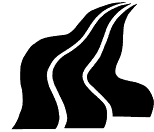
Faculty of Engineering and Science
Aalborg University



Department of Computer Science

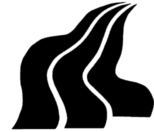
Join/Leave Protocol for Structured Peer-to-Peer Networks

Master Thesis

Group B1-215e - 10th semester

February 1st - June 10th, 2003

Faculty of Engineering and Science
Aalborg University



Department of Computer Science

TITLE: Join/Leave Protocol for Structured P2P Networks
Master Thesis

SEMESTER PERIOD:
SSE4, 10th semester
February 1st - June 10th, 2003

PROJECT GROUP:
B1-215e

GROUP MEMBERS:
Arūnas Vrubliauskas, aras@cs.auc.dk

SUPERVISOR:
Josva Kleist, kleist@cs.auc.dk

NUMBER OF COPIES: 3

NUMBER OF PAGES: 75

SYNOPSIS:

The aim of this project is to *implement* and *test* a Peer-to-Peer communication protocol, whose purpose is to assure a basic connectivity in the structured Peer-to-Peer network. The protocol implementation is based on the object-oriented analysis and design (OO&D) methodologies and it was implemented in C++ Programming Language using Standard Template Library (STL), POSIX *threads* and *sockets* Application Programmable Interfaces (APIs). Systems tests were conducted to verify conformance to the functional and non-functional requirements, and its target environment. The experimental results indicate that functional and non-functional (performance) requirements are met and further improvements are proposed.

Preface

This report has been written by project group B1-215e as a report for the second part of the Master Thesis in the International Masters Program in Software Systems Engineering of the Faculty of Engineering & Science, in the Computer Science Department at Aalborg University, Denmark, during the period from the 1st of February to the 10th of June, 2003.

This report is directed to people interested in distributed systems and peer-to-peer applications.

Figures, tables and formulas in the report are numbered in succession inside each chapter. Cross-references to formulas, figures, tables and appendix are written directly in the text. Cross-references to source material are specified with square brackets after the part of the text, where they are used, e.g. [2].

The source code of the Join/Leave protocol can be found at the following location:
http://www.cs.auc.dk/~aras/sse4/

Arūnas Vrubliauskas

Contents

1 Introduction **5**
- 1.1 Peer-to-Peer Systems . . . 5
  - 1.1.1 Peer-to-Peer Concept . . . 5
  - 1.1.2 Peer-to-Peer Systems Definition . . . 6
  - 1.1.3 Overview of the Peer-to-Peer Systems . . . 6
- 1.2 The FROST System . . . 8
  - 1.2.1 Limitations of the FROST System . . . 8
  - 1.2.2 Problem Statement . . . 8
- 1.3 Join/Leave Protocol Concepts . . . 9
  - 1.3.1 FROST Architecture Model . . . 9
  - 1.3.2 Node Data . . . 10

2 Join/Leave Protocol Implementation **12**
- 2.1 The Task . . . 12
  - 2.1.1 Purpose . . . 12
  - 2.1.2 Corrections to the Analysis . . . 12
  - 2.1.3 Quality Goals . . . 13
- 2.2 Technical Platform . . . 14
- 2.3 Architecture . . . 14
  - 2.3.1 Process Architecture . . . 14
  - 2.3.2 Component Architecture . . . 15
- 2.4 Model Component . . . 16
  - 2.4.1 Structure . . . 16
  - 2.4.2 Classes . . . 16
- 2.5 Function Component . . . 23
  - 2.5.1 Structure . . . 23
  - 2.5.2 Classes . . . 24
- 2.6 System Interface Component . . . 32
  - 2.6.1 Connection Class . . . 32
- 2.7 User Interface Component . . . 33

3 Join/Leave Protocol Testing **34**
- 3.1 Equipment . . . 34
  - 3.1.1 Cluster at Aalborg University . . . 34
  - 3.1.2 PlanetLab . . . 34
  - 3.1.3 Functional Tests . . . 35
  - 3.1.4 Stress Tests . . . 37
  - 3.1.5 Timing Tests . . . 39
- 3.2 Test Description . . . 40
  - 3.2.1 Functional Tests . . . 40

3.2.2 Stress Tests . . . 43
3.2.3 Timing Tests . . . 49
3.3 Test Results Analysis . . . 53
3.3.1 Functional Tests . . . 53
3.3.2 Stress Tests . . . 53
3.3.3 Timing Tests . . . 53

4 Conclusion **54**
4.1 Implementation . . . 54
4.2 Testing . . . 55
4.3 Further Work . . . 55

A Functional Tests: Actual Output **59**
A.1 Functional Test 1 . . . 59
A.2 Functional Test 2 . . . 62

B Functional Tests: Functional Testing System Output **63**
B.1 Functional Test 1: FT1.out . . . 63
B.2 Functional Test 2: FT2.out . . . 65

C Stress Tests: Ethereal Output **66**
C.1 Stress Test 1 . . . 66
C.2 Stress Test 2 . . . 67
C.3 Stress Test 3 . . . 68

D Timing Tests Output **72**
D.1 Timing Test 1 . . . 72
D.1.1 Test Case 1 . . . 72
D.1.2 Test Case 2 . . . 74

Chapter 1

Introduction

1.1 Peer-to-Peer Systems

1.1.1 Peer-to-Peer Concept

In general, Peer-to-Peer (P2P) systems are distributed systems without any centralized control or hierarchical organization, where Peer-to-Peer is a system architecture model in which each party has the same capabilities and either party can initiate a communication session. Other models with which P2P system architecture model might be contrasted include the client/server model and the master/slave model. In computer networking, master/slave is a model for a communication protocol in which one device or process (known as the master) controls one or more other devices or processes (known as slaves). The client/server model describes the relationship between two computer programs in which one program, the client, makes a service request to another program, the server, which fulfills the request. The main distinction between P2P and master/slave, client/server models is that in P2P system architecture model each machine can be both, a server and a client within the context of a given application. The machine which can serve as a server and a client is called the Peer. The analogy of a Peer-to-Peer system architecture model could be a telephone system where any person can call another person. Then the telephone model could be called a Person-to-Person system architecture model, where Person could be a receiver of a call or a caller, and no other Person has more privileges to make a call or receive a call. Same ideology can be applied to a P2P system architecture model where Peers are connected in a network and all Peers are working and communicating on an equal basis. The example of a P2P network can be seen in Figure 1.1.

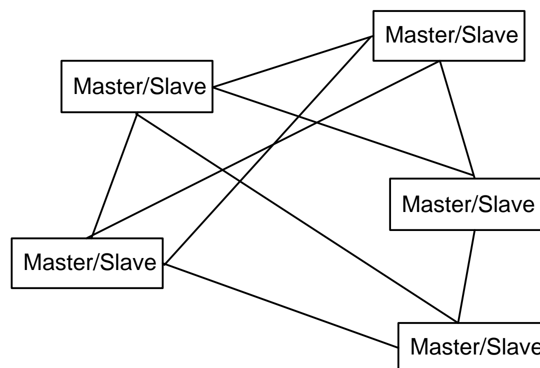


Figure 1.1: Example of a Peer-to-Peer network. Rectangular shapes represents the Peers, a line from one Peer to another represents a communication link.

1.1.2 Peer-to-Peer Systems Definition

In general it is agreed that there are two major architectures of a P2P systems: **hybrid** P2P systems (Figure 1.2 (a)) and **pure** P2P systems (Figure 1.2 (b)).

Hybrid P2P Systems: Hybrid P2P systems are regarded as centralized. They have a central server to perform administrative tasks. The server usually has a catalog of the Peer addresses that are referenced by a set of indexes. The main function of the server is to process lookup queries issued by Peers. The example of a lookup query can be as follows:

1. Peer A asks the server S to find the Peer X which has the resource R.
2. Server S performs search in its database on who has the resource R and if resource R is available, then server S returns the address of Peer X to Peer A.
3. Peer A connects directly to Peer X to use/get the resource R.

Pure P2P Systems: Pure P2P systems has no central server or router. All nodes are Peers, and each Peer may function as router, client, or server. Pure P2P systems can be classified depending on how the routing is achieved:

Distributed index: The resource index is fragmented and distributed to Peers.

Hashing index: Nodes and associated resources are indexed by unique IDs. Each ID is a hash value of a certain property (e.g. node ID - IP hash, resource ID - file name hash).

Flooding broadcast: A query is recursively broadcasted from one host to all its neighbors. Then query propagates until the resource is found or application-level counter TTL (Time To Live) reaches zero.

Also a combination of hybrid and pure P2P architectures has been successfully applied for some applications and has shown its potential use. **Super-Peer** architecture (Figure 1.2 (c)) presents a cross between pure and hybrid systems. A Super-Peer is a Peer that acts as a centralized server to a subset of clients. Clients submit queries to their Super-Peer and receive results from it, as in a hybrid system. However, Super-Peers are connected to each other as Peers in a pure system architecture.

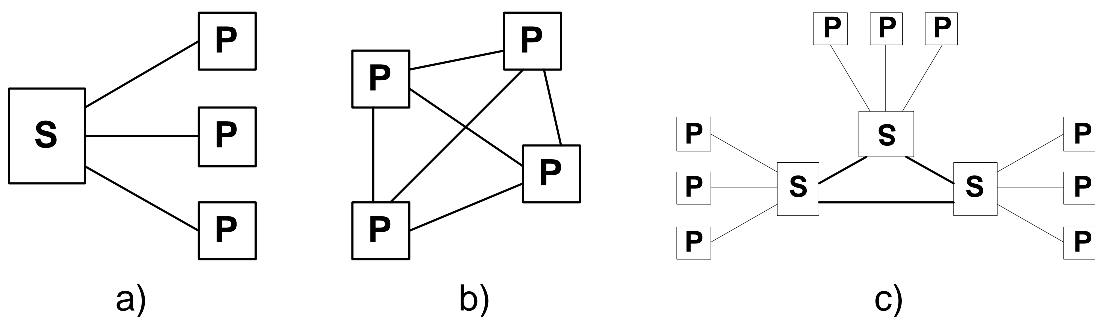


Figure 1.2: a) Hybrid P2P architecture, b) Pure P2P architecture, c) Super-Peer P2P architecture. S-box represents index server and P-box represents Peer.

1.1.3 Overview of the Peer-to-Peer Systems

P2P systems have recently received significant attention in both academia and industry. Most successful examples of P2P file sharing systems are systems such as Gnutella [15], Freenet [16] (and some other systems with similar features), where the main idea is to unite the users, who

wants to share the files, into a P2P network where users can easily find what they want, get what they want and give what they want. Another kind of P2P systems are systems that utilizes unused or wasted CPU-cycles of the idle machines in a P2P network. Those systems are mostly used by scientists who needs to get the results within a reasonable time for parallelizable computational problems that require a lot of CPU-cycles. Examples of such a systems could be SETI@home [6], THINK [13] and distributed.net [14].

SETI@home: The Search for Extraterrestrial Intelligence. They scan the sky using a large radio telescope and record the signals in a certain frequency. The computational problem here is that they must calculate Fast Fourier Transformation (FFT) of each signal to perform the analysis on the power spectrum of the signal. By analyzing the power spectrum of a signal they want to find the ”unusual” patterns in a signal, as a proof that extraterrestrial intelligence exists.

SETI@home is a master/slave distributed calculation system where master distributes work to the slaves. Master splits the signals into the work units, where length of the signal is around 100 seconds (a file size is around 340kb) and then distributes them to the clients. On the average home computer the processing of a data (one work unit) should take between 10 and 50 hours.

THINK: THINK project is a drug discovery system. System analyses each of the hundreds of millions of molecules to see if they are likely to interact with a target protein. THINK calculates and studies the many possible shapes, or conformers, the molecule might adopt interacting with the protein.

THINK is a master/slave distributed calculation system and like SETI@home, they have dedicated master server which distributes tasks to the slaves. The work unit contains approximately 10Kb of data and the CPU-time required varies from 4 hours to several days.

distributed.net: distributed.net project is based on solving mathematical problems such as Optimal Golomb Ruler, RSA, etc. At the moment when this report has been written the distributed.net was solving RC5-72, which is 72-bit RSA Data Security Secret Key Challenge. It is also master/slave distributed calculation system. The only difference compare to previous examples is that a slave can control how much work he can get depending on the available resources.

Common characteristic of those systems is that they are based on a client/server architecture where people around the world offer their spare CPU-cycles for a particular computational problem. The main drawback in the systems mentioned above is that a central server must be really powerful to be able to serve all the clients.

A more challenging approach to the distributed calculation could be to use a pure P2P architecture where all the clients are equal and allow everybody in the P2P network issue the computational problems, meaning that everybody can use others spare CPU-cycles for theirs purposes. One of such a systems is a distributed heterogeneous calculation platform FROST [1] which was developed in the Aalborg University.

1.2 The FROST System

The FROST system [1] was designed and implemented in the Aalborg University. The aim was to develop an API that will aid a programmer in developing applications such as SETI@home [6], THINK [13] and distributed.net [14]. From the FROST perspective the programmer is a user which defines a computational problem, defines how to split the computational problem into several pieces, or work units, that can be processed independently, and finally defines how to combine the results when they have been processed. Moreover, a user has to specify the algorithm that performs the calculations on the work units. Administration of the network communication between the machines, distribution of the work units to the machines in the FROST network and other administrative tasks are hidden from the user and are performed by the FROST system. Work units can be distributed only to the machines that are members of the FROST network. A machine is said to be a member of the FROST network if that machine has a FROST software running and other participants of the FROST network can communicate with that machine and use the CPU-cycles of that machine for solving some computational problem. The FROST network is based on a pure P2P architecture where all the machines are working on an equal basis, meaning that anybody in the network can use or give spare CPU-cycles to each other. Moreover, the machines or nodes in the FROST network are non-dedicated workstations, which are used for a daily purposes. Whereas the FROST system on these workstations run with low priority to assure that FROST uses only those CPU-cycles that are unused.

1.2.1 Limitations of the FROST System

One of the limitations of the FROST system is that a current implementation scales only to a local area network (LAN). The FROST developers indicated that a problem in scaling the FROST system is the bottleneck induced by master and information sharing. The bottleneck induced by master is a situation when a master node must handle thousands of clients and thus the master becomes a bottleneck either because of the network bandwidth or speed of a master node. Another problem in scaling the FROST system is the information sharing, since all the nodes in the FROST network has to share the information required for the load balancing and other administrative tasks and it is currently done by using broadcast communication. It is normal to use broadcast communication on the local area network, but it is unprofitable for the Internet wide communication.

1.2.2 Problem Statement

A scalable solution for the FROST system was proposed in [2] where the Join/Leave protocol was designed and verified using the SPIN [18, 19] verification tool. This work is a continuation of a previous work [2] and there are two main goals that motivate this study:

1. *Prototype Implementation.* Prototyping is an efficient software development technique which helps to better understand the environment and the requirements being addressed. A prototype is a demonstration of what's actually feasible with existent technology, and where the technical weak spots still exists. In this part the main goal is to implement a prototype of the Join/Leave protocol and prepare it for the system testing.

2. *System Testing.* System testing is an important process for assuring software quality in an environment of complex defect-prone components. In general, system testing focuses on the complete system, its functional and non-functional requirements, and its target environment. The following system tests will be conducted:

   Functional testing. Functional testing, also called requirements testing, tests if the system perform as promised by the requirements specification.

Performance testing. Performance testing is used to test if the non-functional requirements are met. Two types of performance tests were conducted:

Stress tests: The purpose of the stress tests is to evaluate the system when stressed to its limits over a short period of time.

Timing tests: The purpose of the timing tests is to validate conformance to behavioral and performance constraints and evaluate if the system is fast enough.

1.3 Join/Leave Protocol Concepts

In this section some required concepts, which were defined in the previous work [2] will be introduced.

1.3.1 FROST Architecture Model

The structured indirect communication model was chosen because it promises to avoid the problem induced by master and the information sharing can be done efficiently. The nodes in such a model form the groups of nodes, where each group has one master node. Information between the nodes is shared inside the group, whereas group masters can share the information between other group masters as can be seen in Figure 1.3. Another advantage of the model is that it is a decentralized model meaning that there is no need to invest in dedicated machines.

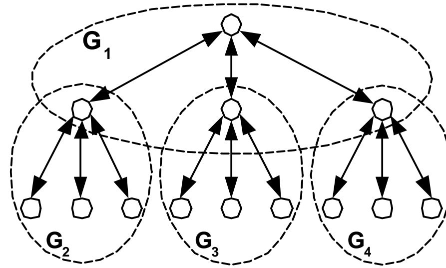


Figure 1.3: Simple structured indirect communication architecture model with 4 groups of nodes.

The basic structured indirect communication model was modified by removing the root node from the architecture, with intent to remove the central point of failure in the model. Then the FROST architecture model will look as presented in Figure 1.4. The FROST architecture model can be described using 3 parameters:

Base: **Base** parameter gives an upper bound for the number of slaves in a group of nodes. For example, the **Base** of the architecture model presented in Figure 1.4 is: **Base** $= 4$, thus the fictitious node N_0 can have a maximum of 4 slaves and they are N_1, N_2, N_3 and N_4. This rule holds for all master nodes (N_1,N_2, etc.) in the architecture.

Level: **Level** parameter gives the number of levels in the model. For example, the **Level** of the architecture model presented in Figure 1.4 is: **Level** $= 3$, meaning that model has three hierarchical levels starting from level L_1 to L_3.

Size: **Size** parameter gives the number of nodes in the model. The size can be calculated using the following equation: $\frac{1-B \cdot l}{1-B} - 1$, where $l = B^L$, B - **Base**, L - **Level**. For example, the **Size** of architecture model presented in Figure 1.4 is: **Size** $= 84$.

The FROST model could scale to a large number of nodes, for instance if base and level of a model is $B = 100$ and $L = 7$ respectively, then the size $S \approx 10^{14}$, which is very large number

of nodes and the worst case number of hops from one node to another is only $HOPS = 13$, where $HOPS = 2 \cdot L - 1$. For example, in Figure 1.4 the worst case number of hops is $HOPS = 5$.

