter case, when the Daemon behaving as the redirection proxy crashes, the FTM migrates the application process to a new daemon, which has a different armor-id from the one it had before the crash. Since the id of the daemon is the logical armor-id of the application in this case, the point of reference for the application process changes, and this change needs to be reflected throughout the system.

Consider the case in which the daemon associated with the server process crashes. After recovery, the logical armor-id of the server changes to the armor-id of the new daemon overlooking its operation. Thus the clients connected to the server before the crash need to be informed of the change, so that they can start transmitting packets to the new destination armor-id. When the server application is restarted by the daemon and it executes the *bind* system call with the same IP address and port as before, the APP_SERVER_DB element sends the list of old clients to the APP_COMM_SERVER element. As explained earlier, the APP_COMM_SERVER element in turn requests the clients to replay the connection request to trigger the completion of the connection with the restarted server. As a part of the processing of this request, the client-side daemons update their entry for the remote server's logical armor-id, so that they can resume communication with the new server.

4.5.4 Node Failure

Node failure is handled in a very similar manner to daemon failure. The failure of a node is detected by the absence of a response from the daemon on that node. Thus once the FTM establishes the failure of the node, recovery is the same as the case of daemon failure. All processes running on the failed node are migrated to a new node with a fresh daemon.

4.6 *Demo Application: Audio Streaming Server*

The main characteristics desired of an application to demonstrate the applicability of the proposed scheme are as follows:

TCP based client-server architecture: A real life application based on the client-server architecture is needed. The client and server should communicate with each other over TCP sockets using the UNIX system calls.

Multithreaded application: It is worthwhile to demonstrate using an application based on a multithreaded model of execution, since most real-life applications would be of this nature.

High availability requirements: Most client server applications require the server to be highly available (meaning the server should be up all the time). Since in real life, applications do suffer from crashes, the clients should require that the server transparently recover from such crashes.

High bandwidth requirements: Since the new scheme primarily affects the latency of message transfer between the client and server due to redirections, the application should have reasonably high bandwidth requirements.

The application we choose for the purpose of demonstrating the feasibility of the idea is a lightweight audio streaming client-server application. Multiple clients can connect to the

multithreaded server. The clients want the server to recover from failures transparently, and the application has reasonably high bandwidth requirements.

In the demonstration application, client and server processes communicate with each other using the UNIX based TCP socket system call interface. The server binds itself to a well-known IP address of the host machine at a port number that the client knows. It then waits for clients to connect to it from remote hosts by issuing the *accept* system call. The client process knows the IP address and port number of the server and issues a *connect* system call to complete the connection with the server. The server, upon receiving the *connect* request, spawns a new thread to handle the request from this client. A separate thread is created for each connection between client and server. This thread then streams a prerecorded audio file in the raw audio format to the client, which plays the file as it receives it. The application requires a moderate bandwidth for communication between the client and server: approximately 8 KBps for the audio quality to be sufficiently good for playback at the client end. Performance measurements with this application are provided in Chapter 6.

5. CHAMELEON FOR WIRELESS APPLICATIONS

5.1 *Introduction*

TCP has been finely tuned over the years for traditional networks comprising of wireline links and stationary hosts. It assumes congestion in the network to be the primary cause of packet losses and unusual delays. TCP performs well over such networks by suitably adapting to the end-to-end packet delays and congestion losses. The TCP sender uses the cumulative acknowledgement scheme to determine which packets have reached the receiver, and it provides for reliable communication by retransmitting the lost packets. The sender identifies the loss of a packet either by the arrival of several duplicate acknowledgements or by the absence of an acknowledgement for the packet within a timeout interval. TCP reacts to packet losses by initiating the congestion control and avoidance mechanisms such as slow start, and exponentially backing off its retransmission timer according to Karn's Algorithm. These measures reduce the load on the intermediate links, thereby controlling the congestion in the network.

Unfortunately, when packets are lost in networks for reasons other than congestion, these measures result in unnecessary reduction in end-to-end throughput and performance. Communication over wireless links is often characterized by sporadic high bit-error rates and intermittent connectivity due to weak signal strength. TCP performance in such networks suffers from significant throughput degradation and very high interactive delays.

Introduction of mobility and wireless links in the network architecture introduce many challenges to the design of applications, and network applications designed for wireline topologies do not normally work well when used in wireless networks. TCP, which deals with the vagaries of network communication like lost packets, out-of-order packets, and timeouts very efficiently over fast and reliable wired networks, is not suitable to provide a robust connectivity for the mobile and wireless communications. It does not effectively handle such wireless networking problems as devices moving out of range or across network boundaries, disconnects due to limited battery life, or other circumstances in which the device is

temporarily not in contact with the network. Some of the problems with using TCP in a wireless environment are as follows:

- The TCP/IP retransmission policy uses up a significant amount of network bandwidth. Recovery from lost data may require the retransmission of all outstanding data, even when the client has already received most of the transmission.
- When the mobile device (laptop) goes out of range, the virtual circuit established by TCP between the two end points is dropped, and applications then fail to recover when the device returns back into the covered range. When the device reattaches to the network, the user must log in again, restart all the applications, re-enter lost data, and essentially recreate its earlier state. This wastes time and bandwidth.
- As mentioned earlier, TCP assumes that a lost packet is the result of network congestion. In a wireless environment, packets are more often lost due to noise on the wireless link or the device being out of range. When a packet is sent, and the sender does not receive an acknowledgement, TCP assumes the failure was due to congestion and applies the exponential back-off algorithm. This degrades performance and frustrates users.