The nodes in the model are organized according to the performance of nodes, meaning that higher performance nodes are located *higher* in the hierarchy, for instance in Figure 1.4, nodes $N_1, ..., N_4$ have highest performance and are located in level L_1, nodes with low performance are located in level L_3.

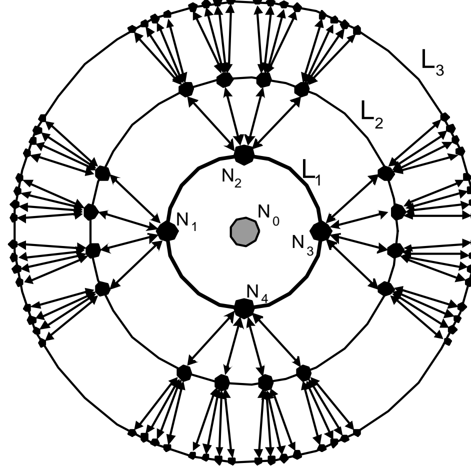


Figure 1.4: FROST architecture model. Base 4, Level 3, Size 84.

1.3.2 Node Data

The following describes the data structures used by the Join/Leave protocol:

Static performance (SP): *SP* is a static node performance.

Global performance (GP): *GP* is a global performance of a group including subgroups. A knowledge about a global performance will support a decision making on where the new nodes should join to sustain as well balanced FROST architecture as possible. When the FROST system is operational, the *GP* values of the highest level nodes should be close. For instance, if nodes N_1, N_2 and N_3 are from highest level, then: $GP_1 \approx GP_2 \approx GP_3$. A global performance is maintained by each node in the FROST system and is calculated as shown in equation 1.1. Note, that when a node has no slaves, the sum evaluates to zero and the *GP* is equal to a static node performance *SP*.

$$GP_i = SP_i + \sum_{j \in slaves(GP_i)} GP_j \qquad (1.1)$$

GP_i - global performance of node N_i,

GP_j - global performance of node N_j, node N_j is a slave of master N_i,

For example, consider the fragment of a FROST system in Figure 1.5. The calculation of global performance GP_1 of node N_1 is as follows: using the equation 1.1 and following the bottom-up direction in the FROST system the global performance of node N_1 is:
$GP_1 = GP_2 + GP_3 + GP_4$, where $GP_2 = GP_5 + GP_6 + GP_7$ and $GP_4 = GP_8 + GP_9$.

Base (B): *B* is a constant which defines the base of a FROST architecture. The base value is an upper bound for the number of slaves a master can have. For instance, in the FROST

architecture fragment in Figure 1.5, the base is $B = 3$, and the maximum number of slaves each node can have is 3. Thus, node N_1 and node N_2 has maximum number of slaves, node N_4 could have one more slave and node N_3 could have three more slaves.

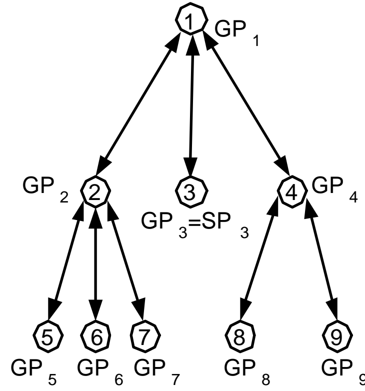


Figure 1.5: The FROST system fragment.

Level (level): *level* parameter shows to which level in the FROST architecture a particular node belongs. For instance, assume that in Figure 1.5 node N_1 belongs to the highest level, then level parameter for node N_1 is $level = 1$, N_2 level $level = 2$, N_5 level $level = 3$, etc.

Local slaves (s): s parameter shows how many slaves particular node has. For instance, in Figure 1.5 node N_1 has three slaves $s = 3$, node N_3 has no slaves $s = 0$, etc.

Slave address list (SAL): *SAL* is a list of the slave addresses. The *SAL* list is the fundamental data structure in the FROST system, since it specifies the relations between the nodes in the FROST architecture.

Chapter 2

Join/Leave Protocol Implementation

2.1 The Task

2.1.1 Purpose

The purpose of the Join/Leave protocol is to handle the node joins and departures in the FROST network, whose system architecture and concepts were described in [2]. In general the Join/Leave protocol can be divided into two parts based on the function it should perform:

Join: The Join/Leave protocol must assure that a node which joins the network will be organized in the hierarchy according to its static performance measure - *SP*, which represents the relation between the available node resources (bandwidth, CPU and main memory) and its place in the hierarchy. If nodes are arranged according to their static performance, the workload to maintain the network structure will be accordingly distributed between the nodes.

Leave: When nodes leave the network (voluntarily or by failing) the Join/Leave protocol has to assure the integrity of the communication architecture by appropriately rearranging the related nodes in the network.

The Join/Leave protocol should be used as the communication component of the FROST system first described in [1].

2.1.2 Corrections to the Analysis

Some corrections have been made from the analysis and design [2] of the Join/Leave protocol:

Static performance (*SP*): Previously, the static node performance was derived by using the static performance parameters: bandwidth, CPU and main memory. However, the network bandwidth (and not storage space or computation time) is presently the most limited resource in P2P networks [4]. Any node joining the network must send at least some number of maintenance messages. According to the join procedure described in [2] the nodes start joining the network from the highest level and that would yield higher traffic of the maintenance messages at the higher levels of the FROST communication network architecture. The implication is that the static node performance measure can be derived considering only the bandwidth. Thus, the nodes which have more bandwidth will join the higher levels and nodes with less bandwidth will join the lower levels of the network architecture.

Join procedure: The joining procedure is performed by node N_{join} and was previously defined as follows:

1. Get a list of the highest level nodes from node N_0.
2. Ask any node in the highest level where to join. (Answer is an address of some node N_{ask}).
3. Ask node N_{ask} where to join. (Answer is an address of some node N_{ask}).
4. Repeat 3, until N_{ask} accepts node N_{join}. (N_{join} is a leaf node after the join phase).
5. Trigger the adaptation to the network if necessary. (Adaptation follows the bottom-up direction).

An optimization was made to the join procedure presented. Instead of joining at the bottom of the tree and then triggering the adaptation to the network, the joining node could be aware of the possible adaptation while performing step 3; i.e. if it turns out that the *SP* value of the joining node N_{join} is higher than of node N_{ask}, then node N_{join} could trigger the adaptation to the network by pushing the node N_{ask} downwards. Thus the overall effect of this new scheme should reduce the rate of change in the network and thus the number of maintenance messages, which in turn will reduce the bandwidth consumption. To summarize, the optimized joining procedure is performed as follows:

1. Get a list of the highest level nodes from node N_0.
2. Ask any node in the highest level where to join. (Answer is an address of some node N_{ask}).
3. Ask node N_{ask} where to join (answer is an address of some node N_{ask}) and trigger the adaptation to the network if necessary. (Adaptation follows the top-down direction).
4. Repeat 3, until N_{join} joins.

2.1.3 Quality Goals

Table 2.1 shows the prioritization of design criteria. A special weight is placed on reliability, correctness and usability since these characteristics are critical for whether the system will be used at all. The main intent to implement a prototype of the Join/Leave protocol is to test and measure the performance of the protocol. It should be possible to test the system for ensuring that the system performs its intended functions. Also the system should be flexible and comprehensible to

Criterion	Very important	Important	Less important	Irrelevant	Trivially fulfilled
Usable	X				
Secure				X	
Efficient	X				
Correct	X				
Reliable	X				
Maintainable			X		
Testable		X			
Flexible		X			
Comprehensible		X			
Reusable				X	
Portable			X		
Interoperable			X		

Table 2.1: Prioritization of design criteria.

reduce the cost of modification to the protocol implementation if necessary. To concentrate on the functionality of the protocol and ability to evaluate it all other characteristics have been prioritized lower or irrelevant. However it should be noted that some characteristics (i.e. security, portability, interoperability) that were left out have to be considered as important in later development of the Join/Leave protocol:

Secure. The maintenance protocols are especially susceptive to the DoS (Denial of Service) attacks. Since the Join/Leave protocol is intended to operate Internet wide there is a high risk of such attacks.

Portable. The protocol should be able to operate on various technical platforms to increase the number of potential users of the FROST system [1].

Interoperable. This characteristic is important when coupling the Join/Leave protocol with FROST system [1].

2.2 Technical Platform

Equipment. The computerized system is designed for use on the non-dedicated workstations that are interconnected via network (LAN, Internet, etc.). There is no need to have an expensive high speed machine to assure the basic connectivity in the FROST network since the main limitation is the available bandwidth. Thus, the minimum requirements are: non-archaic PC (Personal Computer) with NIC (Network Interface Card) or modem installed and an active connection to the network.

System Software. Linux OS (Operating System) will be used to implement, test and run the Join/Leave protocol. The design is based on implementing the system in C++ programming language. The C++ programming language has to have an API (Application Programmable Interface) to POSIX *threads* and *sockets*.

Design Language. The design is based on the UML (Unified Modeling Language) notation.

2.3 Architecture

2.3.1 Process Architecture

The physical architecture of the FROST network managed by the Join/Leave protocol (Frost Client component) is shown in Figure 2.1. A Node refers to a PC which fulfills the requirements of the technical platform and uses the FROST client software to be a part of the FROST network. Nodes communicate using TCP/IP Internet protocol, where reliable data delivery is provided by a connection-oriented TCP transport protocol. Technical platform component has an interface to the various OS components, including POSIX *threads* and *sockets*. The Frost Client component comprises the model and functions of the Join/Leave protocol and is responsible for the basic connectivity in the FROST network.

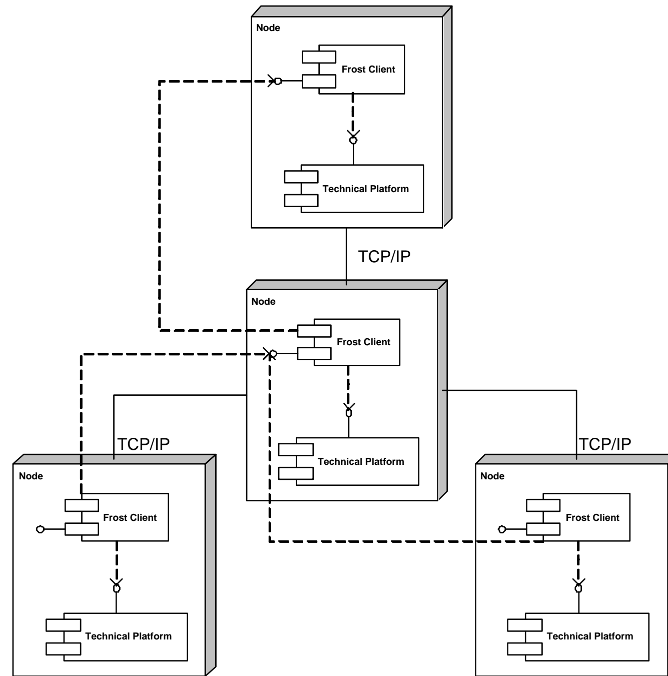


Figure 2.1: Deployment Diagram. The FROST network of four nodes. Dashed arrows represent dependency associations between nodes.

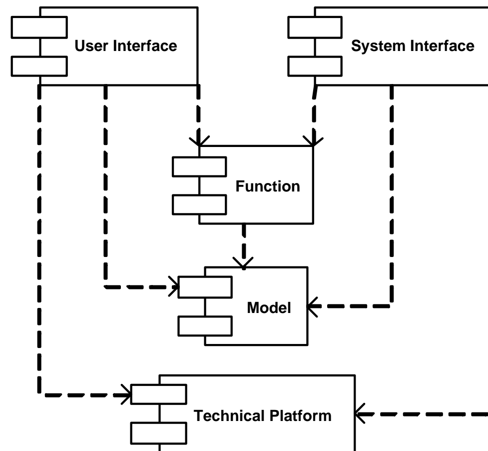


Figure 2.2: Component Diagram. Dashed arrows represent dependency associations between components.

2.3.2 Component Architecture

The Frost Client component could be decomposed using a design pattern in Figure 2.2 as follows:

User Interface Component. A part of a system implementing the interaction with users.

System Interface Component. A part of a system implementing the interaction with other systems.

Model Component. A part of a system that implements a model of the Join/Leave protocol.

Function Component. A part of a system that implements functional requirements of the Join/Leave protocol.

2.4 Model Component

The Model component is a part of a Join/Leave protocol that handles data storage. The purpose of the component is to control and deliver data to functions, interfaces, users and other computerized systems. The event Table 2.2 for the Model component follows from the use case diagram of the Frost client component shown in Figure 2.3. As can be seen from the event table there are three

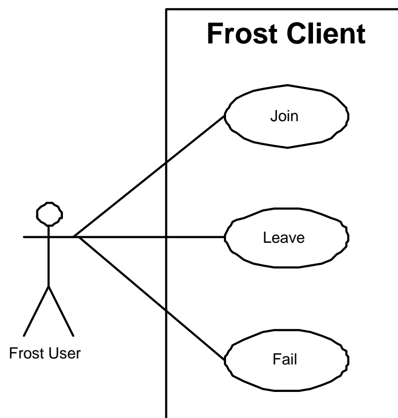


Figure 2.3: Use Case Diagram for the Frost client component.

	Class				
Event	Expector	Connector	SlaveList	AddrList	ConnQueue
Joined	$\star$	$\star$	+	+	$\star$
Left			+	+	+
Failed			+	+	

Table 2.2: Event table for the Model component. $\star$ - multiple modifications to an object; + - onetime modification to an object.

main events that causes the change of state in a model. The system is in the *joined* state when a node is connected to the FROST network and the system is in the *left* or *failed* state if a node has disconnected from the FROST network either by voluntarily leaving or by failing respectively. The behavioral pattern of this situation can be seen in Figure 2.4.

2.4.1 Structure

The class diagram for the Model component is shown in Figure 2.5. All classes are described in the following.

2.4.2 Classes

The following contains a specification of the classes from the class diagram in Figure 2.5.

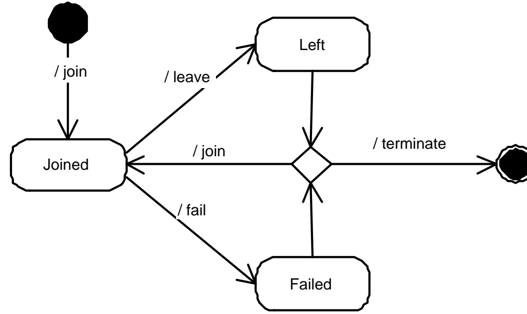


Figure 2.4: State Chart Diagram for the Model component.

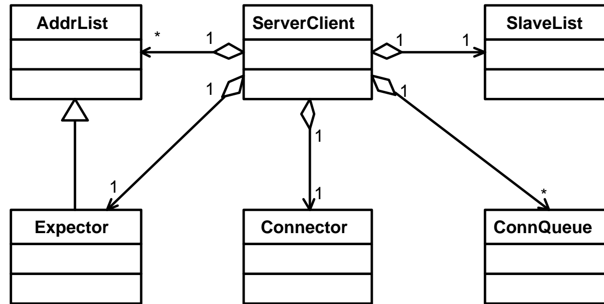


Figure 2.5: Class Diagram for the Model component.

Class ServerClient

Purpose: It is the main class in the system, it contains the data model of the Join/Leave protocol. The rest of the Model component classes are the parts of this class (aggregation relation) as can be seen from the class diagram in Figure 2.5. Also, it has a control over the Function component (see Figure 2.2) by being able to use its functions accordingly.

Attributes: To classify the attributes by their purpose, the attributes are represented in distinct tables: attributes that are fundamental for the Join/Leave protocol are shown in Table 2.3, attributes that are used to interact with Function component (see Section 2.5) are shown in Table 2.4 and some other important attributes of the class are shown in Table 2.5.

Attribute	Type	Purpose
ID	int	The unique identifier of a node.
SP	int	The static performance parameter of a node.
GP	int	The global performance parameter of a node.
level	int	Shows to which performance level a particular node belongs.
base	const int	The base of the FROST architecture.
SAL	**AddrList** $\star$	The list which contains the addresses of the slave nodes.

Table 2.3: Join/Leave protocol specific attributes. Abbreviation "$\star$" represents a pointer to the object leftwards.

Attribute	Type	Purpose
acceptor	**Acceptor** $\star$	See section 2.5.2 for more details.
queue	**QueueHandler** $\star$	See section 2.5.2 for more details.
vips	**VIPHandler** $\star$	See section 2.5.2 for more details.
joiner	**JoinHandler** $\star$	See section 2.5.2 for more details.
askers	**AskersHandler** $\star$	See section 2.5.2 for more details.
pusher	**PushHandler** $\star$	See section 2.5.2 for more details.
leaver	**LeaveHandler** $\star$	See section 2.5.2 for more details.
failer	**FailHandler** $\star$	See section 2.5.2 for more details.
master	**MasterHandler** $\star$	See section 2.5.2 for more details.
SHL	**SlaveList** $\star$	The list which contains pointers to the **SlaveHandler** objects (see section 2.5.2 for more details about the **SlaveHandler** class).

Table 2.4: Attributes that are used to interact with the Function component. Abbreviation "$\star$" represents a pointer to the object leftwards.

Attribute	Type	Purpose
connector	**Connector** $\star$	See **Connector** class for details.
expector	**Expector** $\star$	See **Expector** class for details.
cq	**ConnQueue** $\star$	The **ConnQueue** class contains a list of the **Connection** objects (see class **Connection** for details). The *cq* list is the waiting list for the active connections that were accepted by the **Acceptor** (see in section 2.5.2) thread and added by **Expector** object (see **Expector** class for details). The list is processed by the **QueueHandler** (see section 2.5.2) thread.
vip	**ConnQueue** $\star$	The *vip* list is a waiting list for the active connections that were accepted by the **Acceptor** thread and added to the *vip* list by the **Expector** object. The list is processed by the **VIPHandler** (see section 2.5.2) thread.
AskWL	**AddrList** $\star$	The *AskWL* list is a waiting list for the nodes that are waiting for an answer where to join. Nodes are added to the list by the **QueueHandler** thread and processed by the **AskersHandler** thread. (See section 2.5.2 for more details.)
mutex	pthread_mutex_t	*mutex* is a mutual exclusion device, which is used for protecting shared data structures from concurrent modifications.
mutex_pl	pthread_mutex_t	*mutex_pl* is a mutual exclusion device, which is used to control the concurrent executions of the functions (threads) provided by the Function component.