The current work tries to overcome the disadvantages of using TCP over wireless networks by proposing a scheme that can transparently be used by all the existing applications to improve connectivity over wireless networks.

5.2 *Robust connectivity over Wireless Networks*

As described in the previous section, the mobility made possible by wireless networks introduces challenges that are not an issue in a wireline environment and hence cause problems for applications that have not been designed with these in mind. In particular, network applications that rely on an always-on connection do not function properly on wireless networks. In many cases, applications crash when a mobile user encounters interference or moves outside of the wireless network coverage. Most client-server applications based on TCP are stateful, requiring a communication link with the network

server that is always open. If the link between the mobile device and the server is broken or the wireless signal is weak – a frequent occurrence in wireless networks – the application loses the information it needs to run correctly.

One way to resolve the problem is to rewrite or replace the applications with the ones that are specifically designed to handle network disruptions. But to take this approach with every client-server application to be used over a wireless network would be very expensive. It would also require an in-depth knowledge of the various problems of TCP over wireless by the application programmer, and providing for this feature might be more complex than the entire application itself.

This thesis proposes a new scheme to handle such intermittent loss of connectivity and implements a solution that offers persistent connections to all client-server applications over wireless networks. The scheme keeps the existing network connection alive at both ends of the connection while the physical connection might be down. Thus, the actual network session is maintained as the mobile user moves in and out of the wireless coverage.

5.3 *ARMOR-based solution to TCP over Wireless: Proxy ARMOR*

The scheme is a variant of the standard split connection approach with two split points instead of one. Each of the two split points acts as a proxy, which we call the *Proxy ARMOR*. A typical scenario in a client-server application is when the server runs on a host on the wireline LAN and the clients are on mobile devices. The server-side proxy runs on the wireline network at the gateway host, and the client-side proxy runs on the mobile host itself. Each of the proxies is a lightweight process whose primary purpose is to listen for packets on one end of a socket connection and transfer those packets on to another socket connection. Figure 5.1 illustrates the approach.

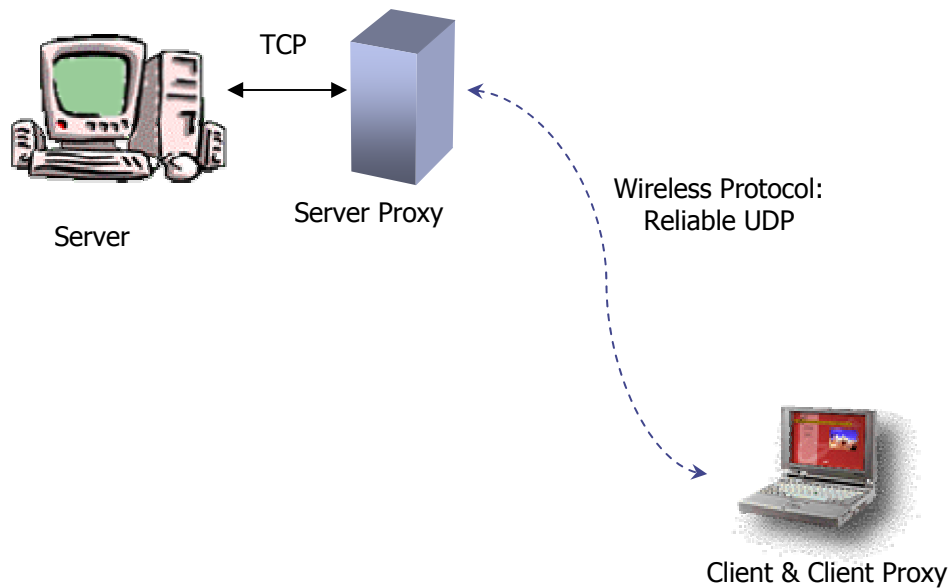


Figure 5.1: Dual Proxy-based Solution to Provide Persistent Connections on Wireless Links

The TCP packets from the end host are converted to some special wireless protocol, which is designed especially for the wireless link. This wireless protocol needs to be connectionless and reliable. We use a reliable version of UDP as the wireless protocol – UDP with explicit acknowledgements for each packet. These packets are transferred via one of the wireless links and then converted back to TCP at the other split point, which is executing in the mobile device. The server thus communicates with the gateway instead of the ultimate client, and the client receives the data from the proxy located in the wireless device, thinking that it is receiving the data directly from the remote server.

Our proxy-based solution to support client-server applications over Chameleon infrastructure uses essentially the same architectural concepts as the above-mentioned scheme to provide the complete framework for the double-proxy solution for TCP over wireless. Since we can control the communication protocol used between the ARMORs, we implement an R-UDP protocol for this communication.

Some of the advantages of the double-split connection approach are as follows:

- Delays that occur due to fading over the wireless link can be hidden from the sender.
- The introduction and the design of the wireless protocol does not depend on possible future changes to TCP.
- The approach is completely transparent and does not involve changing either end of the application.

The issue of breaking the end-to-end semantics of TCP is not significant, as the breakage already happens in wireless networks for typical mobile devices because of masquerading and tunneling at the gateway firewalls.