Table 2.5: Other attributes. Abbreviation "$\star$" represents a pointer to the object leftwards.

Operations: The class operations are summarized in Table 2.6.

Operation	Purpose
join()	This operation is used to join the FROST network.
leave()	This operation is used to voluntarily leave the FROST network.
fail()	This operation is used to simulate a fail situation.
spawn_acceptor()	Starts **Acceptor** thread. See section 2.5.2 for details.
spawn_queue()	Starts **QueueHandler** thread. See section 2.5.2 for details.
spawn_vip()	Starts **VIPHandler** thread. See section 2.5.2 for details.
spawn_joiner()	Starts **JoinHandler** thread. See section 2.5.2 for details.
spawn_askers()	Starts **AskersHandler** thread. See section 2.5.2 for details.
spawn_pusher()	Starts **PushHandler** thread. See section 2.5.2 for details.
spawn_leaver()	Starts **LeaveHandler** thread. See section 2.5.2 for details.
spawn_failer()	Starts **FailHandler** thread. See section 2.5.2 for details.
spawn_master()	Starts **MasterHandler** thread. See section 2.5.2 for details.
spawn_slave()	Starts **SlaveHandler** thread. See section 2.5.2 for details.

Table 2.6: **ServerClient** *operations.*

Behavior: The general behavioral pattern of this class can be seen in Figure 2.4. An interaction between the actor and the **ServerClient** class is shown in Figure 2.6. Two use cases are

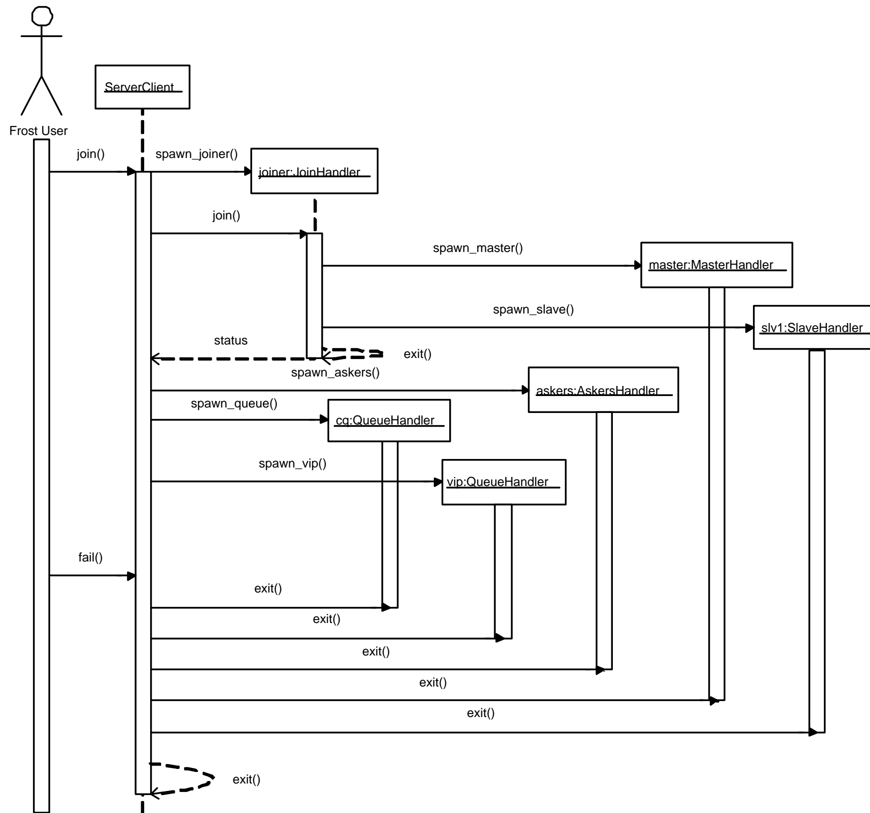


Figure 2.6: Sequence Diagram with concurrent objects.

shown: Join use case and Fail use case. Sequence diagram in Figure 2.6 also shows the

creation and lifetime of certain objects involved in the use cases.

Class Connector

Purpose: Given an Internet address the **Connector** class is responsible for opening a connection to a remote node.

Attributes :

Attribute	Type	Purpose
PORT	const int	It is a well known port used for communication in a FROST network.

Table 2.7: **Connector** *attributes.*

Operations :

Operation	Purpose
connect_to()	Given an Internet address it opens a connection to a remote node. The operation returns **sockaddr_in** structure and socket descriptor *fd*.

Table 2.8: **Connector** *operations.*

Class AddrList

Purpose: This class is used to store and maintain a list of Internet addresses.

Attributes :

Attribute	Type	Purpose
al	vector<uint32_t>	*al* is a STL (Standard Template Library) container used to store the 32-bit Internet addresses.
bounded	const bool	Determines if a list is bounded or not.
base	const int	The base of the FROST architecture. If *al* list is bounded then *base* parameter is used to check the boundaries of *al* list.
mutex	pthread_mutex_t	*mutex* is a mutual exclusion device, which is used for protecting the *al* list from concurrent modifications.
wl_not_empty	pthread_cond_t	*wl_not_empty* is a condition variable, which is used to signal the waiting thread if *al* list changed its state from empty to not empty.

Table 2.9: **AddrList** *attributes.*

Operations :

Operation	Purpose
add()	This operation is used to add an Internet address to the *al* list.
rem()	This operation is used to remove an Internet address from the *al* list.
get()	This operation is used to retrieve an Internet address from the *al* list.
copy()	This operation is used to make a copy of the *al* list.
clear()	This operation is used to remove all elements from the *al* list.
getSize()	This operation is used to get the size of the *al* list.

Table 2.10: **AddrList** *operations.*

Class Expector

Purpose: This class is derived from the base class **AddrList** and inherits all the attributes and operations from the **AddrList** class. However, additional functionality is added to this class. If a node is expecting a connection from a particular remote node or nodes then the **Expector** class is responsible for storing the Internet addresses of the expected nodes. When the **Acceptor** accepts the new connection **Expector** verifies if the connection is expected or not. If connection is expected, then **Expector** adds a **Connection** object to the *vip* queue (class **ConnQueue**), otherwise a **Connection** object is added to a conventional queue *cq* (class **ConnQueue**).

Attributes :

Attribute	Type	Purpose
master	uint32_t	If node is expecting a connection from the master node, then an Internet address of the mater node is stored in the *master* attribute.

Table 2.11: **Expector** *attributes.*

Operations :

Operation	Purpose
exp_master()	This operation is used to set the *master* attribute with an Internet address.
exp_slave( )	This operation is used to add an Internet address to the *al* list.
isExpected()	This operation is used to determine if a given address *addr*: $addr = master$ or $addr \in al$ list. If one of the two statements is true then a remote node with address *addr* is expected. Return value is *true* if node is expected.

Table 2.12: **Expector** *operations.*

Class SlaveList

Purpose: This class is used to store and maintain a list of pointers to active **SlaveHandler** threads (see section 2.5.2 for **SlaveHandler** class details).

Attributes :

Attribute	Type	Purpose
sl	vector<**SlaveHandler** ⋆ >	*sl* is a STL (Standard Template Library) container used to store pointers to the **SlaveHandler** objects.
mutex	pthread_mutex_t	*mutex* is a mutual exclusion device, which is used for protecting the *sl* list from concurrent modifications.

Table 2.13: **SlaveList** *attributes.*

Operations :

Operation	Purpose
add()	This operation is used to add a **SlaveHandler** object pointer to the *sl* list.
rem()	This operation is used to remove a **SlaveHandler** object pointer from the *sl* list.
get()	This operation is used to retrieve a **SlaveHandler** object pointer from the *sl* list.
clear()	This operation is used to remove all elements from the *sl* list.
getSize()	This operation is used to get the size of the *sl* list.

Table 2.14: **SlaveList** *operations.*

Class ConnQueue

Purpose: **ConnQueue** class implements the FIFO buffer of **Connection** objects (see System Interface component in section 2.6 for details about **Connection** class).

Attributes :

Attribute	Type	Purpose
conn_queue	list<**Connection** ⋆ >	*conn_queue* is a STL (Standard Template Library) container used to store pointers to the **Connection** objects.
queue_mutex	pthread_mutex_t	*queue_mutex* is a mutual exclusion device, which is used for protecting the *conn_queue* buffer from concurrent modifications.
queue_not_empty	pthread_cond_t	*queue_not_empty* is a condition variable, which is used to signal the waiting thread if *conn_queue* buffer is not empty.

Table 2.15: **ConnQueue** *attributes.*

Operations :

Operation	Purpose
add()	This operation is used to add a **Connection** object pointer to the *conn_queue* buffer.
rem()	This operation is used to remove a **Connection** object pointer from the *conn_queue* buffer.
pop()	This operation is used to retrieve the first element from the *conn_queue* buffer.
clear()	This operation is used to remove all elements from the *conn_queue* buffer.
getSize()	This operation is used to get the size of the *conn_queue* buffer.

Table 2.16: **ConnQueue** *operations.*

2.5 Function Component

The Function component implements the functional requirements of the Join/Leave protocol described in [2]. In this section the functional requirements will be transformed into a collection of operations, each of which is tied to a new class in the Function component.

2.5.1 Structure

The class diagram for the Function component is shown in Figure 2.7. Each class that implements an operation is derived from the abstract base class ***Thread***, which also implements a system interface to the POSIX threads. The implication is that each operation has its own execution thread and thus each operation could be executed concurrently with other operations. All classes are described in the following.

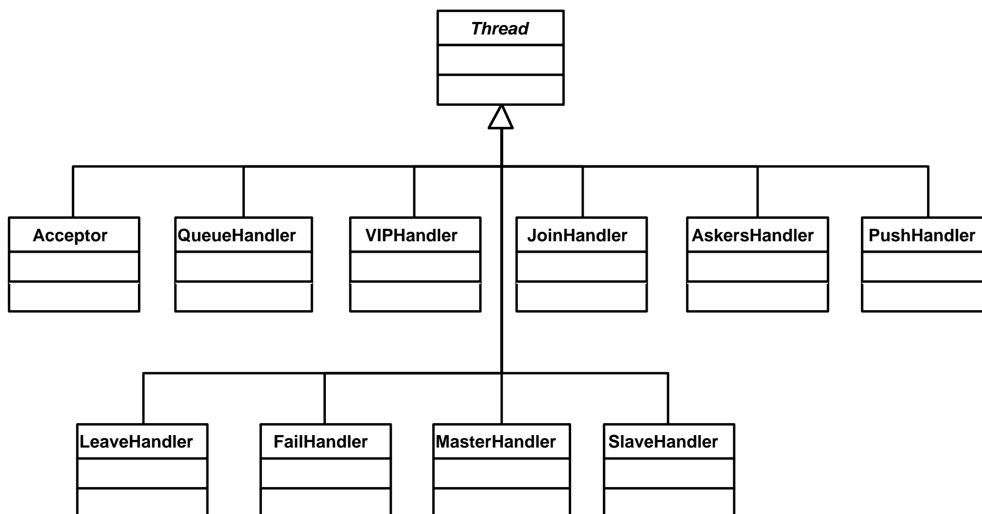


Figure 2.7: Class Diagram for the Function component.

2.5.2 Classes

Thread

Purpose: ***Thread*** class is an abstract class (objects cannot be created from this class), which implements an interface to the POSIX threads and provides with a set of operations to control and synchronize the execution of a thread.

Attributes :

Attribute	Type	Purpose
thread	pthread_t	Thread identification number.
mutex	pthread_mutex_t	*mutex* is a mutual exclusion device, which is used in combination with the *cond* (see below) attribute to suspend and resume a thread identified by *thread*.
cond	pthread_cond_t	*cond* is a condition variable, which is used to signal the waiting thread identified by *thread*.

Table 2.17: ***Thread*** *attributes.*

Operations :

Operation	Purpose
run()	This operation is executed by the thread identified by *thread* attribute. *run()* operation is *virtual* (derived class should override this operation and provide an implementation for it.)
join()	This operation suspends the execution of the caller until the thread identified by *thread* attribute terminates, either by calling *t_exit()* or by being canceled (*cancel()*).
wait()	This operation suspends the execution of the thread identified by *thread* attribute until the *signal()* or *cancel()* operation is called.
signal()	This operation is used to resume the execution of the thread identified by *thread* attribute.
t_exit()	This operation is used to terminate the execution of the thread identified by *thread* attribute.
cancel()	This operation is used to cancel the execution of the thread identified by *thread* attribute.
clean()	This operation is executed after the thread identified by *thread* attribute has been canceled. The purpose of this operation is to free the resources that a thread may hold at the time it terminates. *clean()* operation is *virtual* (derived class should override this operation and provide an implementation for it.)

Table 2.18: ***Thread*** *operations.*

Acceptor

Purpose: **Acceptor** class is derived from the base class ***Thread***. **Acceptor** is responsible for handling the incoming connection requests.

Behavior: First, **Acceptor** opens a socket on a well known port and starts to listen for the incoming connection requests. When such a request is received, **Acceptor** accepts the connection and creates a **Connection** object for it. Then the **Expector** is used to verify if this connection is expected. If connection is expected then a **Connection** object is added to the *vip* queue, otherwise a **Connection** object is added to the *cq* queue (see Table 2.5 for *vip* and *cq* queue details). Finally, when a **Connection** object is dispatched the **Acceptor** is ready to accept new connections. **Acceptor** behavioral pattern is shown in Figure 2.8.

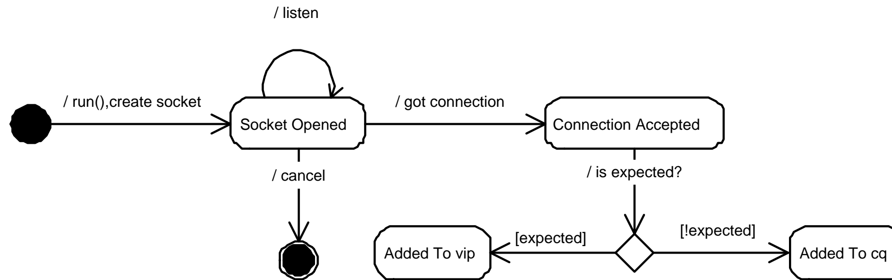


Figure 2.8: State Chart Diagram for the **Acceptor** *operation.*

QueueHandler

Purpose: **QueueHandler** class is derived from the base class ***Thread***. The purpose of this class is to process the **Connection** objects waiting in the *cq* queue (see Table 2.5 for *cq* queue details). The processing consist of finding out the reason why a remote node established a connection and decide how the connection should be processed further.

Behavior: If the *cq* queue is empty then the **QueueHandler** is in the idle state (the execution thread is suspended). **QueueHandler** resumes its execution if a **Connection** object or objects were added to the *cq* queue. When a **Connection** object is retrieved from the queue the **QueueHandler** communicates (using a **Connection** object) with a remote node to find out the reason why the connection has been established. According the Join/Leave protocol specification, if connection is not expected (and it is not, since the expected connections are in the *vip* queue) then a remote node is trying to join the FROST network. However, the **QueueHandler** class is designed with a perspective that there could be some other reasons (including attacks) that could be processed by **QueueHandler**. Behavioral pattern for the **QueueHandler** class is shown in Figure 2.9.

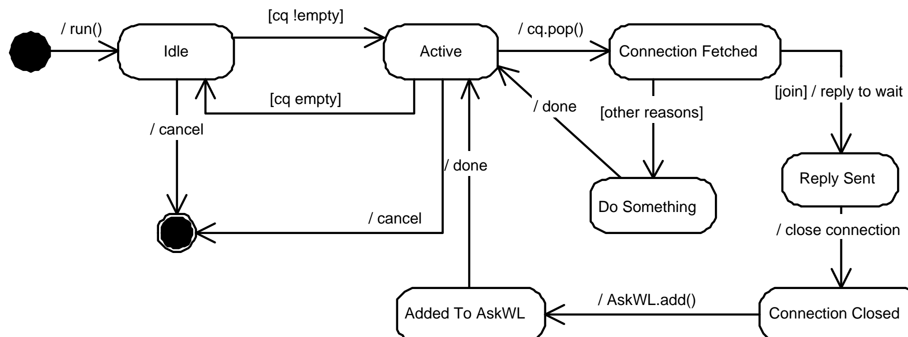


Figure 2.9: State Chart Diagram for the **QueueHandler** *operation.*

VIPHandler

Purpose: **VIPHandler** class is derived from the base class ***Thread***. The purpose of this class is to process the **Connection** objects waiting in the *vip* queue (see Table 2.5 for *vip* queue details). The reason to have an additional queue together with the conventional *cq* queue is to provide the means of almost immediate processing of connections that are expected to be established.

Behavior: Same as with **QueueHandler** the **VIPHandler** is suspended if a queue it has to process is empty and it will be resumed when queue is not empty. There are two types of expected connections, either the connection is meant to be with a new slave or a new master. If connection established is with new slave then a **SlaveHandler** object is created to handle a **Connection**, otherwise a **MasterHandler** object is created. Behavioral pattern for the **QueueHandler** class is shown in Figure 2.10.

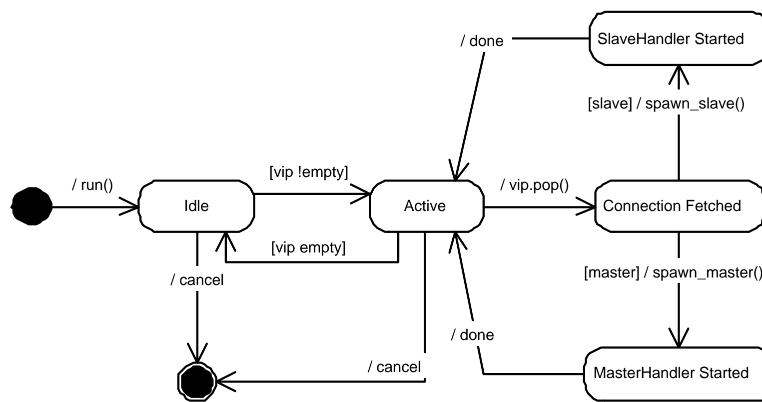


Figure 2.10: State Chart Diagram for the **VIPHandler** *operation.*

JoinHandler

Purpose: **JoinHandler** class is derived from the base class ***Thread***. **JoinHandler** is responsible for joining the FROST network.