5.4 Supporting TCP and UDP in Chameleon

The problem of providing robust connectivity to client-server applications over wireless networks is solved in our scheme by providing simultaneous support for multiple communication protocols between ARMORs in Chameleon. The ARMORs that communicate with other ARMORs on the wireline network can use the TCP communication channel, and those that communicate with the ARMORs on a wireless node can use a different communication protocol that is more suited to the wireless channel. We demonstrate the flexibility and the ease of adding new protocols to the architecture by providing simultaneous support for both TCP and UDP based protocols for inter-ARMOR communication.

Inter-Daemon TCP Communication

Each daemon in the Chameleon network contains a mandatory element to assist it in the task of communicating with remote daemons in the environment. This element is called the TCP/IP connection element (*daemon_tcp_mgmt_t*). This is the element that manages TCP/IP connections to other daemons in the Chameleon environment. The central data structure for this TCP connection element is a table that maps daemon ids to IP addresses and port numbers. The element updates the table on demand with help from the FTM daemon. The FTM daemon is the first daemon installed in the Chameleon environment and is the manager of all the other daemons in the network.

When a daemon needs to route an outgoing message to a remote daemon, it attempts to locate the destination daemon's id in its connection table. If it cannot find the remote daemon in its connection table, it requests the remote daemon's IP address and port number from its manager (FTM daemon). The FTM daemon maintains a list of all daemons in the network, and thus each daemon has to register itself with the FTM daemon as a part of the installation procedure. The registration consists of sending the daemon's IP address and the port number on which the daemon accepts new connections.

The TCP/IP protocol requires a separate connection to be established for communicating with each remote daemon. Each daemon establishes a connection with the remote daemon the first time it needs to communicate with it. This connection is persistent and remains open once it is established. This is essential to keep the communication latency within bounds, or else a new connection would need to be established each time a packet has to be sent to the remote daemon. The connection establishment would also require a lookup into the runtime connection table. Thus daemon ARMORs keep connections open once they are established until the time the current configuration is in place. This is the primary reason for the need for improvement in the underlying inter-ARMOR communication of Chameleon. It makes ARMORs more suitable for wireless channels in which the connections can drop and the inter-daemon connections would be affected by the limitations of TCP due to mobility of the client, as described earlier.

The main responsibilities of the TCP connection element are as follows:

- Maintain a table of TCP/IP parameters needed to establish a connection with remote daemons, which is updated from the FTM daemon on an as-needed basis,
- Maintain a small cache of open connections to remote daemons,
- Routing of messages to remote daemons,
- Monitor each open connection for incoming messages, and
- Accept new connections from remote daemons.

Support for stateless UDP communication

The new element added to the Chameleon system to provide support for stateless UDP communication amongst daemons is the UDP connection element (*daemon_udp_mgmt_t*). This element is very similar to the TCP connection element in terms of the functionalities it provides to ARMORs. However, instead of using the connection-oriented protocol like TCP, which mandates the maintenance of open connections for entire duration of the system, this element allows the ARMORs to communicate with each other by using a connectionless protocol, UDP. When used over wireless hops, it provides for all the advantages of the dual-proxy solution for wireless TCP.

The messages subscribed to by the UDP connection element are the same as those subscribed to by the TCP connection element. Such a design helps to replace the underlying communication protocol by simply replacing the *daemon_tcp_mgmt_t* element in the appropriate daemons by the *daemon_udp_mgmt_t* element. Since both elements respond to the same messages in different ways depending upon the corresponding protocol specifications, a daemon cannot contain both the elements as active entities. The current implementation does not allow a daemon to communicate using both TCP and UDP protocols to other daemons in the distributed environment. However, the entire Chameleon run-time environment may comprise of nodes with daemons of both types. A subset of nodes could be using TCP communication, while the remaining might be using UDP to communicate with each other.

The UDP connection element uses the *socket_udp_t* class instead of the *socket_tcp_t* class for instantiating the socket. The UDP socket does not provide for primitives like *accept* and *listen*. However, it does provide for *bind*. When a UDP socket is bound to a particular IP address and port number, it does not establish any physical connection to the remote entity, but it does store the information about the other end of the connection. So, once bound to a particular address, all data sent on that socket would arrive at the same destination using the underlying UDP protocol, eliminating the persistent connection established in case of TCP. If the mobile user goes into a weak signal area, no connections are reset and the application can resume normally once the client comes back into the range of the wireless access point.

However, packets may get lost in the time frame when the client is out-of-range. This might or might not be a problem, depending on the type of application targeted. If the application is a real-time streaming application where a late-arriving packet is useless, then the loss of packets is acceptable. If the arrival of the packet is essential to the proper functioning of the application, this scheme needs to be modified. For this reason, we introduce the concept of Reliable UDP in the next subsection – UDP with explicit acknowledgements for the packets.

5.5 *Reliable UDP: Introducing Acknowledgements*

Chameleon provides an elegant way to add explicit acknowledgements to packets sent to remote ARMORs. If an ARMOR wants an acknowledgement for a packet it sends to a remote ARMOR, it adds a new bundle on the message stack after the actual message. The new bundle contains the message *MSG_BUNDLE_ACK*. It also turns on the *MSG_EXPECT_ACK* flag in the message bundle and addresses the bundle to itself. This starts a local timer in the ARMOR waiting for the ACK to arrive. If the timer goes off before the acknowledgement arrives, the whole packet is sent again.

When the ARMOR at the destination end receives the bundle and is done processing the message from the originating ARMOR, it encounters the *MSG_BUNDLE_ACK* message. In response to the message, it sends back an ACK to the sender ARMOR. The arrival of this acknowledgement is sufficient to determine that the original message was received and processed by the remote ARMOR. If we add this message transmission scheme, in which each message pushes a self-addressed acknowledgement bundle after the main message, we get a form of UDP that is reliable and guarantees the arrival of the message on the remote end of the connection.

6. PERFORMANCE MEASUREMENTS

6.1 Experimental Setup

This section gives an overview of the experimental test-bed used to measure the performance of the proposed high-availability architecture for client-server applications. Two experimental setups were used to evaluate the performance of the system on wireline and wireless networks respectively. The following subsections describe these testbeds in detail.

6.1.1 Wireline Network Testbed

The experimental setup for the wireline performance measurements uses the CRHC (Center for Reliable and High Performance Computing) network of SUN workstations running the Solaris 7.0 operating system. These nodes are connected by 100 Mbps switched Ethernet. Figure 6.1 illustrates a configuration of the environment with the client-server application executing on the testbed.

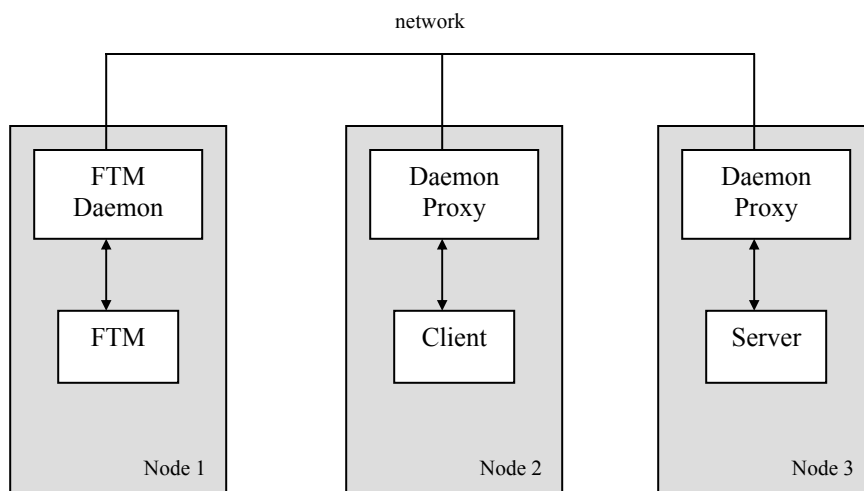


Figure 6.1: Wireline Testbed for ARMOR-Protected Client-Server Applications

Before executing the client and server applications, the FTM (Fault Tolerance Manager) is launched on one of the nodes. Following this, a Daemon proxy is installed on each of the nodes in the testbed. The FTM-daemon has an APP_SERVER_DB element, and the proxy

daemons include the APP_COMM_SERVER and APP_COMM_CLIENT elements. The applications are then started on the nodes with the proxy daemons installed. A fourth node (not shown in Figure 6.1) is used during recovery from a daemon or node crash. The FTM periodically exchanges heartbeat messages with each daemon (every 1 second) to detect node crashes and hangs. If the FTM does not receive a response by the next heartbeat round, it assumes that the node has failed. Upon detection of such a failure, the FTM migrates the processes that were running on the crashed node to the new node. In addition, the FTM maintains a run-time list of the all server application processes installed in the network. The client-side proxy does a lookup into this run-time table when it wants to connect to the server.

6.1.2 Wireless Network Testbed

The applicability and the robustness of the proposed framework over a wireless network is demonstrated using a Linux testbed. The experimental setup uses two Intel Pentium based machines running the Linux 2.4.7 operating system connected by a 100 Mbps switched Ethernet. A wireless access point is attached to this local network to allow communication between wireless and wireline hosts. The FTM is launched on one of the wired nodes during startup, followed by the installation of a proxy daemon on each of the nodes participating in the network. One of these daemons instantiates the server process, and the client process is started on the wireless node. Figure 6.2 illustrates the experimental setup.

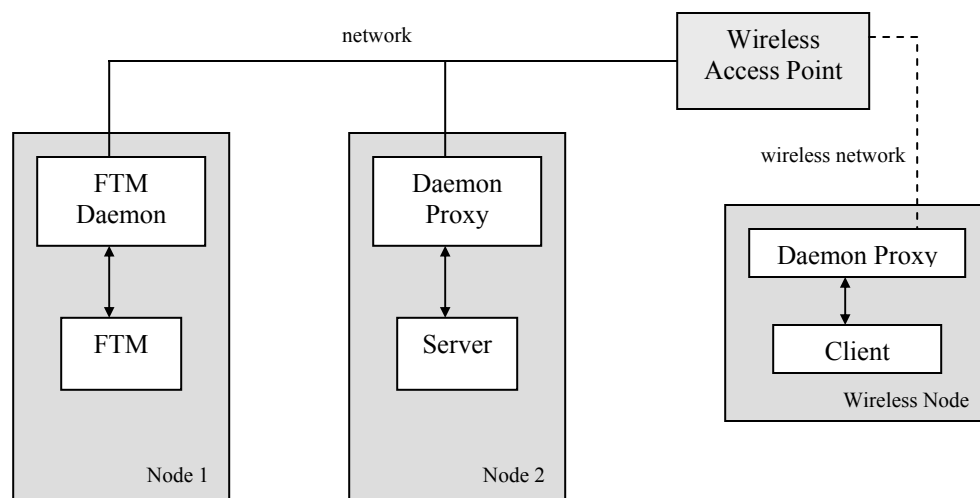


Figure 6.2: Wireless Testbed for ARMOR-protected client-server applications

Applications

Two applications are used for the performance measurements:

1. The audio streaming server [Section 4.6] is used to demonstrate the feasibility of the proposed framework for communication-intensive TCP-based applications. The server transmits live audio packets to the client. The client application feeds each packet to the output device as soon as it is received, i.e the client does not buffer data at its end. Thus, a delay in the arrival of packets at the client end would be noticed in the form of poor playback quality.
2. For the purpose of measuring the latencies of socket calls and the throughput achievable with the new framework, an interactive client-server application is used. The server process sends a data packet to the client, and the client responds with a data packet of same length. This gives us a good estimate of the RTT (Round Trip Time) for a packet to travel between the server and client processes.