Behavior: When **JoinHandler** is started, first, it contacts the discovery server (node N_0) to retrieve a list of nodes located at the highest level of the FROST architecture. One node is randomly chosen from a list to ask where to join. If, however, a chosen node is not responding then there is a possibility to choose another node from a list. After a query is dispatched a node waits for an answer about further join instructions. Since the FROST network is dynamic the answer could be provided by another node which took responsibility to process asking node. Behavioral pattern for the **JoinHandler** class is shown in Figure 2.11. The are three types of replies an asking node could receive:

Ask next: It says that there are no vacant positions in a group and a node has to ask the next group master, which is chosen by a current group master. The next master an asker has to contact will be a slave node with min(*GP*) value in a current group. The reply includes an address of a chosen slave (next master).

Join granted: "Join granted" reply is sent if current master has a vacant position in a group and asker is welcome to join. This reply also indicates that the *SP* value of the master node is larger than asker's and there was no need for adaptation.

Join granted (push): This reply indicates that the *SP* value of an asker node is larger than master's and that master is ready to concede its place by pushing itself downwards in the FROST architecture. The reply includes the addresses of slave nodes and master node (master of the current master).

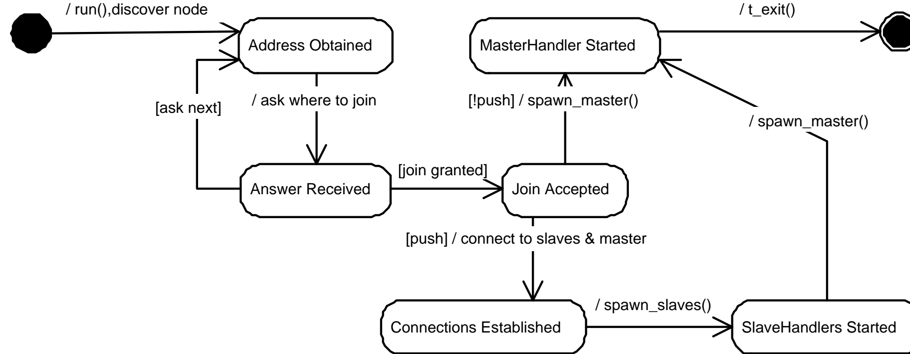


Figure 2.11: State Chart Diagram for the **JoinHandler** *operation.*

AskersHandler

Purpose: **AskersHandler** class is derived from the base class ***Thread***. **AskersHandler** is responsible for providing the join instructions for a node which asks for them. The addresses of nodes that are waiting for the join instructions are stored in the *AskWL* waiting list.

Behavior: **AskersHandler** is idle if *AskWL* waiting list is empty, otherwise **AskersHandler** pops an address from the list and establishes the connection with an asker node. When the connection is established (**Connection** object is created), one of the three situations could happen:

- $SP_{local} > SP_{remote}$, $s < B$: asker is allowed to join as a slave. **AskersHandler** sends the "Join granted" message to an asker node and creates a **SlaveHandler** object to handle a **Connection**.
- $SP_{local} > SP_{remote}$, $s = B$: a group is complete and there are no vacant places. In this case the "Ask next" answer message is sent with an address of a slave node, which has the lowest *GP* value.
- $SP_{local} < SP_{remote}$: asker is allowed to join, but as a new master of a group by downgrading this master to a slave. In this case **AskersHandler** invokes a **PushHandler** to execute the pushing routine and suspends itself until **PushHandler** completes its execution and terminates. When **AskersHandler** is resumed, it sends the "Join granted (push)" reply message and creates a **MasterHandler** object to handle a **Connection**.

Behavioral pattern for the **AskersHandler** class is shown in Figure 2.12.

PushHandler

Purpose: **PushHandler** class is derived from the base class ***Thread***. **PushHandler** is responsible for the adaptation to the network operation. The adaptation to the network operation is performed by pushing a local node downwards in the FROST architecture. **PushHandler** could be invoked either by **AskersHandler** or **MasterHandler**. **MasterHandler** invokes **PushHandler** if such command is sent by master of a group, because it is also being pushed.

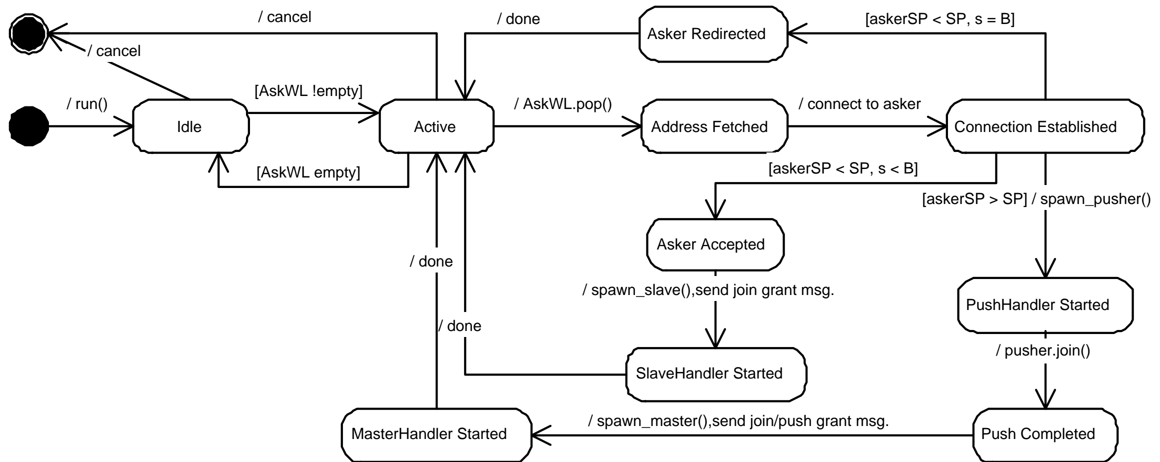


Figure 2.12: State Chart Diagram for the **AskersHandler** *operation.*

Behavior: Behavior of **PushHandler** depends on which handler has invoked it:

- If **PushHandler** was invoked by **AskersHandler**, then the connection with master node has to be closed, but before closing, the master has to grant the permission for the adaptation to the network. After the permission is granted the local information (*SAL, AskWL*) has to be sent to the asker node.
- If **PushHandler** was invoked by **MasterHandler**, then the local information (*SAL, AskWL*) has to be sent to the master.

Behavioral pattern for the **PushHandler** class is shown in Figure 2.13. From here the be-

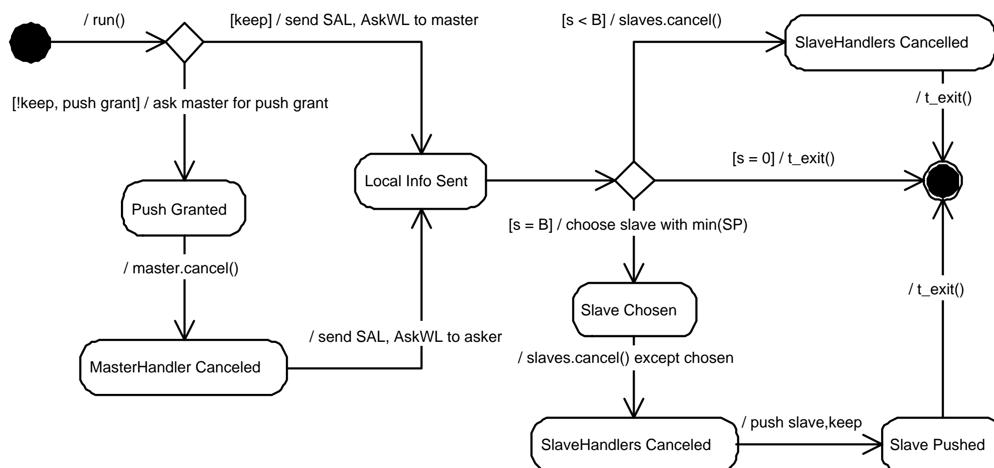


Figure 2.13: State Chart Diagram for the **PushHandler** *operation.* [keep] *guard indicates that* **PushHandler** *was invoked by* **MasterHandler**.

havior is the same for both cases of invocation. Depending on the number of slaves in a group, master has the following options:

$s = 0$**:** In this case, because master is a leaf node there is nothing to be done.

$s < B$**:** In this case all the connections with slaves has to be closed. After the adaptation to the network operation is completed a master will become a leaf node.

$s = B$**:** In this case a group is complete and slave with min(*SP*) has to be chosen to be pushed also. The remaining connections with slaves has to be closed.

LeaveHandler

Purpose: **LeaveHandler** class is derived from the base class ***Thread***. **LeaveHandler** is responsible for handling the voluntary leaving from the FROST network.

Behavior: **LeaveHandler** operation is performed in one of the three modes:

"I leave": This mode indicates that a FROST user has pressed the "Exit" option in a user interface and he wants to exit the FROST network. In this case **LeaveHandler** asks the master of a group for permission to leave. When permission is granted voluntary leave operation can be continued.

"Master leaves": This mode indicates that master of a group is leaving the FROST network and this node is a chosen to be a new master of a group. In this case the connection with leaving master has to be closed and new connection has to be opened to the master of the leaving node.

"Master relocation": This mode indicates that master of a group is leaving a group because it was chosen to be a master of another group and this node is chosen to be a master of this group.

From here the **LeaveHandler** behavior is common for all leave modes. If leaving node is not a leaf node then it has to choose a slave with max(*SP*) to be the new master of a group. The local information (*SAL, AskWL*) has to be sent to chosen slave and remaining connections with slaves has to be closed. Finally, if node is in one of the master leave modes it has to establish the connections with new slave nodes. Behavioral pattern for the **LeaveHandler** class is shown in Figure 2.14.

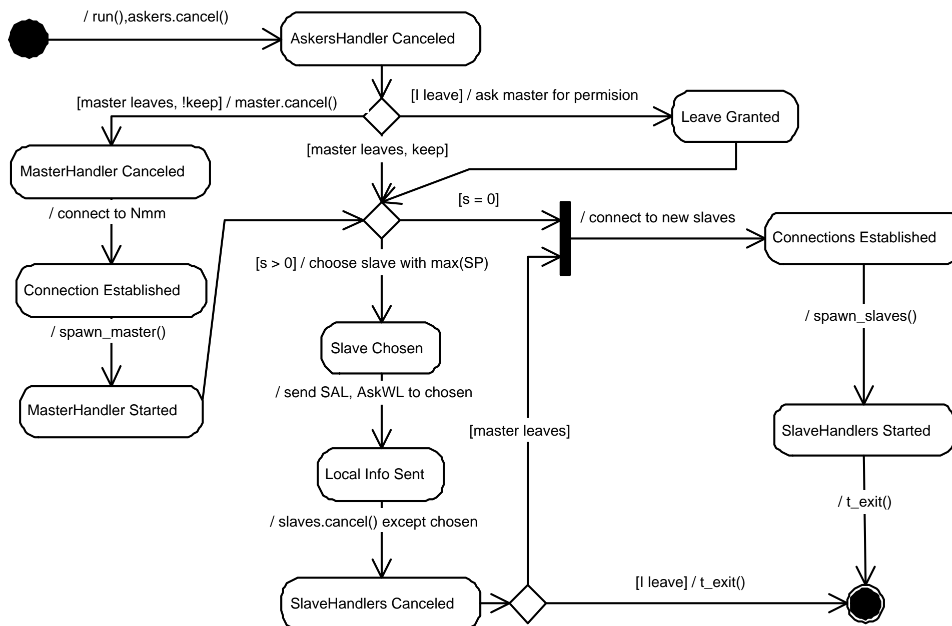


Figure 2.14: State Chart Diagram for the **LeaveHandler** *operation.* [keep] *guard indicates that* **LeaveHandler** *will perform leave operation in* "Master relocation" *mode. Node N_{mm} is master of leaving master.*

FailHandler

Purpose: **FailHandler** class is derived from the base class ***Thread***. **FailHandler** is responsible for handling node failures in the FROST network. Fault tolerance can be divided into two parts: discovery of remote node which failed and recovery. A remote node is considered as failed if the connection between local and remote node has been unexpectedly closed. The *sockets* implementation provides the means of discovering such failures and this feature has been used in the **Connection** class to detect unexpected disconnections. The purpose of the recovery operation is to assure the integrity in the FROST network. The purpose of **FailHandler** class is to recover when remote node fails and it should be invoked either by **MasterHandler** or **SlaveHandler** depending on what kind of node (slave or master) has failed.

The recovery operation has not been implemented yet. The intent is to primarily test and measure the Join/Leave protocol performance assuming that all nodes are reliable and then concentrate on the recovery issue.

MasterHandler

Purpose: **MasterHandler** class is derived from the base class ***Thread***. This class is responsible for maintaining the connection with the master of a group. **MasterHandler** has a **Connection** object assigned to it, which provides an interface to send and receive messages and data.

Behavior: When started, **MasterHandler** enters a message processing loop in which it receives messages from the master of a group. Message types and purpose of each is shown in Table 2.19. Behavioral pattern for the **MasterHandler** class is shown in Figure 2.15.

Message Received	Purpose and Response
PUSH	Master commands to start the pushing routine. **PushHandler** is invoked and when it finishes PUSH_READY and ASK_MY_GP messages are sent.
LEAVE	Master is leaving and this node has been chosen to be the new master of a group. **LeaveHandler** is invoked and when it finishes LEAVE_READY and ASK_MY_GP messages are sent.
PUSH_GRANT	Master grants the permission for pushing. **AskersHandler** is signaled.
LEAVE_GRANT	Master grants the permission for leaving. **LeaveHandler** is signaled.
TAKE_ID	Master sends its ID.
TAKE_SAL	Master sends its *SAL* list.
TAKE_WL	Master sends its AskWL list.
GET_ID	Master asks to send this node ID.
GET_SP	Master asks to send this node SP value.
GET_GP	Master asks to send this node GP value.
GET_STATUS	Master asks if this node is waiting for some permission. This message is received from the master which recently became one.
ASK_AGAIN	Master tells the slave that it should ask again for a permission it waits.
EXPECT	Master sends an address of a new master which will connect soon. Address is added to **Expector** object.
DISCONNECT	Master closes the connection. **MasterHandler** terminates.

Table 2.19: **MasterHandler** *messages.*

SlaveHandler

Purpose: **SlaveHandler** class is derived from the base class ***Thread***. This class is responsible for maintaining the connection with a slave node. **SlaveHandler** has a **Connection** object assigned to it, which provides an interface to send and receive messages and data.

Behavior: When started, **SlaveHandler** enters a message processing loop in which it receives messages from the slave node. Message types and purpose of each is shown in Table 2.20. Behavioral pattern for the **SlaveHandler** class is shown in Figure 2.15.

Message Received	Purpose and Response
PUSH	Slave asks for a permission to start the pushing routine. If *mutex_pl* is not locked then this node locks it and grants the permission to push.
LEAVE	Slave asks for a permission to start the leaving routine. If *mutex_pl* is not locked then this node locks it and grants the permission to leave.
TAKE_ID	Slave sends its ID.
TAKE_SP	Slave sends its SP value.
TAKE_GP	Slave sends its GP value.
TAKE_SAL	Slave sends its *SAL* list.
TAKE_WL	Slave sends its AskWL list.
TAKE_STATUS	Slave sends its status.
GET_ID	Slave asks to send this node ID.
ASK_MY_GP	Slave tells the master to update its *GP* value.
PUSH_READY	Slave has finished push operation. **PushHandler** is signaled.
LEAVE_READY	Slave has finished leave operation. **LeaveHandler** is signaled.
READY	Slave has finished push or leave operation. Unlocks *mutex_pl*.
EXPECT	Slave sends an address of a node which will connect soon. Address is added to **Expector** object.
DISCONNECT	Slave closes the connection. **SlaveHandler** terminates.

Table 2.20: **SlaveHandler** *messages.*

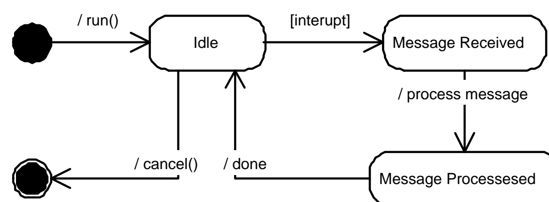


Figure 2.15: State Chart Diagram for the **MasterHandler** *and* **SlaveHandler** *operations.*

2.6 System Interface Component

System interface component implements necessary facilities used to interact with technical platform. Classes ***Thread*** and **Connection** implements an interface to POSIX *threads* and *sockets* respectively. ***Thread*** class is already described in section 2.5.2. The **Connection** class is specified in the following.

2.6.1 Connection Class

Purpose

The purpose of **Connection** class is to provide the means of sending and receiving data between two nodes using TCP/IP protocol stack. Also it contains an information about a remote node.

Attributes

Attribute	Type	Purpose
sock	int	Socket descriptor.
ID	int	Remote node *ID*.
SP	int	Remote node *SP* value.
GP	int	Remote node *GP* value.

Table 2.21: **Connection** *attributes.*

Operations

Operation	Purpose
sendID()	Send local *ID*.
sendSP()	Send local *SP* value.
sendGP()	Send local *GP* value.
send_msg()	Send a message.
sendAddr()	Send an address.
send_list()	Send a list of addresses.
recvID()	Receive remote *ID*.
recvSP()	Receive remote *SP* value.
recvGP()	Receive remote *GP* value.
recv_msg()	Receive a message.
recvAddr()	Receive an address.
recv_list()	Receive a list of nodes.

Table 2.22: **Connection** *operations.*

2.7 User Interface Component

User interface requirements follows from the use case diagram shown in Figure 2.3. A simple user interface on a character-based terminal has been implemented. It prints the list of options on a terminal screen and waits for an input from a user. User has to enter the option number and press *Enter* to execute the task indicated by the option number. The user interface options are shown in Table 2.23.