6.2 Performance Evaluation

Performance is evaluated by measuring the impact of the ARMOR-based framework on socket call latencies. The results provided in this section are from the experiments done on the wireline network testbed. The results obtained from experiments on the wireless testbed were very similar, and have therefore not been presented here. The following latencies are measured in the experiments:

1. *Initialization* – One-time overhead incurred when initializing the sockets.
2. *Socket bind* – Overhead incurred by the server process to register itself with the FTM.
3. *Connection establishment* – Overhead incurred by the client to connect to the server.
4. *RTT* – Round-trip latency of a packet exchanged between client and server.

These performance results are then compared with the equivalent times for the baseline dual-proxy implementation using system-level sockets. The baseline proxy-based implementation of the interactive client-server application uses server and client proxies coded in C++. Each proxy is a lightweight process that transfers data from one connection to another connection

[Section 5.3]. This comparison gives us a measure of the overhead incurred as a result of using the ARMOR-based framework.

6.2.1 Throughput

The throughput is measured in terms of the RTT of a packet from the server to the client and back to the server. The packet size is varied from 32 bytes to 2048 bytes. The comparison of RTT for direct connection with the RTT for proxy-based implementation gives us an estimate of the overhead in the communication latency as a result of using the dual-proxy solution for robust connectivity. Measuring the RTT for the ARMOR-based proxy framework gives us the overhead of providing reliability to the dual-proxy solution. These numbers are provided in Table 6.1 below.

Packet Size	Direct Connection (usecs)	Dual-Proxy (usecs)	ARMOR protected (usecs)
32 bytes	816	2218	33526
64 bytes	967	2601	33314
128 bytes	1075	2934	33549
512 bytes	1756	3877	32716
1024 bytes	2708	5142	35660
2048 bytes	4653	6801	39889

Table 6.1: Round-trip Latency

Round-trip TCP latency for data transfer from process to process at varying packet sizes. Times shown are average over 1000 packet exchanges.

As can be seen, the overhead for using ARMORs to provide reliability to the proxy-based scheme is around 30ms for a 1 KB packet. The RTT remains almost the same if the packet size is varied, as expected. This is because the current version of the ARMOR messaging system uses fixed size messages for all inter-ARMOR communication. The size of a typical message is around 8 KB, of which 1 KB can be used for the actual data. The additional space is used for book-keeping purposes and to transfer payload information (a data area which can

be read from or written to by ARMOR elements) between ARMORs. Thus, no matter what the application frame size is, the ARMOR messaging system introduces a significant overhead. This overhead mostly goes unnoticed in the case of stand-alone applications, as these applications do not use the ARMOR messaging system to communicate with other application processes. However, in our case, the proxy Daemons communicate with each other using the fixed size messages, hence incurring a relatively high overhead each time the client and server exchange data. This is because the ARMORs are now an integral part of the application, instead of being entities that just monitor the application.

We confirmed this explanation for the high overhead by sending a packet of 8 KB from the server to the client using the dual-proxy setup. Thus, if we eliminate the overhead due to the high packet size used by ARMORs, the true overhead introduced by Chameleon processes to protect the application can be measured as the difference between the RTT of a 1 KB data packet in the ARMOR-based setup [35.6 ms] and the RTT of an 8 KB packet in the dual-proxy setup [18.9 ms]. This overhead is approximately 80%.

Besides reducing the overhead by compressing the ARMOR messages, we propose an extension to the framework. This extension would maintain the information about all the connections established between ARMOR-protected applications in the same way as the current framework, but after the connection is established, the infrastructure would limit its responsibility to overseeing the application processes, and would establish a direct socket connection between the client and the server processes for communication. This improvement would drastically reduce the overhead incurred during the transfer of each message. We expect the packet RTT to become essentially the same as the direct-connection setup. At the same time, ARMORs would be able to detect the failure of either end of the connection and re-establish the connections using the information that was stored in the run-time table during connection establishment. This extension is described in further detail in Chapter 7.

6.2.2 Initialization

The time to make system calls to `socket_ct::init` and `socket_ct::bind` have been measured and compared for the dual-proxy implementation and the ARMOR-based implementation. The call to `bind` registers the server process with the FTM, which adds an entry for the server into its run-time table. This run-time table contains a mapping for the IP address and port number of the server to the logical armor-id for the server (ARMOR id of the server proxy daemon). The results are shown in Table 6.2 below.

Type of socket call	Dual-Proxy (usecs)	Chameleon protected (usecs)
init	368	424
bind	178	7545

Table 6.2: Socket Initialization and Bind Latencies
Average time to make system calls to `init` and `bind` over 100 calls.