Option number	Option	Purpose
0	EXIT	Voluntary leave the FROST network.
1	Join	Join the FROST network.
2	Fail	Leave the network unexpectedly.
3	Status	This option is added for the testing purposes. It prints the status of the Join/Leave protocol in the terminal window. Status information example is shown in Figure 2.16.

Table 2.23: Menu options.

ID	SP	GP	MST	SLV	SLV
5	10	16	6	3	N/A

Acceptor	1
QueueHandler	1
VIPHandler	1
JoinHandler	0
AskersHandler	1
PushHandler	0
LeaveHandler	0
FailHandler	0
push_granted	0
leave_granted	0
mutex_locked	0

Figure 2.16: Example of status information. "0" - false/non-existent, "1" - true/active.

Chapter 3

Join/Leave Protocol Testing

To test the functionality and performance of the implemented Join/Leave protocol a number of system tests were conducted. System testing focuses on the complete system, its functional and non-functional requirements, and its target environment. The following system tests were conducted:

Functional testing. Functional testing, also called requirements testing, tests if the system perform as promised by the requirements specification. Functional testing is a black box technique: testing finds differences between the test cases derived from the use case model and the observed system behavior. In systems with complex functional requirements, it is usually not possible to test all use cases for all valid and invalid inputs. Therefore, only the tests that are relevant and have a high probability of uncovering a failure are selected.

Performance testing. Performance testing is used to test if the non-functional requirements are met. Two types of performance tests were conducted:

Stress tests: The purpose of the stress tests is to evaluate the system when stressed to its limits over a short period of time.

Timing tests: The purpose of the timing tests is to validate conformance to behavioral and performance constraints and evaluate if the system is fast enough.

This chapter presents the purpose and specification of the tests and finally the analysis of the test results.

3.1 Equipment

3.1.1 Cluster at Aalborg University

Development and some of the tests were conducted on an SCI cluster of seven homogeneous dual 733 MHz Pentium III Coppermine workstations on Asus CLS motherboards with ServerWorks LE chipset and Linux OS with kernel 2.4.19. The nodes are interconnected by a 100 Mbit Ethernet LAN, connected by a Cisco System Catalyst 3500 Series switch. The nodes are additionally interconnected in a ring topology with Dolphinics D330 SCI adapters. The SCI adapters is mounted on 33 MHz PCI-G4 buses. Each node is equipped with 2 GB memory.

3.1.2 PlanetLab

PlanetLab [5] is an open, globally distributed testbed for developing, deploying and accessing planetary-scale network services. There are currently more than 115 machines at 45 sites worldwide available to support both short-term experiments and long-running network services. Since

the beginning of 2003, more than 70 research projects at top academic institutions including MIT, Stanford, UC Berkeley, Princeton and the University of Washington have used PlanetLab to experiment with such diverse topics as distributed storage, network mapping, peer-to-peer systems, distributed hash tables, and distributed query processing.

PlanetLab creates a unique environment in which to conduct experiments at Internet Scale. The most obvious is that network services deployed on PlanetLab experience all of the behaviors of the real Internet where the only thing predictable is unpredictability (latency, bandwidth, paths taken). A second advantage is that PlanetLab provides a diverse perspective on the Internet in terms of connection properties, network presence, and geographical location. The broad perspective on the Internet enables development and deployment of a new class of services that see the network from many different angles. For example, to date, researchers using PlanetLab have created worldwide Internet mapping software and identified a common cause of router failure.

Each node consists of a Linux-based PC running specially developed virtual machine technology allowing experiments to be conducted independently.

3.1.3 Functional Tests

The purpose of the Join/Leave protocol is to handle node joins and departures in the FROST network. The implication is that a reasonable number of nodes are required to participate in join and/or leave activities (use cases) to test the functionality of the protocol. The functional requirements of the Join/Leave protocol can be tested by verifying the structural relations between the nodes in the resulting FROST network against the predefined test cases. In general, the functional testing can be seen as comparing two networks of nodes where one is formed by the implemented Join/Leave protocol and another is specified in the test case as it is shown in Figure 3.1. Each node

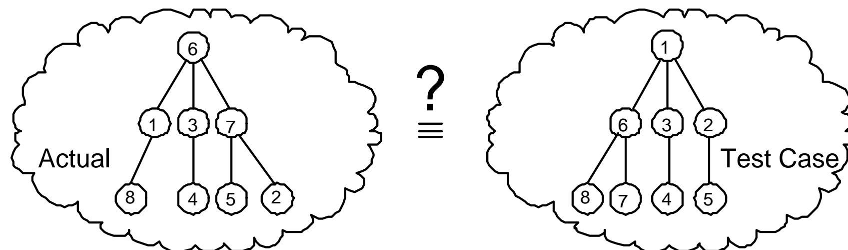


Figure 3.1: Functional Testing.

maintains an *SAL* list that specifies the master/slave relations in the network, and thus the testing problem can be reduced to the problem of comparing *SAL* list of each node in the actual case with corresponding *SAL* list in the test case. However, when the number of nodes in the observed system grows, manual specification of the test cases is not efficient because of the time needed to specify them. For the efficient test case specification an automated test case generator (*TCG*) has to be additionally implemented. The deployment diagram for the functional testing system is shown in Figure 3.2 and is explained in the following.

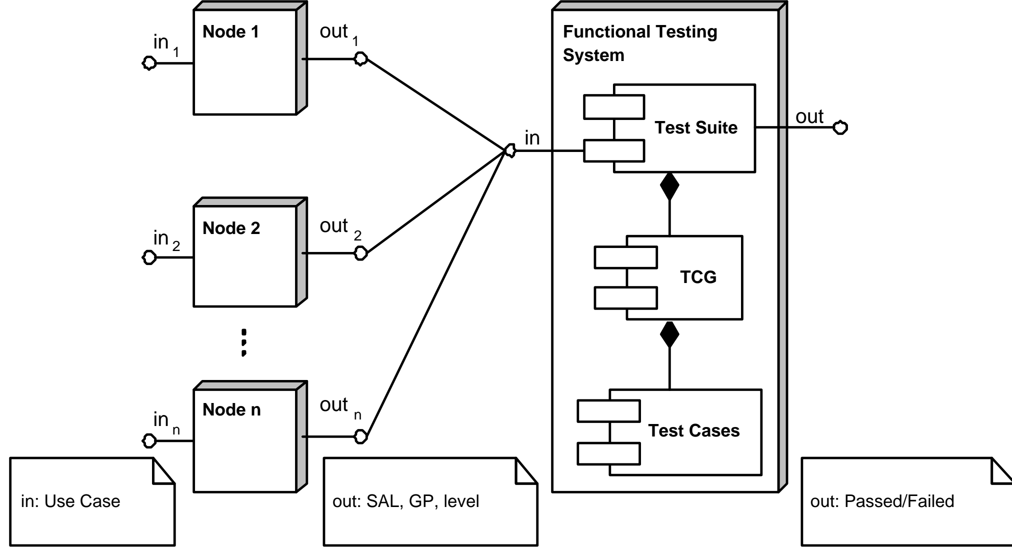


Figure 3.2: Deployment Diagram for the Functional Testing System. TCG - Test Case Generator.

Functional Testing System Specification

To test the functional requirements of the Join/Leave protocol a number of nodes have to be observed by the functional testing system. The observed nodes can be defined as a *n-tuple* of nodes, where the sequence number represents sequential events (join or leave) ordered in time:

$$O = (N_1, N_2, ..., N_i, ..., N_n), \quad \forall 1 \leq i, j \leq n : \forall i < j : time(i) < time(j), \tag{3.1}$$

where N_i is a *pair*: $N_i = (ID_i, SP_i)$.

Each node (see Figure 3.2) given an input data produces an output data, which later is used by the functional testing system. Input and output for all nodes are defined as follows:

Input: The *n-tuple O* from definition 3.1 can be used to generate the *n-tuple IN* of join or leave events ordered in time:

$$IN = (in_1, in_2, ..., in_i, ..., in_n), \quad \forall 1 \leq i, j \leq n : \forall i = j : in_i = e, \tag{3.2}$$

where $e \in \{join, leave\}$ and j is an index in *O*.

Output: An event e at the node input triggers the corresponding behavior of the Join/Leave protocol:

$e = leave$**:** If node leaves the network, then no output is produced, since the node is not a part of the network anymore and that situation will be reflected by the rest of the nodes still in the network.

$e = join$**:** In this case node joins the FROST network and by being a part of the network it provides the join or leave services for the rest of the nodes, if requested. The output information is generated by reading current state of the nodes at some constant time *t* and can be defined as a *n-tuple OUT*:

$$OUT = (out_1, out_2, ..., out_i, ..., out_n), \quad \forall 1 \leq i, j \leq n : t(i) = t(j), \tag{3.3}$$

where out_i is a *3-tuple*: $out_i = (SAL_i, GP_i, level_i)$.

Expected Results

The functional testing system given the output *OUT* has to verify it against the predefined test case OUT_{test}. If $OUT \equiv OUT_{test}$, then the functional requirements for that particular test case are met, otherwise there is an error in the protocol functionality. For the efficient test cases specification the *TCG* component is introduced into the functional testing system. The *TCG* component has the same functional requirements as the implemented Join/Leave protocol, and given the *O* and *IN* specification *TCG* simulates the behavior of Join/Leave protocol and produces the simulated output OUT_{test}, which will be used as a test case for that particular setup.

3.1.4 Stress Tests

The purpose of the stress tests is to evaluate the system when stressed to its limits over a short period of time. During the stress tests, as distinct from the functional tests, the observations will be made on one particular node, which will be stressed to process continuous joining and/or leaving of nodes over a short period of time. During the stress tests the following measurements will be taken:

CPU usage. The CPU usage will be measured using **time** utility, which is used to run programs and summarize system resource usage. The default output of **time** is as follows:

real: elapsed real time in seconds.

user: total number of CPU-seconds that the process spent in user mode.

sys: total number of CPU-seconds that the process spent in kernel mode.

Percentage of the CPU usage is computed as $CPU = \left(\frac{user+sys}{real}\right) \cdot 100\%$.

Bandwidth usage. The bandwidth usage will be measured by capturing and analyzing TCP packets sent and received by the Join/Leave protocol. The packets will be captured using **tcpdump** [7] utility and analyzed by using network protocol analyzer **Ethereal** [8], which is able to provide with various network usage statistics given the captured data.

Throughput. For this case, the measure of throughput is the ratio of nodes processed by the observed node per second. Let the number of nodes processed by the observed node be n, and since the elapsed real time is given by **time** utility, then throughput is computed as follows: $R = \frac{n}{real}, \left[\frac{nodes}{s}\right]$.

Expected Results

The CPU usage is expected to be reasonably low, otherwise the Join/Leave protocol is not usable at all. The FROST system [1] is designated for exploiting wasted CPU-cycles on the client machines and if the maintenance protocol such as Join/Leave protocol will utilize most of the spare CPU-cycles then only a little real work will be done.

The network bandwidth usage is expected to be the mostly consumed resource, however it should be kept as small as possible. From the global point of view, a single user usually has the powerful gaming machine, which probably waste lots of the CPU-cycles, but the connection to the Internet is not necessarily fast. Thus, if the requirements for the network bandwidth will be high, then probably a large number of potential CPU-cycle volunteers will not be able to use the FROST system. The network bandwidth measurements will settle the approximate requirements for it.

Throughput is another important characteristic of the Join/Leave protocol. If the throughput will be lower that the rate of the node joins and/or leaves, then the Join/Leave protocol would become

the bottleneck for the nodes that are waiting to join or leave the network. The estimation of the rate of arrivals and departures in the FROST system is quite a challenging task. However, the rate of arrivals and departures has been explored in other P2P systems:

- The CoopNet [9] P2P system has very similar tree based network topology as the FROST network and it is also used to distribute the bandwidth usage between the peers. The study of the CoopNet network dynamics was made in [10]. The CoopNet system is a video streaming system which was evaluated using the trace of node arrivals and departures gathered at MSNBC [12] on September 11, 2001. The average rate of node arrivals and departures in the 911 trace was 180 per second while the peak rate was about 1000 per second. The authors indicate that one reason for the high rate of change may be that users were discouraged by the degradation in audio/video quality caused by the flash crowd, which caused short lifetime of nodes (i.e. peer disconnects during the streaming session and tries to reconnect again).

- Another measurement study of P2P systems was made in [11]. Two file sharing P2P systems, Napster [17] and Gnutella [15], has been studied to evaluate various characteristics of the P2P networks. One of the characteristics, which has been studied, is the *lifetime* of peers in the system, i.e., how frequently peers connect to the systems, and how long they choose to remain connected. Both, Napster and Gnutella nodes have similar lifetime measures during 12 hours of observation as it is shown in Figure 3.3, which was presented in [11]. The rate of change was not provided by the authors, however, an approximation of rate can be computed using the data given in [11]. The computations are as follows:

  $N = 17125$. N - the number of Gnutella nodes observed (during 12 hours).

  $H = \frac{N}{2} = 8562.5$. Half of nodes N.

  $T_H = 3600[s]$. T_H - median session duration, i.e. approximately H nodes will leave the system after T_H time elapsed (see Figure 3.3).

  $R = \frac{H}{T_H} \approx 2 \left[\frac{nodes}{s}\right]$. R - an approximate rate of change in the Gnutella system (assuming that leaving nodes will be exchanged be the joining nodes). The exact rate could be

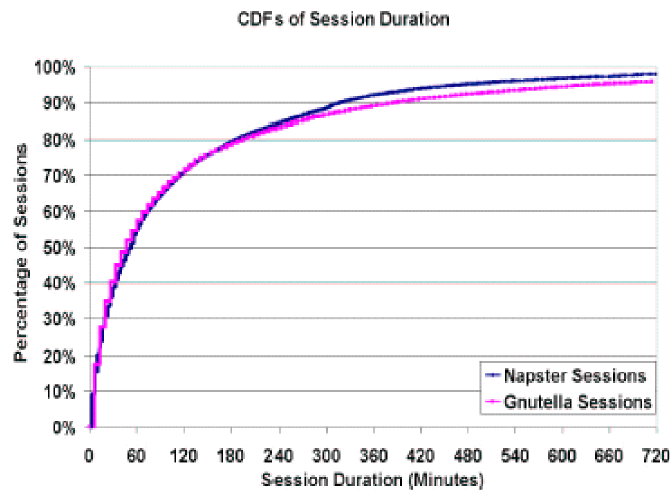


Figure 3.3: The distribution of Napster/Gnutella session durations. CDFs - cumulative distribution functions.

computed by calculating the integral, however an approximate value of R is sufficient for further analysis.

Both studies (CoopNet in [10] and Napster/Gnutella in [11]) gives an insight about the rate of change in a P2P networks. However, before evaluating the rate of change given in both studies, the human factor issues has to be considered also, i.e. what are the reasons of using one or another system and what is the relation between the human factor and the rate of change? Clearly, the more the system is popular the higher is the rate of change. What makes the system being popular is another human factor issue. For instance, the thirst of knowledge during the events of the high or very high importance made the MSNBC [12] news company very popular on September 11, 2001. Another example is sharing of the popular media content (music, video), which makes the P2P system a popular between the clients that are looking for entertainment. In both examples, the clients were motivated to use the system because of their own needs (thirst of knowledge, entertainment, etc.). In the FROST system, however, the users have to be motivated more to volunteer than exploit, which probably will not make the FROST system more popular than the systems presented above. Thus, the expectation is that the rate of change in the FROST network, with high probability, will not exceed an approximate rate of change in the Gnutella network (i.e. $R \approx 2 \left[\frac{nodes}{sec}\right]$).

3.1.5 Timing Tests

The purpose of the timing tests is to validate conformance to behavioral and performance constraints and evaluate if the system is fast enough. During the timing tests the following measurements will be conducted:

T_{join}: the time required to build the FROST network from an arbitrary number of nodes.

T_{leave}: the time required to dismantle the FROST network consisting of an arbitrary number of nodes.

T_{jl}: the time required for the half of the nodes to join the FROST network while the other half is leaving.

The timing measurements described above will be made on several distinct FROST network configurations using different *base* parameters, but with constant number of nodes, to test which configuration performs better (wide or narrow tree).

Another important timing measurement would be to measure the times described above with constant *base* parameter, but with distinct number of nodes to establish a relation between the time and number of nodes. Then it could be possible to estimate the timing for the larger number of nodes without performing actual test.

Expected Results

The results of the timing tests will give an insight on how fast is the system. The results will be evaluated against the rate of change ($R \approx 2$) settled in the specification of stress tests. It is expected that Join/Leave protocol will be able to operate at the rate R_{actual} at least equal to R ($R_{actual} \geq R$). For instance, if 50 nodes were able to join the FROST network per 14 seconds, then the actual rate of change at which protocol is able to operate is $R_{actual} = 3.571 > R$.

3.2 Test Description

3.2.1 Functional Tests

The purpose of the functional tests is to verify if the Join/Leave protocol operates as promised by the functional requirements. Functional tests were conducted on the global overlay network - PlanetLab [5] (see PlanetLab specification on page 34). An observation of availability was made to select the nodes that are almost always available. 92 nodes were selected to conduct the functional tests with Join/Leave protocol.

Functional Test 1

The purpose of this test is to verify if the protocol correctly builds the FROST network while stressing the high level of adaptation in the network.

Test Data :

Base: Base parameter $B = 3$ is selected, because it will result in a reasonably complex network structure with common functional characteristics for networks with $B \geq 3$. The minimum value for base parameter is $B = 2$, however it was not selected because it will not reflect all the functional characteristics of the Join/Leave protocol. For example, if $B = 2$ and node is leaving, then a possible choice space has a multiplicity of $1 - 0..1$, which means that one slave will be selected as a replacement, and at most one slave will stay at its position. Whereas the networks with $B \geq 3$ have a choice space with multiplicity of $1 - 0..*$ (one to many), i.e. if one slave is selected as a replacement, then 1 or more slaves will stay at their positions. Thus the implication is that networks with $B \geq 3$ have common functional characteristics. The complexity of network depends on the depth of the network *tree*. The larger is depth the more complex is network, and that is because join or leave inflicts updating in the network structure down to the bottom of the tree. Clearly, reducing the base parameter will increase *tree* depth for a constant number of nodes and vice versa: $\left(\lim_{B \to \infty} depth = 1\right)$. In general case, for any base parameter the time complexity for join and leave operations is $O(\log N)$, where N is the number of nodes in the network.