The difference in the initialization time of the socket is negligibly small. Moreover, these overheads are incurred only during the server startup phase, when it declares its presence in the ARMOR-based environment. Similarly, when the server process executes the `bind` system call, a message is sent to the FTM that is running on a separate node. The FTM responds with success/failure notification, after which the function call returns. This overhead was expected, but since it is a one-time expense, it can be accounted for. The client process is started only after the server returns from the `bind` call.

6.2.3 Connection

We measure the time it takes for the client to connect to the server in case of the dual-proxy implementation and compare it with the time in case of ARMOR-based implementation. We time 100 application calls to `socket_ct::connect` and `socket_ct::close` functions. Results are provided in Table 6.3.

Type of socket call	Dual-Proxy (usecs)	Chameleon protected (usecs)
connect	10560	31339
close	275	273

Table 6.3: TCP Connection Time

Average time to make system calls to connect and close over 100 calls.

The connection time for the ARMOR-based implementation is about 3 times that of the dual-proxy implementation. This extra time was expected, as the ARMOR-protected client does a lookup into the run-time table of servers maintained by the FTM on another node to obtain the logical ARMOR id of the server. This is roughly the same as the RTT of a packet between the server and client processes. We expect this number to go down with the next version of Chameleon when the overhead due to large inter-ARMOR communication packets is reduced. As with the *bind* system call, this overhead is a one-time overhead and is incurred by the client when it wants to connect to the server process.

6.2.4 Recovery Time

Recovery time is the interval between the time at which a failure is detected and the time at which the target process restarts. To simulate the failure of processes due to errors, we send the SIGKILL signal to the target process. The experiments do not attempt to analyze the cause of the errors; the aim is to be able to generate a process crash in a controlled and repeatable manner.

The high availability framework protects against various kinds of failures. In our experiments, we study the consequences of the failure of the server process, the daemon, and the node running the server and its proxy. The detection of failure of the server process is almost instantaneous. However, the detection of failure of the daemon process and the node depends on when the crash takes place. Chameleon uses heartbeats to detect such crashes, and the heartbeat interval can be customized using a run-time environment variable. We run all

experiments with the heartbeat interval set to 1 second. Thus, depending on when the crash occurs in the heartbeat interval, the detection can take anywhere from 0 to 1 second.

The Chameleon environment protects both the client and server process from failures and recovers them from such crashes equally well. In this section, we present measurements for server-side crashes. Numbers for client crashes are almost identical. We measure the time it takes to recover the server process (on a different node in case of daemon and node crash) and re-establish the lost TCP connections with the clients. This gives us an estimate of the recovery time for the various failure scenarios. Results for 20 runs per target are summarized in Table 6.4.

Failed entity	Recovery time (milliseconds)
Server	10
Daemon	135
Node	145

Table 6.4: Recovery Time for Various Failure Scenarios
Average time to recover the crashed processes and re-establish the server-client TCP connections. Times shown are average over 20 failure tests.

The recovery from the server crash is almost instantaneous. We simulate a node crash by rebooting the node that hosts the server process. It takes much less than a second to recover from such crashes. These results are very encouraging and illustrate the usefulness of the framework. Without the use of ARMORs, responsibility for detecting and restarting the server rests with the user. This would require the user to manually kill all the client processes, restart the server, and then restart the clients, which might even take up to a minute. This might be even more difficult if the clients are at different geographical locations and would thus force the clients to do the detection and recovery themselves. In contrast, with the ARMOR-based framework, the error detection and recovery of the server occurs automatically and is completely transparent to the client processes, which need not be restarted. This solution would be even more elegant if the server process could checkpoint its state to stable storage at regular intervals and then recover from such a checkpoint.

6.3 *Execution ARMOR Proxy versus Daemon-based Proxy*

As explained in the earlier chapters on the design of the high-availability framework, we can use two different approaches to build the proxies in Chameleon. The proxy elements, `APP_COMM_CLIENT` and `APP_COMM_SERVER`, can be made a part of either the Execution ARMOR, or they can be added to the Daemons on the nodes running the server and client processes. In the former case, the Execution ARMOR is the primary entity responsible for overseeing the operation of the application process and for recovering it when errors are detected. The proper functioning of the Execution ARMOR is in turn guaranteed by the Daemon, which oversees the ARMOR. This is the standard mode in which Chameleon has been used to protect black-box applications. However, as pointed out in the previous section, unlike other stand-alone applications, the client and server applications in our framework communicate with each other through the ARMORs and the daemons. Thus the use of Execution ARMORs introduces an extra level of indirection in the messaging system, and increases the communication overhead tremendously.

An ARMOR is a collection of elements that provide for all the desired functionality. Thus if we merge the elements of an Execution ARMOR that are responsible for overseeing an application process with the elements of the Daemon, we get a modified daemon. This modified daemon can then do the job of overseeing the application in addition to allowing it to communicate with the remote entity. We can therefore eliminate an unnecessary level of indirection from the architecture, which would result in a significant performance boost. Figure 6.3 illustrates the difference.