SP: Node static performance parameter is a key parameter when building or dismantling the network. To stress the high level of adaptation in the network, *SP* values have to be properly assigned to the nodes. If observed nodes *O* is defined as:

$$O = (N_1, N_2, ..., N_i, ..., N_n),\ \forall 1 \leq i, j \leq n : \forall i < j : time(i) < time(j), \quad (3.4)$$

where N_i is a *pair*: $N_i = (ID_i, SP_i)$, then:

$$\forall 1 \leq i \leq 3 : SP_i = i + 200 : ID_i = i, \quad (3.5)$$

$$\forall 4 \leq i \leq n : SP_i = ID_i = i, \quad (3.6)$$

where n is the number of nodes. 3.5 guarantees that nodes N_1 to N_3 will keep their position at the highest level during the tests. 3.6 guarantees that a joining node whose $ID \geq 13$ will trigger the adaptation to the network procedure (see Fig. 3.4 (a)).

Test Procedure : The test was performed in the following steps:

1. Start node N_0 on PlanetLab machine which is known for joining nodes.
2. Run a script, which spawns nodes N_1 to N_{92}. Nodes are forced to join following an order in time (see Fig. 3.4 (b)) to avoid concurrent node joins, otherwise the resulting network would be non-deterministic. Consequently it wouldn't be possible to test the actual network against the test cases generated by the test case generator.
3. When network is built, a signal has to be sent to all nodes forcing to write their status information to the files (e.g. **ssh user@planetlab1.diku.dk "less -F pid | xargs -i kill -USR1 {}"**). Each node writes its *pid* (process id) to **"pid"** file when started. A signal handler which is able to catch and process **USR1** signal has been implemented into the Join/Leave protocol. When **USR1** signal is caught the status information is written to two files:

   ID+".out" (e.g. "1.out") : file contains the data (*SAL*, *GP*, *level*, etc.), which will be used by the functional testing system.

   hostname+".out" (e.g. "planetlab1.diku.dk.out") : file contains human readable detailed status information, which can be used to find a cause if system doesn't work as expected.
4. Stop node N_0.
5. Fetch status information (***.out**) from nodes used in test.
   (e.g. **sftp user@planetlab1.diku.dk:*.out**).
6. Run the functional testing system to verify if actual network OUT_{actual} is equivalent to the network OUT_{test}, which is generated by the test case generator. Two outcomes are possible:
   (a) If $OUT_{actual} \neq OUT_{test}$, then find a cause in **hostname+".out"** files, correct an implementation mistake and start from step 1 again.
   (b) If $OUT_{actual} \equiv OUT_{test}$, then move to Functional Test 2 to dismantle the network.

Results : During the test some mistakes, which are not related to the design of the protocol, were found and repaired. A final result of this test is shown in Appendix A.1 and B.1.

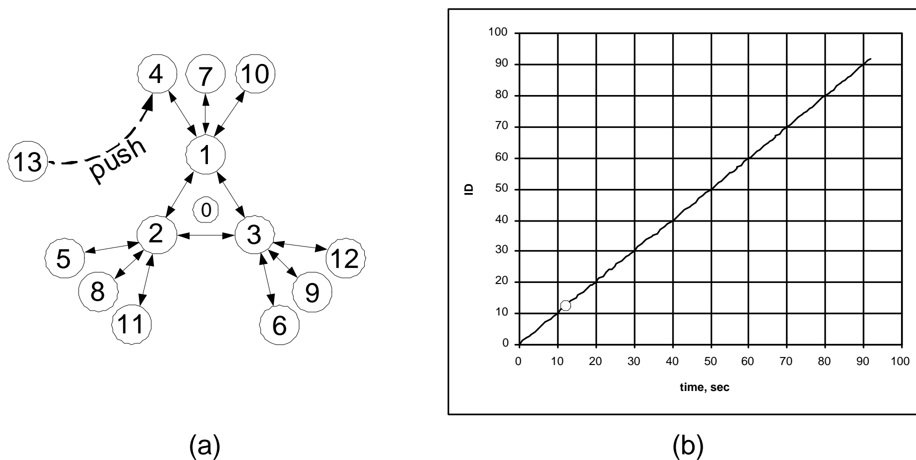


Figure 3.4: (a) - network snapshot at the time (circle mark in (b)) when node N_{13} is joining. (b) - time diagram of node joins.

Functional Test 2

The purpose of this test is to verify if the protocol correctly dismantles the FROST network while stressing the high level of of adaptation in the network.

Test Data : For this functional test the FROST network, which was build during the Functional Test 1, will be used as a test data.

Test Procedure : The test was performed in the following steps:

1. Run a script, which sends the termination signal to nodes N_{13} to N_{92} following an order in time as it is shown in time diagram in Figure 3.5 (b). (e.g. **ssh user@planetlab1.diku.dk "less -F pid | xargs -i kill -INT {}"**). When **INT** signal is caught a node performs voluntary leave procedure and leaves the FROST network.
2. After nodes N_{13} to N_{92} have left , a signal has to be sent to remaining nodes forcing to write their status information to the files (e.g. **ssh user@planetlab1.diku.dk "less -F pid | xargs -i kill -USR1 {}"**). When **USR1** signal is caught the status information is written to two files:

   ID+".out" (e.g. "1.out") : file contains the data (*SAL*, *GP*, *level*, etc.), which will be used by the functional testing system.

   hostname+".out" (e.g. "planetlab1.diku.dk.out") : file contains human readable detailed status information, which can be used to find a cause if system doesn't work as expected.
3. Fetch status information (***.out**) from nodes used in test.
   (e.g. **sftp user@planetlab1.diku.dk:*.out**).
4. Send termination signal to the remaining nodes.
5. Run the functional testing system to verify if actual network OUT_{actual} is equivalent to the network OUT_{test}, which is generated by the test case generator. If $OUT_{actual} \equiv OUT_{test}$, then actual network and generated network should be the same as in Figure 3.4 (a).

Results : During the test some mistakes, which are not related to the design of the protocol, were found and repaired. A final result of this test is shown in Appendix A.2 and B.2.

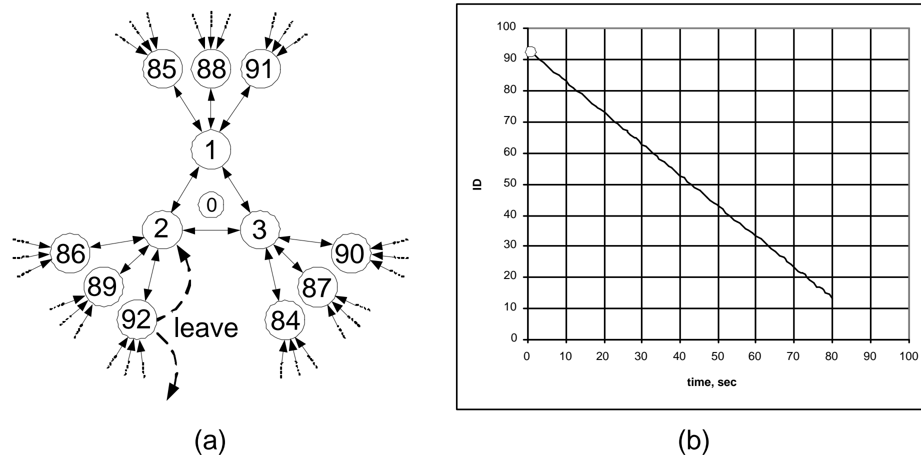


Figure 3.5: (a) - network snapshot at the time (circle mark in (b)) when node N_{92} is leaving. (b) - time diagram of node leaves.

3.2.2 Stress Tests

Stress tests were conducted on a cluster of 7 nodes at Aalborg University (see cluster specification on page 34).

Stress Test 1

The purpose of this test is to measure a throughput - how many nodes per second an observed node can handle, CPU usage and bandwidth usage when an observed node is stressed to process an arbitrary number of *pure* node joins. Node join is called *pure* if node after joining an observed node can leave without involving special processing at observed node, i.e. when joined node decides to leave, it just fails (all threads are canceled) and can start joining procedure again. This way of leaving is appropriately handled by an observed node, since it can detect failed node and delete all information related to that node, thus other nodes can join a vacant place. The purpose of such test with *pure* node joins is to measure the capabilities of an observed node (probably the one at the highest levels) which operates as a guide for the joining nodes, i.e. gives a direction where to join or accepts joining node.

Test Data : All 7 nodes are used for the stress test (see Fig. 3.6), node N_1 is an observed node, nodes N_2 to N_7 are used to concurrently perform an arbitrary number of *pure* node joins to node N_1. Two test cases were used for testing:

Test Case 1: nodes N_2 to N_7 concurrently performs 50 *pure* joins each. Node N_1 will process 300 *pure* joins in total.

Test Case 2: nodes N_2 to N_7 concurrently performs 100 *pure* joins each. Node N_1 will process 600 *pure* joins in total.

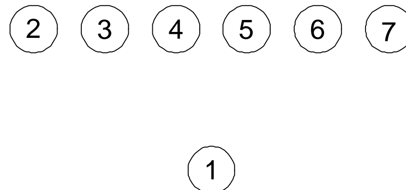


Figure 3.6: Nodes used for stress test 1. N_1 - observed node. N_2 to N_7 - nodes used to stress node N_1.

Test Procedure : For each test case the procedure is performed in the following steps:

1. Start node N_1 with **time** command to measure CPU usage (e.g. **time ./observednode**).
2. On the machine where node N_1 is running, start **tcpdump** to capture and write to a file all incoming and outcoming packets on a well known port used by Join/Leave protocol (e.g. **tcpdump -w filename.out port 60606**).
3. Run the script which spawns the nodes N_2 to N_7 (e.g. **more hosts | xargs -ti ssh -fn ”cd client; ./frostclient”**, where **hosts** is a file which contains hostnames of machines 2 to 7 and **-fn** option tells **ssh** to go into background just before the execution of command string, then **ssh** will return immediately after spawning a process on a remote machine). A situation where the nodes start to operate is shown in Figure 3.7 (a) and at any given moment during the execution of the test case the situation can be similar to one shown in Figure 3.7 (b).

4. Each node writes its *pid* (process id) to **hostname+"_pid"** (**"sister1_pid"**, **"sister2_pid"**, etc.) file when started. After the test case is completed the termination signal has to be sent to all nodes including the observed node N_1 (**rsh sister1 "cd client; less -F sister1_pid | xargs -ti kill -INT {}; rm -f sister1_pid"**, etc.). A signal handler which can catch and process the signals from an OS has been implemented into the Join/Leave protocol and it can provide with any required information about the state of the protocol just before terminating. After this step all the nodes are terminated (see Fig. 3.7 (c)) and the results can be collected and analyzed (see Tables 3.1 and 3.2).

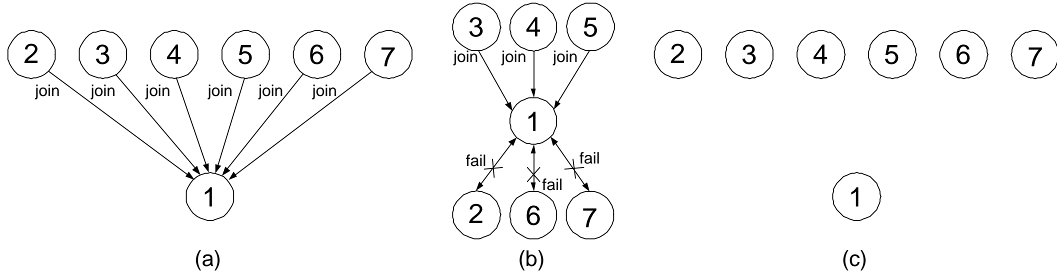


Figure 3.7: (a) - network snapshot when the stress test 1 was started; (b) - network snapshot during the execution of the stress test 1; (c) network snapshot when the stress test 1 was completed.

Test Results :

B = 6	Time			CPU usage			
#	time, s	Δtime, s	nodes/s	user, s	sys, s	CPU, %	$\Delta CPU_{\#}$, %
300	2.23	2.17	138.12	0.05	0.42	21.12	14.20
	2.12			0.03	0.24	12.72	
	2.14			0.01	0.18	8.76	
600	4.41	4.25	141.34	0.13	0.64	17.45	14.07
	4.18			0.18	0.35	12.69	
	4.14			0.02	0.48	12.07	
		Δ nodes/s:	139.73			ΔCPU, %:	14.13

Table 3.1: Measurements of pure join rate (nodes/s) and CPU usage at node N_1. # - is the number of joins processed by node N_1. B - is the base of the FROST architecture.

B = 6		Bandwidth usage		
#	Δtime, s	kbytes of traffic	traffic, Mbits/s	$\Delta traffic_{\#}$, Mbits/s
300	2.17	569.19	2.10	2.15
		568.62	2.20	
		569.64	2.15	
600	4.25	1133.44	2.24	2.24
		1139.42	2.23	
		1138.24	2.25	
			Δ traffic, Mbits/s :	2.19

Table 3.2: Measurements of bandwidth usage at node N_1. # - is the number of pure node joins processed by node N_1. B - is the base of the FROST architecture.

Stress Test 2

The purpose of this test is to measure a throughput - how many nodes per second an observed node can handle, CPU usage and bandwidth usage when an observed node is stressed to process an arbitrary number of concurrent node joins and voluntary leaves.

Test Data : All 7 nodes are used for the stress test (see Fig. 3.6), node N_1 is an observed node, nodes N_2 to N_7 are used to concurrently perform an arbitrary number of node joins followed by voluntary leaves. Two test cases were used for testing:

Test Case 1: nodes N_2 to N_7 concurrently performs 50 joins and 50 voluntary leaves each. Node N_1 will process 300 joins and 300 voluntary leaves in total.

Test Case 2: nodes N_2 to N_7 concurrently performs 100 joins and 100 voluntary leaves each. Node N_1 will process 600 and 600 voluntary leaves in total.

Test Procedure : The test procedure is the same as presented for the stress test 1 on page 43, except that nodes are not failing after each join but leaving according to voluntary leave procedure. The results are shown in Tables 3.3 and 3.4

Test Results :

B = 6	Time			CPU usage			
$join_{\#}/leave_{\#}$	time, s	Δtime, s	nodes/s	user, s	sys, s	CPU, %	$\Delta CPU_{\#}$, %
300/300	6.36 6.24 6.67	6.42	93.41	0.00 0.02 0.16	0.07 0.03 0.50	1.10 0.80 9.89	3.93
600/600	11.23 12.70 13.01	12.31	97.45	0.04 0.04 0.02	0.18 0.11 0.11	1.96 1.18 1.00	1.38
		Δ nodes/s:	95.43			ΔCPU, %:	2.66

Table 3.3: Measurements of rate (nodes/s) and CPU usage at node N_1. $join_{\#}$ - is the number of joins processed by node N_1. $leave_{\#}$ - is the number of voluntary leave processed by node N_1. B - is the base of the FROST architecture.

B = 6		Bandwidth usage		
$join_{\#}/leave_{\#}$	Δtime, s	kbytes of traffic	traffic, Mbits/s	$\Delta traffic_{\#}$, Mbits/s
300/300	6.42	729.05 725.85 725.44	0.94 0.95 0.89	0.93
600/600	12.31	1427.23 1454.80 1460.90	1.04 0.94 0.92	0.97
			Δ traffic, Mbits/s :	0.95

Table 3.4: Measurements of bandwidth usage at node N_1. $join_{\#}$ - is the number of joins processed by node N_1. $leave_{\#}$ - is the number of voluntary leave processed by node N_1. B - is the base of the FROST architecture.

Stress Test 3

The first two stress tests gives a result where the rate of change is noticeably higher than expected ($R_{actual} \approx 100 \gg R_{expected} \approx 2$), whereas bandwidth usage is very high ($\approx$ 1 Mbit/s), which implies that the bandwidth is the most consumed resource as expected. The purpose of this test is to measure bandwidth usage when an observed node is stressed to process an arbitrary number of node joins and voluntary leaves at the rate of change close to $R_{expected}$.

Test Data : All 7 nodes are used for the stress test (see Fig. 3.6), node N_1 is an observed node. To reduce the rate of change in an observed network of nodes only one node (N_7) will perform an arbitrary number of node joins followed by voluntary leaves. Nodes N_2 to N_6 joins the network once and participates in the network activities. The following test cases were used for testing:

Test Case 1: FROST architecture base $B = 6$. Two different scenarios were tested:

1. Node N_7 performs 100 joins and 100 voluntary leaves.
2. Node N_7 performs 200 joins and 200 voluntary leaves.

Test Case 2: FROST architecture base $B = 3$. Two different scenarios were tested:

1. Node N_7 performs 100 joins and 100 voluntary leaves.
2. Node N_7 performs 200 joins and 200 voluntary leaves.

Test Case 3: FROST architecture base $B = 2$. Two different scenarios were tested:

1. Node N_7 performs 100 joins and 100 voluntary leaves.
2. Node N_7 performs 200 joins and 200 voluntary leaves.

Test Procedure : The test procedure is the same as presented for the stress test 1 on page 43. The snapshots of the network during each test case is shown in Table 3.5. The results are shown in Tables 3.6, 3.7, 3.8, 3.9, 3.10 and 3.11.

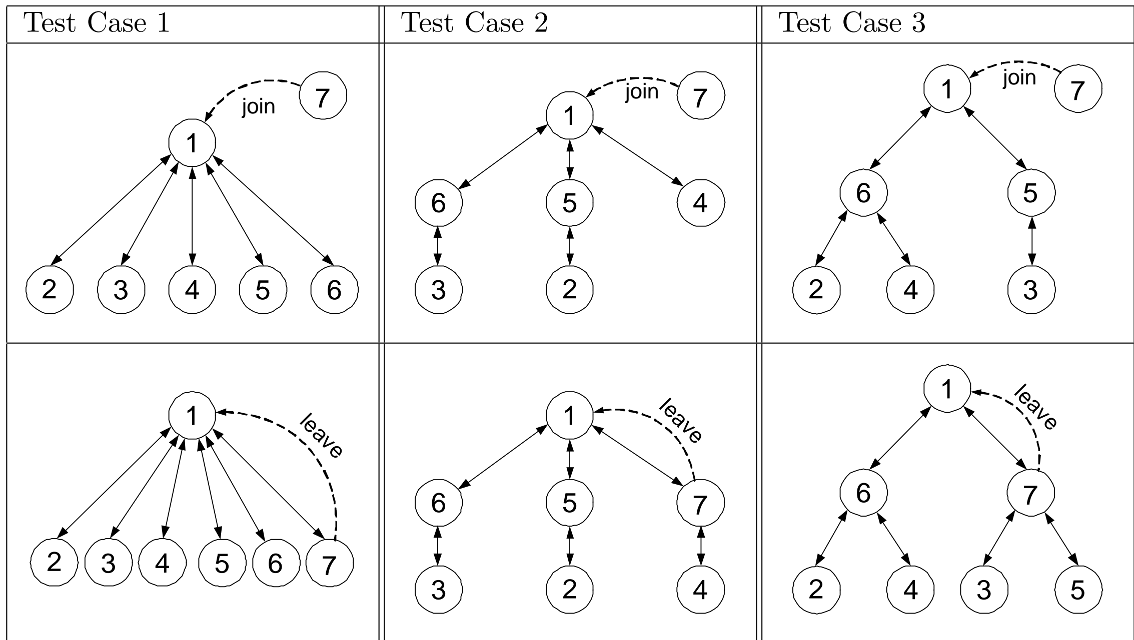


Table 3.5: Test cases for stress test 3. N_1 - observed node. N_2 to N_6 - nodes are participants in the network. N_7 is used to stress node N_1.

Test Results :

B = 6	Time			CPU usage			
$join_{\#}/leave_{\#}$	time, s	Δtime, s	nodes/s	user, s	sys, s	CPU, %	$\Delta CPU_{\#}$, %
100/100	21.89 17.50 16.81	18.73	10.68	0.02 0.01 0.06	0.02 0.02 0.02	0.18 0.17 0.48	0.28
200/200	28.31 26.65 28.48	27.81	14.38	0.10 0.08 0.02	0.24 0.60 0.01	1.20 2.55 0.11	1.29
		Δ nodes/s:	12.53			ΔCPU, %:	0.78

Table 3.6: ***Test Case 1****. Measurements of rate (nodes/s) and CPU usage at node N_1.* $join_{\#}$ *- is the number of joins processed by node N_1.* $leave_{\#}$ *- is the number of voluntary leave processed by node N_1.*

B = 6		Bandwidth usage		
$join_{\#}/leave_{\#}$	Δtime, s	kbytes of traffic	traffic, kbits/s	$\Delta traffic_{\#}$, kbits/s
100/100	18.73	237.43 238.53 238.27	91.14 114.69 118.78	108.20
200/200	27.81	476.37 477.60 477.08	141.31 150.53 140.29	144.04
			Δ traffic, kbits/s :	126.12

Table 3.7: ***Test Case 1****. Measurements of bandwidth usage at node N_1.* $join_{\#}$ *- is the number of joins processed by node N_1.* $leave_{\#}$ *- is the number of voluntary leave processed by node N_1.*

B = 3	Time			CPU usage			
$join_{\#}/leave_{\#}$	time, s	Δtime, s	nodes/s	user, s	sys, s	CPU, %	$\Delta CPU_{\#}$, %
100/100	27.09 24.40 25.55	25.68	7.79	0.01 0.03 0.08	0.02 0.01 0.21	0.11 0.16 1.14	0.47
200/200	49.71 41.78 41.82	44.44	9.00	0.13 0.15 0.15	0.61 0.46 0.40	1.49 1.46 1.32	1.42
		Δ nodes/s:	8.40			ΔCPU, %:	0.95

Table 3.8: ***Test Case 2****. Measurements of rate (nodes/s) and CPU usage at node N_1.* $join_{\#}$ *- is the number of joins processed by node N_1.* $leave_{\#}$ *- is the number of voluntary leave processed by node N_1.*

B = 3		Bandwidth usage		
$join_{\#}/leave_{\#}$	Δtime, s	kbytes of traffic	traffic, kbits/s	$\Delta traffic_{\#}$, kbits/s
100/100	25.68	418.90 418.66 418.84	130.05 144.38 137.22	137.22
200/200	44.44	849.48 851.49 852.25	143.36 171.01 171.01	161.79
			Δ traffic, kbits/s :	149.50

Table 3.9: ***Test Case 2****. Measurements of bandwidth usage at node N_1.* $join_{\#}$ *- is the number of joins processed by node N_1.* $leave_{\#}$ *- is the number of voluntary leave processed by node N_1.*

B = 2	Time			CPU usage			
$join_{\#}/leave_{\#}$	time, s	Δtime, s	nodes/s	user, s	sys, s	CPU, %	$\Delta CPU_{\#}$, %
100/100	26.69 21.31 24.51	24.17	8.28	0.04 0.13 0.03	0.60 0.62 0.09	2.40 3.52 0.49	2.14
200/200	49.71 41.78 41.82	46.16	8.67	0.06 0.08 0.07	0.27 0.45 0.32	0.79 1.28 0.70	0.93
		Δ nodes/s:	8.47			ΔCPU, %:	1.53

Table 3.10: ***Test Case 3****. Measurements of rate (nodes/s) and CPU usage at node N_1.* $join_{\#}$ *- is the number of joins processed by node N_1.* $leave_{\#}$ *- is the number of voluntary leave processed by node N_1.*

B = 2		Bandwidth usage		
$join_{\#}/leave_{\#}$	Δtime, s	kbytes of traffic	traffic, kbits/s	$\Delta traffic_{\#}$, kbits/s
100/100	24.17	418.51 417.38 403.11	131.07 163.84 138.24	144.38
200/200	46.16	848.36 847.42 921.59	169.98 172.03 139.26	160.43
			Δ traffic, kbits/s :	152.41

Table 3.11: ***Test Case 3****. Measurements of bandwidth usage at node N_1.* $join_{\#}$ *- is the number of joins processed by node N_1.* $leave_{\#}$ *- is the number of voluntary leave processed by node N_1.*

3.2.3 Timing Tests

The purpose of timing tests is to evaluate if the system is fast enough, while assuring that it performs according functional requirements.

Timing Test 1

The purpose of this test is to measure the time T_{join} required to build the FROST network from an arbitrary number of nodes and the time T_{leave} required to dismantle the FROST network consisting of an arbitrary number of nodes.

The Join/Leave protocol implementation has been modified to automatically gather the following time statistics:

1. *start*, *end* and *duration* of join procedure. Data is gathered in the following steps:
   (a) capture *start* time: **time(&start)**;
   (b) run join procedure: **join()**;
   (c) capture *end* time: **time(&end)**;
   (d) calculate *duration*: **duration=difftime(end, start)**;
   (e) write *start*, *end* and *duration* times to the file **ID+"_endjoin.out"** (e.g. **"1_endjoin.out"**):
   fprintf(file,"started: %s", asctime(localtime(&start)));
   fprintf(file,"finished: %s", asctime(localtime(&end)));
   fprintf(file,"elapsed: %d", duration);
2. *start*, *end* and *duration* of leave procedure. Data is gathered in the following steps:
   (a) capture *start* time: **time(&start)**;
   (b) run leave procedure: **leave()**;
   (c) capture *end* time: **time(&end)**;
   (d) calculate *duration*: **duration=difftime(end, start)**;
   (e) write *start*, *end* and *duration* times to the file **ID+"_endleave.out"** (e.g. **"1_endleave.out"**):
   fprintf(file,"started: %s", asctime(localtime(&start)));
   fprintf(file,"finished: %s", asctime(localtime(&end)));
   fprintf(file,"elapsed: %d", duration);

Test Data: Two test cases were used for testing:

Test Case 1: Build the network using 40 randomly selected nodes deployed at various locations in the world. Base $B = 3$.

Test Case 2: Use the network from Test Case 1 to dismantle it.

Test Procedure. Test Case 1: The test was performed in the following steps:

1. Start N_0
2. Spawn all 40 nodes in parallel to stress concurrent joins. (e.g. **ssh -fn user@planetlab1.diku.dk "cd client; ./frostclient" &**).
3. When the network is built, collect the results (e.g. **sftp user@planetlab1.diku.dk:*.out**). See Fig. 3.8 (a) for results.
4. Stop N_0

Test Procedure. Test Case 2: The test was performed in the following steps:

1. Signal the nodes to start leaving the network. (e.g. **ssh user@planetlab1.diku.dk "less -F pid | xargs -i kill -INT {} &"**)
2. When network is dismantled, collect the results. (e.g. **sftp user@planetlab1.diku.dk:*.out**). See Fig. 3.8 (b) for results.

Test Results :

Test Case 1: The network was built in about 20 seconds - $T_{join} \approx 20\,[sec]$ and the join rate is $R_{join} \approx 2\left[\frac{nodes}{sec}\right]$. Two nodes ($N_{17}$ and N_{30}) have failed to join, and that is because the discovery server N_0 for unknown reasons refused to establish a connection with them.

Test Case 2: The network was dismantled in about 8 seconds - $T_{leave} \approx 8\,[sec]$ and the leave rate is $R_{leave} \approx 5\left[\frac{nodes}{sec}\right]$.

This test was repeated several times and similar failures occurred, when one or few nodes were not able to contact node N_0. Also in some cases few unexpected connection losses were experienced while a node was performing a join or leave procedure causing some nodes to deadlock.

Because of different bandwidth load on PlanetLab at different day time the measured times T_{join} and T_{leave} may change, because of delays in the network.

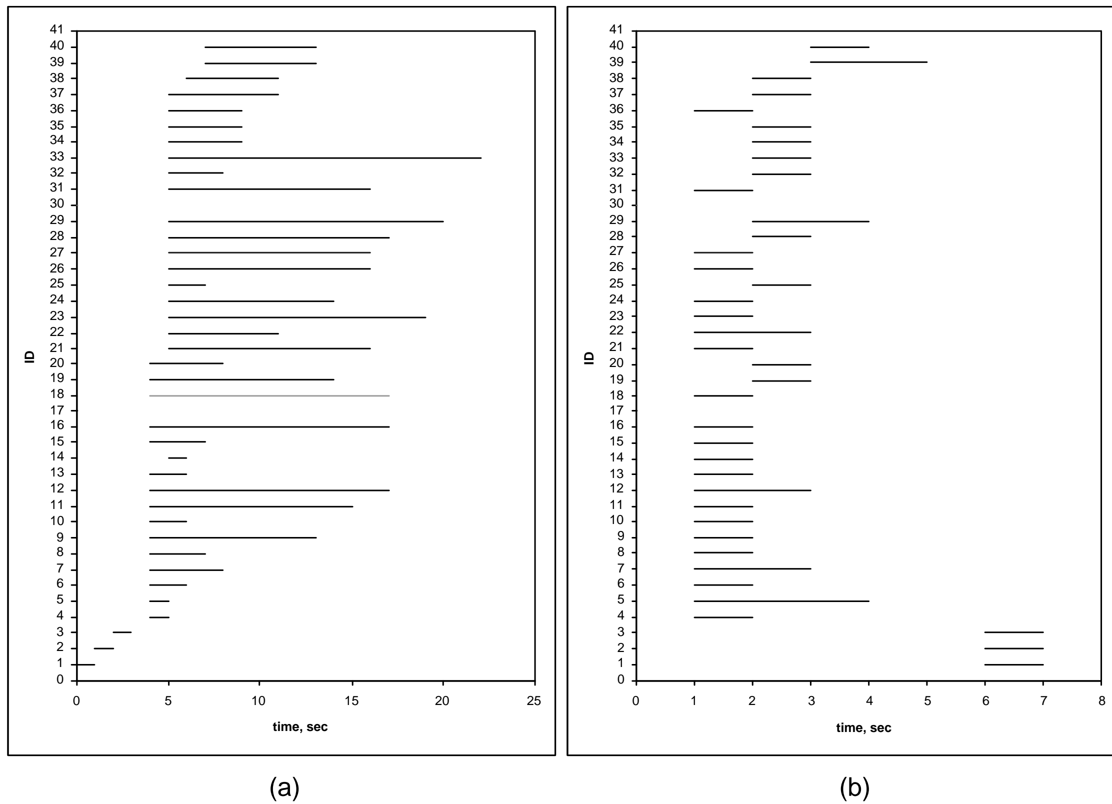


Figure 3.8: (a) - time diagram of joins. (b) - time diagram of departures. A horizontal line in diagram represents time duration of join or leave operation. Error of measurement: ± 1 second.

Timing Test 2

The purpose of this test is to measure the times T_{join} and T_{leave} as described in Timing Test 1, but with distinct FROST network configurations using different *base* parameters and with constant number of nodes, to test which configuration performs better (wide or narrow tree). Also an important timing measurement would be to measure the times described above with constant *base* parameter, but with distinct number of nodes to establish a relation between the time and number of nodes. Then it could be possible to estimate the timing for the larger number of nodes without performing actual test.

Test Data: The test data is summarized in Table 3.12.

	Base		
#	3	6	9
20	Test Case 1	Test Case 4	Test Case 7
40	Test Case 2	Test Case 5	Test Case 8
60	Test Case 3	Test Case 6	Test Case 9

Table 3.12: ***Timing Test 2.*** *Test data. # - is the number of nodes used in a test case.*

Test Procedure: For each test case the times T_{join} and T_{leave} will be measured following the procedure described in Timing Test 1.

Test Results : The results are summarized in Table 3.13 and Figure 3.9.

	Base					
	3		6		9	
#	T_{join}	T_{leave}	T_{join}	T_{leave}	T_{join}	T_{leave}
20	9	3	5	1	6	2
40	20	8	12	5	13	5
60	34	10	22	9	20	9

Table 3.13: ***Timing Test 2.*** *Test results. # - is the number of nodes used in a test case. Timing results are presented in seconds. Error of measurement: ± 1 second.*

Timing Test 3

The purpose of this test is to measure the time T_{jl} - required for the half of the nodes to join the FROST network while the other half is leaving.

Test Data: 40 nodes, which were used in Timing Test 1, will be used to build the network and leave it while other 40 nodes will join the network.

Test Procedure: The test was performed in the following steps:

1. Build the network of 40 nodes as described in Timing Test 1, Test Case 1.
2. When built, dismantle the network as described in Timing Test 1, Test Case 2 and at the same time start to build the network from other 40 nodes as described in Timing Test 1, Test Case 1.
3. When network is built, collect the results and terminate all nodes.

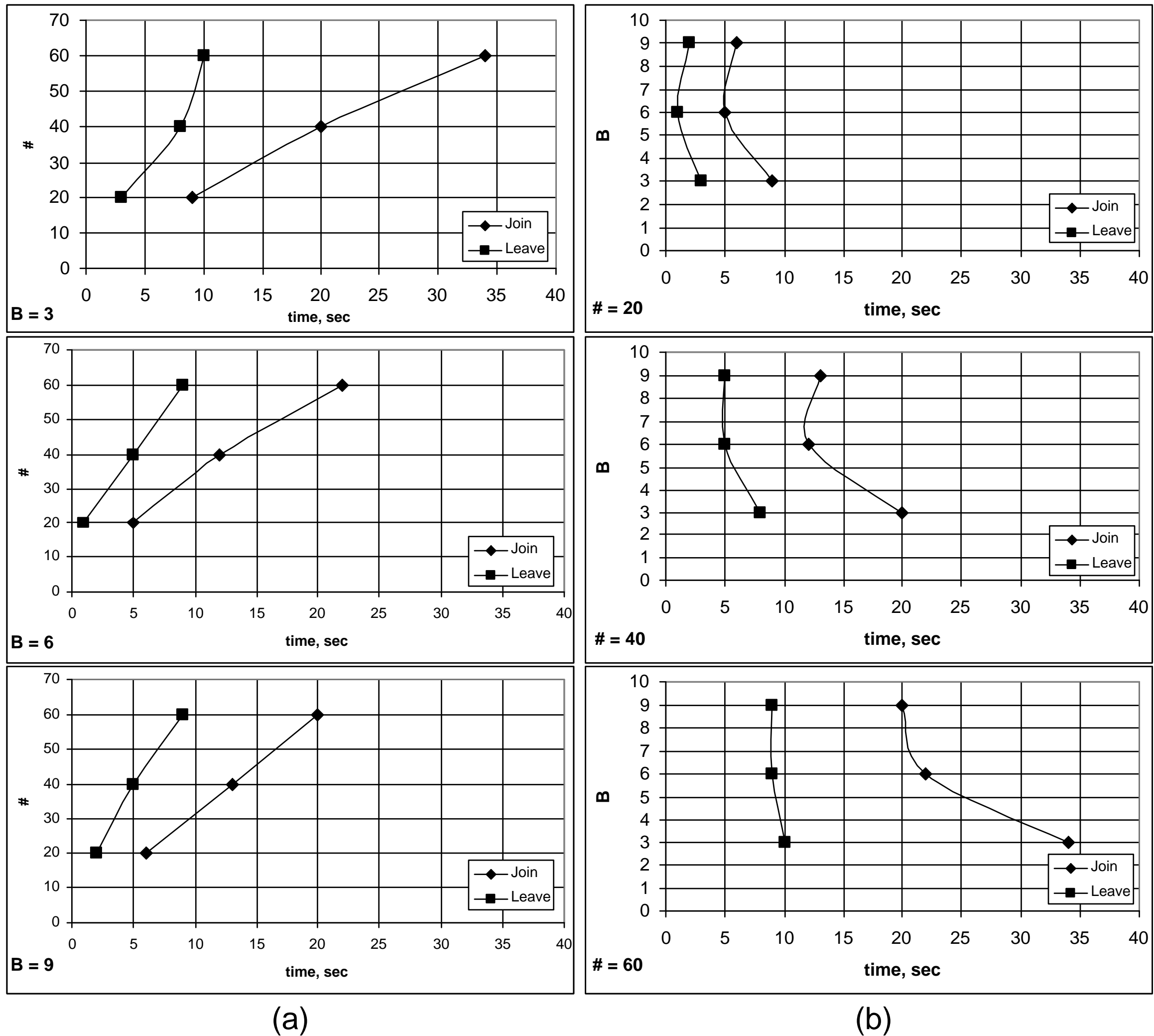


Figure 3.9: (a) - the results of the Timing Test 2 with the constant base parameter. (b) - the results of the Timing Test 2 with the constant number of nodes. # - is the number of nodes used in the test.