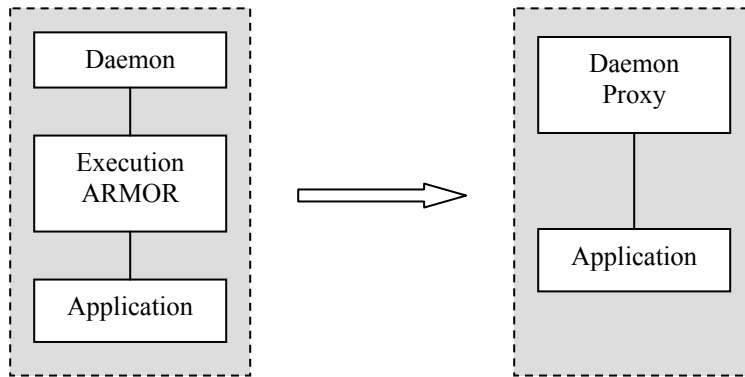


Figure 6.3: Execution ARMOR Proxy versus Daemon Proxy

To give an idea of the performance improvement achieved as a result of merging the task of overseeing the application with the other tasks of the Daemon and by making the Daemon ID as the logical armor-id of the application process, we re-run all the experiments with the Execution ARMOR included. The results are shown in Table 6.5.

Experiment	Entity overseeing the application process	
	Execution ARMOR (usecs)	Daemon (usecs)
Initialization Time	467	424
Bind Time	7621	7545
Connection Time	75659	31339
RTT (512 bytes packet)	514885	32716

Table 6.5: Performance Comparison of Execution ARMOR versus Daemon
The initialization, bind, and connection times are average over 100 calls to the respective functions. RTT is average over 1000 packet (512 bytes) exchanges.

Table 6.6 shows the elapsed times for recovery from various kinds of failures over multiple runs of the experiment. A new failure scenario for the crash of the Execution ARMOR has been added.

Failed entity	Recovery time (milliseconds)
Server	25
Execution ARMOR	346
Daemon	1513
Node	1546

Table 6.6: Recovery Time with Execution ARMOR

Average time to recover the crashed processes and re-establish the server-client TCP connections. Times shown are average over 20 failure tests.

As can be seen, the throughput achievable from the framework is greatly reduced when we include the Execution ARMOR in the framework. The time to recover a crashed ARMOR is of the same order as the recovery of a daemon in the case in which we do not use the Execution ARMOR. However, it takes much longer to recover a crashed daemon in this case. Upon detection of a daemon/node crash, the FTM migrates all the processes including the Execution ARMOR to a new node, which is an expensive operation in terms of time. Recovering from such a crash takes approximately 1.5 seconds.

Analyzing the performance measurements, we conclude that the use of Execution ARMORs unnecessarily introduces an extra level of indirection in the client-server communication and can be done away with. We therefore strongly recommend using the proxies as daemons without the Execution ARMORs.

7. CONCLUSIONS AND FUTURE WORK

This thesis presented the design and performance of an end-to-end high availability infrastructure for client-server applications. The architecture provides a framework for improving the availability of services in a transparent manner, not requiring any changes to either end of the connection, or to the operating system. The need for such support is apparent from the large class of applications being built that require the service to be up and running all the time. Our solution is generic and can help make any existing server system fault tolerant. Moreover, the architecture can be used directly to protect services in a wireless environment, where the connection between the client and server may often drop due to deterioration in the strength of the physical signal.

The salient features of our framework are:

- The architecture provides an *end-to-end high availability framework* for client-server applications. Not only are the client and server applications protected from crashes, also protected are the connections established between them. The architecture is self-contained in terms of error detection and recovery and does not rely on external entities for detecting the fault.
- The framework provides high availability in a *transparent* manner. The application does not need to be modified or recompiled, and no changes need to be made to the operating system kernel. The recovery of the server process from a crash is completely hidden from the client.
- The architecture is *wireless-friendly* and can be used to protect client-server applications in an environment in which the wireless signal strength is unpredictable. The connections between client and server are automatically resumed when the wireless host comes back into the range of the physical signal.

The architecture is proxy-based and is implemented in user-level code. We use an audio streaming server application to demonstrate the feasibility of the solution and to measure the performance overhead introduced as a result. This application serves as a model to provide high availability to other client-server applications by using this framework.

Future Work

The performance results indicate a message transfer overhead of about 80% as compared to the unprotected dual-proxy implementation. This overhead is incurred each time a message needs to be transferred between the client and the server. In order to eliminate this overhead introduced as a result of redirection of messages, we propose a modification to the current design of the high availability framework.

In the current design, the client and server exchange all messages via the Daemon on their respective nodes. These Daemons act as proxies and introduce the unwanted delay in the message transfer. During the connection establishment phase, the FTM stores the information about the clients that are connected to the server. This information is then used during the recovery phase to re-establish the lost connections. We can reduce the message transfer overhead considerably if the client and server are allowed to communicate directly after the connection establishment phase, without involving the Daemons. Once the FTM has the necessary information about client-server connections, direct socket communication can be established between the client and server. Daemons limit their responsibility to monitoring the application processes, once the connections are established. Since the messages would no longer need to be redirected, the message-transfer latency would be essentially the same as that of a direct socket connection. In the event of a crash, the FTM uses the information stored during connection establishment to re-establish the connections between client and server.

The interface of the Socket Abstraction Layer is derived from a standard C++ class interface, but this does not limit the class of applications that can benefit from the architecture. The Socket Abstraction Layer can be modified to conform to any socket class interface used by the application, thus allowing all applications to achieve high availability using the ARMOR-based framework.

Finally, the ARMOR messaging system needs to be optimized to reduce the message size overhead. The current implementation allocates a fixed size contiguous chunk of memory for each message and then packs that memory into an ARMOR message. This scheme needs to be modified to use the memory more efficiently.

LIST OF REFERENCES

- [1] Z. Kalbarczyk, R.K. Iyer, S. Bagchi, K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Transactions on Parallel and Distributed Systems*, June 1999.
- [2] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, R.K. Iyer, "Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment," *IEEE Transactions on Knowledge and Data Engineering*, 2000.
- [3] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, R.K. Iyer, "Incorporating Reconfigurability, Error Detection and Recovery into Chameleon ARMOR Architecture," *Technical Report CRHC-98-13*, UIUC, 1998.
- [4] H. Balakrishnan, V.N. Padmanabhan, S. Seshan, R.H. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links," *In Proceedings of ACM SIGCOMM*, 1996.
- [5] Z. Morley Mao, H.W. So, B. Kang, R.H. Katz, "Network Support for Mobile Multimedia Using a Self-adaptive Distributed Proxy," *International Workshop on Network and OS Support for Digital Audio and Video*, June 2001.
- [6] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, D. Zagorodnov, "Wrapping Server-side TCP to Mask Connection Failures," Technical Report, Department of Computer Sciences, The University of Texas at Austin, July 2000.
- [7] F. Sultan, K. Srinivasan, L. Iftode, "MTCP: Transport Layer Support for Highly Available Network Services," *DCS TR-429*, Rutgers University.
- [8] V.C. Zandy and B.P. Miller, "ROCKS: Reliable Sockets," *Submitted for Publication*, March 2002.
- [9] M. Aron, P. Druschel, W. Zwaenepoel, "Efficient Support for P-HTTP in Cluster-Based Web Servers," *USENIX 1999*.
- [10] Ajay Bakre, B.R. Badrinath, "Handoff and System Support for Indirect TCP/IP," *USENIX Symposium on Mobile and Location-dependent Computing*, April 1995.
- [11] H. Balakrishnan, S. Seshan, E. Amir, R.H. Katz, "Improving TCP/IP Performance over Wireless Networks," *ACM Conference on Mobile Computing and Networking*, November 1995.
- [12] A. Snoeren, H. Balakrishnan, "An End-to-End Approach to Host Mobility," *ACM Conference on Mobile Computing and Networking*, August 2000.

- [13] A. Snoeren, D. Andersen, H. Balakrishnan, "Fine-Grained Failover Using Connection Migration," *Technical Report MIT-LCS-TR-812*, September 2000.
- [14] C. Yang, M. Luo, "Realizing Fault Resilience in Web-Server Cluster," *Super-Computing 2000*, November 2000.
- [15] D. Maltz and P. Bhagwat, "MSOCKS: An Architecture for Transport Layer Mobility," *In Proceedings of IEEE Infocom*, March 1998.
- [16] Mohit Aron, Darren Sanders, Peter Druschel, Willy Zwaenepoel, "Scalable Content-aware Request Distribution in Cluster-based Network Servers," *USENIX 2000*.
- [17] D. Maltz and P. Bhagwat, "TCP Splicing for Application Layer Proxy Performance," *Technical Report*, IBM, March 1998.
- [18] Y. Zhang and S. Dao, "A Persistent Connection Model for Mobile and Distributed Systems," *International Conference on Computer Communications and Networks*, September 1995.
- [19] X. Qu, J.X. Yu, and R.P. Brent, "A Mobile TCP Socket," *Technical Report TR-CS-97-08*, CSL, The Australian National University, Canberra, Australia, April 1997.
- [20] C. Liu and R. Jain, "Leading Causes of Wireless TCP Performance Degradation," *ACM Wireless Networks Journal*, August 2001.
- [21] G. Xylomenos, G. C. Polyzos, "TCP and UDP Performance over a Wireless LAN," *Proceedings of IEEE INFOCOM*, 1999.
- [22] C.A. Thekkath, T.D. Nguyen, E. Moy and E.D. Lazowska, "Implementing Network Protocols at User Level," *In Proceedings of ACM SIGCOMM*, September 1993.
- [23] W.R. Stevens and G.W. Wright, "TCP/IP Illustrated," Volumes I, II and III, *Addison Wesley Publications*.
- [24] J. Postel, "Transmission Control Protocol," *RFC 793, IETF*, September 1981.
- [25] C. Perkins, "IP Mobility Support," *RFC 2002, IETF*, October 1996.
- [26] W.R. Stevens, "UNIX Network Programming," *Prentice Hall Publications*.

APPENDIX A: Socket Class Interface

```
/*=====*/
SOCKET.HPP -- Interface to the socket_ct class
*=====*/

#ifndef _SOCKET_HPP
#define _SOCKET_HPP

// Included Header files
// Constants

#define MAX_NAMED_PIPE_LENGTH      32
#define SOCKET_MAX_IP_STRING_LENGTH 16

#define SOCK_CLIENT      1
#define SOCK_SERVER      2

#define CLI_MQNAME      "/app_comm_client.ahuja"
#define SRV_MQNAME      "/app_comm_server.ahuja"

class socket_ct
{
public:
    socket_ct ();    // Default constructor
    socket_ct (int);

    int  connect (char* servaddr, int port);
    int  bind (char* servaddr, int port);
    int  accept ();
    int  read (char *msg, int nbytes);
    int  write (char *msg, int nbytes);
    void close ();

    char* get_ip_local (void);
    int  get_port_local (void);

private:
    named_pipe_ct np;    // Named pipe
    int  mqfd; // Message Queue for incoming requests

    int  srvPort;
    char srvAddress[SOCKET_MAX_IP_STRING_LENGTH];
};

#endif // _SOCKET_HPP
```