Test Results : The test was not able to complete, because of unexpected connection losses between the nodes, causing some nodes to deadlock. Fault tolerance was not implemented in the protocol assuming that all nodes will be reliable and no connection losses will happen. The assumption was made with intent to primarily test and measure the Join/Leave protocol performance with reliable nodes and then concentrate on the fault tolerance.

Some deadlock situations that happened during the tests are listed below:

- When master node grants an asking node a permission to leave or push, master has to wait until the replacement will connect. If node, which was chosen as a replacement fails or node, which was granted for an operation fails then master node will wait for a replacement forever.
- During leave or push operation a number of nodes are involved in exchanging data and instructions. If, however, at least one connection is lost then some nodes will wait for the events that will never happen, causing the system to deadlock.

3.3 Test Results Analysis

3.3.1 Functional Tests

The main corollary of the functional test results is that the Join/Leave protocol correctly builds and dismantles the network, unless the connection losses during the join or leave operation has happened. The results that are presented in appendix A.1 and A.2 where collected when no connection losses occurred and no nodes have failed. However, there were some cases when connection failures occurred and system deadlocked.

3.3.2 Stress Tests

CPU usage: As expected the CPU usage is reasonably low - $CPU \approx 15\%$, even when stressed to operate at rate $R \approx 140 \left[\frac{nodes}{sec}\right]$ (see Table 3.1). To evaluate and measure the CPU and bandwidth usage at lower join and leave rates a special test was conducted. Three test cases were used in testing, and when operating at rate $R \approx 10$ the CPU usage is $CPU \approx 1\%$ and that is a desired result. (see Tables 3.6, 3.8 and 3.10).

Bandwidth usage: As expected the bandwidth usage is the most consumed resource. When stressed to operate at rate $R \approx 120$ the bandwidth consumption was quite large and unsuitable (*traffic* $\approx 1.5 \left[\frac{Mbit}{sec}\right]$, see Tables 3.2 and 3.4) for Internet wide usage. However, when operating at rate $R \approx 10$ the bandwidth usage is *traffic* $\approx 150 \left[\frac{kbit}{sec}\right]$ (see Tables 3.7, 3.9 and 3.11), which is quite normal if having fast (e.g. $\geq 256 \left[\frac{kbit}{sec}\right]$) Internet connection.

Throughput: The maximum throughput which could be expected under certain conditions is $R \approx 140$. When stressed there were more free resources available: about 85% of CPU and about 97% of bandwidth resources were unused, but they were not consumed. An explanation of it could be in the internals of the protocol itself, i.e. mutual exclusion devices, barriers, etc.

The main corollary of the stress test results is that the Join/Leave protocol should be able to ***efficiently*** operate at the rate of change ($R \approx 2$), which was previously settled in the analysis of the Gnutella system. When operating at rate $R \approx 2$, the expected resource usage could be as follows:

CPU usage: $CPU \approx 1\%$ (for CPU's faster than $1Ghz$).

Bandwidth usage: *traffic* $\approx 30 \left[\frac{kbit}{sec}\right]$.

3.3.3 Timing Tests

Running the Join/Leave protocol in the realistic environment such as PlanetLab confirms that the fault tolerance is a critical part in the FROST system and must be implemented if it is planned to use the system Internet wide, where the machines are not reliable.

The results of timing tests indicate that the rate of change is $R \approx 3.5$ and according the expectations the Join/Leave protocol satisfies the timing constraints.

Also the results of testing the protocol with different configuration setup indicate that the base parameter could be adjusted for better performance in the system (see Figure 3.9 (b)). The results in Figure 3.9 (a) indicate that the time required for building and dismantling the network increases almost linearly when the number of nodes increases.

Chapter 4

Conclusion

The two main goals of this project were:

1. To implement the prototype of the Join/Leave protocol, whose system architecture and concepts were described in [2]. The purpose of the Join/Leave protocol is to handle the node joins and departures in the FROST network. In general the Join/Leave protocol can be divided into two parts based on the function it should perform:

   Join: The Join/Leave protocol must assure that a node which joins the network will be organized in the hierarchy according to its static performance measure - *SP*.

   Leave: When nodes leave the network (voluntarily or by failing) the Join/Leave protocol has to assure the integrity of the communication architecture by appropriately rearranging the related nodes in the network.

2. To conduct a proof-of-concept evaluation of the Join/Leave protocol by performing system testing, which focuses on the complete system, its functional and non-functional requirements, and its target environment. The following system tests were conducted:

   Functional testing. Functional testing, also called requirements testing, tests if the system perform as promised by the requirements specification.

   Performance testing. Performance testing is used to test if the non-functional requirements are met. Two types of performance tests were conducted:

   Stress tests: The purpose of the stress tests is to evaluate the system when stressed to its limits over a short period of time.

   Timing tests: The purpose of the timing tests is to validate conformance to behavioral and performance constraints and evaluate if the system is fast enough.

4.1 Implementation

Some corrections have been made to the design [2] of the Join/Leave protocol before implementing it. Two main corrections were made:

1. The static node performance *SP* has been derived from available network bandwidth resources disregarding the speed of CPU and the size of the main memory.

2. Joining and adaptation to the network procedure has been optimized.

The Join/Leave protocol has been decomposed and implemented as four components:

1. *Model Component*. A part of a system that implements a model of the Join/Leave protocol.
2. *Function Component*. A part of a system that implements functional requirements of the Join/Leave protocol.
3. *System Interface Component*. A part of a system implementing the interaction with other systems.
4. *User Interface Component*. A part of a system implementing the interaction with users.

4.2 Testing

The following system tests were conducted:

Functional testing. The results of functional tests indicate that the Join/Leave protocol correctly builds and dismantles the network, unless the connection losses during the join or leave operation has happened.

Stress tests. The results of stress tests indicate that the Join/Leave protocol should be able to ***efficiently*** operate at the rate of change ($R \approx 2$), which was previously settled in the analysis of the Gnutella system. When operating at rate $R \approx 2$, the expected resource usage is as follows:

CPU usage: $CPU \approx 1\%$ (for CPU's faster than $1Ghz$).

Bandwidth usage: *traffic* $\approx 30 \left[\frac{kbit}{sec}\right]$.

Timing tests. The results of timing tests ($R \approx 3.5$) indicate that the Join/Leave protocol satisfies the timing constraints, unless the connection losses during the join or leave operation has happened. Also the results of testing the protocol with different configuration setup indicate that the base parameter could be adjusted for better performance in the system (see Figure 3.9 (b)). The results in Figure 3.9 (a) indicate that the time required for building and dismantling the network increases almost linearly when the number of nodes increases.

4.3 Further Work

The parts of the system that were left out will be summarized in this section.

Fault tolerance. Fault tolerance was not designed in [2] and it was not planned to implement it in this work. The intent was to primarily test the Join/Leave protocol assuming that all nodes are reliable and then focus on fault tolerance. Running the Join/Leave protocol in the realistic environment such as PlanetLab confirms that the fault tolerance is a critical part in the FROST system and must be implemented if it is planned to use the system Internet wide, where the machines are not reliable. It is expected that fault tolerance solution will introduce an additional overhead to the system and therefore the system testing has to be performed again to evaluate if the protocol still satisfies the functional and non-functional requirements.

Interoperability. To fulfill its purpose, the Join/Leave protocol has to be incorporated to the FROST system [1].

Security. The maintenance protocols are especially susceptive to the DoS (Denial of Service) attacks. Since the Join/Leave protocol is intended to operate Internet wide there is a high risk of such attacks.

Portability. The protocol should be able to operate on various technical platforms to increase the number of potential users of the FROST system [1]. The current implementation of the Join/Leave protocol is based on Linux OS.

Bibliography

[1] Michael Platz Glibstrup and Lars Kringelbach. FROST - A Distributed Heterogeneous Calculation Platform. Student report, Aalborg University - Department of Computer Science, January 2002.

[2] Li Ming and Arunas Vrubliauskas. Scalability of the FROST System. Student report, Aalborg University - Department of Computer Science, January 2003.

[3] Lars Mathiasen, Andreas Munk-Madsen, Peter Axel Nielsen and Jan Stage. Object Oriented Analysis & Design. Forlaget MARKO. ISBN 87-7751-150-6. 1st edition, 2000.

[4] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the Evolution of Peer-to-Peer Systems. ACM Conf. on Principles of Distributed Computing (PODC), Monterey, CA, July 2002.

[5] PlanetLab - An open testbed for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org

[6] SETI@home - The Search for Extraterrestrial Intelligence. http://setiathome.ssl.berkeley.edu/

[7] http://www.tcpdump.org/

[8] Network protocol analyzer. http://www.ethereal.com/

[9] http://www.research.microsoft.com/padmanab/projects/CoopNet/

[10] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou and Kunwadee Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. Microsoft Research Technical Report, MSR-TR-2002-37, April 2002.

[11] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In Proceedings of MMCN, 2002.

[12] http://www.msnbc.com/

[13] THINK project. http://members.ud.com/home.htm

[14] distributed.net project. http://www.distributed.net/

[15] Gnutella file sharing system. http://www.gnutella.com/

[16] Freenet file sharing system. http://freenet.sourceforge.net/

[17] Napster file sharing system. http://www.napster.com/

[18] Gerard J. Holzmann. Design and Validation of Computer Protocols, Prentice Hall, New Jersey, 1991, ISBN 0-13-539925-4.

[19] Gerard J. Holzmann. The Spin Model Checker, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

Appendix A

Functional Tests: Actual Output

File format is as follows: *hostname*, *ID*, *SP*, *GP*, *level*, ID_{master}, SAL_1, SAL_2, SAL_3.

A.1 Functional Test 1

1.out: planetlab1.xeno.cl.cam.ac.uk 1 201 1626 1 0 85 88 91

2.out: planetlab2.xeno.cl.cam.ac.uk 2 202 1657 1 0 86 89 92

3.out: planetlab3.xeno.cl.cam.ac.uk 3 203 1595 1 0 84 87 90

4.out: planetlab1.iis.sinica.edu.tw 4 4 4 4 58 0 0 0

5.out: planetlab2.iis.sinica.edu.tw 5 5 5 4 59 0 0 0

6.out: planetlab-01.bu.edu 6 6 6 4 60 0 0 0

7.out: planetlab-02.bu.edu 7 7 7 4 61 0 0 0

8.out: PLANETLAB-1.CMCL.CS.CMU.EDU 8 8 8 4 62 0 0 0

9.out: PLANETLAB-2.CMCL.CS.CMU.EDU 9 9 9 4 63 0 0 0

10.out: PLANETLAB-3.CMCL.CS.CMU.EDU 10 10 10 4 64 0 0 0

11.out: planetlab1.comet.columbia.edu 11 11 11 4 65 0 0 0

12.out: planetlab2.comet.columbia.edu 12 12 12 4 66 0 0 0

13.out: planetlab3.comet.columbia.edu 13 13 13 4 67 0 0 0

14.out: planetlab1.cs.cornell.edu 14 14 14 4 68 0 0 0

15.out: planetlab2.cs.cornell.edu 15 15 15 4 69 0 0 0

16.out: planetlab1.cs.duke.edu 16 16 16 4 70 0 0 0

17.out: planetlab2.cs.duke.edu 17 17 17 4 71 0 0 0

18.out: planetlab3.cs.duke.edu 18 18 18 4 72 0 0 0

19.out: planet1.pittsburgh.intel-research.net 19 19 19 4 73 0 0 0

20.out: planet2.pittsburgh.intel-research.net 20 20 20 4 74 0 0 0

21.out: planet3.pittsburgh.intel-research.net 21 21 21 4 75 0 0 0

22.out: planet1.cc.gt.atl.ga.us 22 22 22 4 76 0 0 0

23.out: planet.cc.gt.atl.ga.us 23 23 23 4 77 0 0 0

24.out: lefthand.eecs.harvard.edu 24 24 24 4 78 0 0 0

25.out: righthand.eecs.harvard.edu 25 25 25 4 79 0 0 0

26.out: planetlab1.postel.org 26 26 26 4 80 0 0 0

27.out: planetlab2.postel.org 27 27 27 4 81 0 0 0

28.out: kupl1.ittc.ku.edu 28 28 28 4 82 0 0 0

29.out: kupl2.ittc.ku.edu 29 29 29 4 83 0 0 0

30.out: planetlab1.netlab.uky.edu 30 30 30 4 57 0 0 0

31.out: planetlab2.netlab.uky.edu 31 31 31 4 58 0 0 0

32.out: planetlab1.cs-ipv6.lancs.ac.uk 32 32 32 4 59 0 0 0

33.out: planetlab2.cs-ipv6.lancs.ac.uk 33 33 33 4 60 0 0 0

34.out: planetlab1.lbl.gov 34 34 34 4 61 0 0 0

35.out: planetlab2.lbl.gov 35 35 35 4 62 0 0 0

36.out: planetlab1.eecs.umich.edu 36 36 36 4 63 0 0 0

37.out: planetlab2.eecs.umich.edu 37 37 37 4 64 0 0 0

38.out: planetlab1.lcs.mit.edu 38 38 38 4 65 0 0 0

39.out: planetlab2.lcs.mit.edu 39 39 39 4 66 0 0 0

40.out: planetlab3.lcs.mit.edu 40 40 40 4 67 0 0 0

41.out: planetlab1.cs.northwestern.edu 41 41 41 4 68 0 0 0

42.out: planetlab2.cs.northwestern.edu 42 42 42 4 69 0 0 0

43.out: s1_803.ie.cuhk.edu.hk 43 43 43 4 70 0 0 0

44.out: s2_803.ie.cuhk.edu.hk 44 44 44 4 71 0 0 0

45.out: planet1.ecse.rpi.edu 45 45 45 4 72 0 0 0

46.out: planet2.ecse.rpi.edu 46 46 46 4 73 0 0 0

47.out: ricepl-1.cs.rice.edu 47 47 47 4 74 0 0 0

48.out: ricepl-2.cs.rice.edu 48 48 48 4 75 0 0 0

49.out: planetlab-1.Stanford.EDU 49 49 49 4 76 0 0 0

50.out: planetlab-2.Stanford.EDU 50 50 50 4 77 0 0 0

51.out: edi.tkn.tu-berlin.de 51 51 51 4 78 0 0 0

52.out: miranda.tkn.tu-berlin.de 52 52 52 4 79 0 0 0

53.out: pl1.cs.utk.edu 53 53 53 4 80 0 0 0

54.out: pl2.cs.utk.edu 54 54 54 4 81 0 0 0

55.out: planetlab1.cs.ubc.ca 55 55 55 4 82 0 0 0

56.out: planetlab2.cs.ubc.ca 56 56 56 4 83 0 0 0

57.out: PlanetLab1.Millennium.Berkeley.EDU 57 57 87 3 84 30 0 0

58.out: PlanetLab2.Millennium.Berkeley.EDU 58 58 93 3 85 31 4 0

59.out: PlanetLab3.Millennium.Berkeley.EDU 59 59 96 3 86 32 5 0

60.out: Planetlab1.CS.UCLA.EDU 60 60 99 3 87 33 6 0

61.out: Planetlab2.CS.UCLA.EDU 61 61 102 3 88 34 7 0

62.out: planetlab1.ucsd.edu 62 62 105 3 89 35 8 0

63.out: planetlab2.ucsd.edu 63 63 108 3 90 36 9 0

64.out: planetlab3.ucsd.edu 64 64 111 3 91 37 10 0

65.out: planet1.cs.ucsb.edu 65 65 114 3 92 38 11 0

66.out: planet2.cs.ucsb.edu 66 66 117 3 84 39 12 0

67.out: planetlab1.cs.umass.edu 67 67 120 3 85 40 13 0

68.out: planetlab2.cs.umass.edu 68 68 123 3 86 41 14 0

69.out: planetlab1.cs.unc.edu 69 69 126 3 87 42 15 0

70.out: planetlab2.cs.unc.edu 70 70 129 3 88 43 16 0

71.out: planetlab1.cs.unibo.it 71 71 132 3 89 44 17 0

72.out: planetlab2.cs.unibo.it 72 72 135 3 90 45 18 0

73.out: planetlab1.cs.uiuc.edu 73 73 138 3 91 46 19 0

74.out: planetlab2.cs.uiuc.edu 74 74 141 3 92 47 20 0

75.out: planet-lab.cs.umd.edu 75 75 144 3 84 48 21 0

76.out: pl1.ece.toronto.edu 76 76 147 3 85 49 22 0

77.out: pl2.ece.toronto.edu 77 77 150 3 86 50 23 0

78.out: planetlab1.cs.virginia.edu 78 78 153 3 87 51 24 0

79.out: planetlab2.cs.virginia.edu 79 79 156 3 88 52 25 0

80.out: planetlab01.cs.washington.edu 80 80 159 3 89 53 26 0

81.out: planetlab02.cs.washington.edu 81 81 162 3 90 54 27 0

82.out: planetlab03.cs.washington.edu 82 82 165 3 91 55 28 0

83.out: planetlab1.cis.upenn.edu 83 83 168 3 92 56 29 0

84.out: planetlab2.cis.upenn.edu 84 84 432 2 3 75 66 57

85.out: planetlab-1.it.uu.se 85 85 445 2 1 76 67 58

86.out: planetlab-2.it.uu.se 86 86 455 2 2 77 68 59

87.out: planetlab3.flux.utah.edu 87 87 465 2 3 78 69 60

88.out: vn2.cs.wustl.edu 88 88 475 2 1 79 70 61

89.out: vn3.cs.wustl.edu 89 89 485 2 2 80 71 62

90.out: planetlab1.cs.wayne.edu 90 90 495 2 3 81 72 63

91.out: planetlab1.cs.wisc.edu 91 91 505 2 1 82 73 64

92.out: planetlab2.cs.wisc.edu 92 92 515 2 2 83 74 65

A.2 Functional Test 2

1.out: planetlab1.xeno.cl.cam.ac.uk 1 201 222 1 0 10 7 4

2.out: planetlab2.xeno.cl.cam.ac.uk 2 202 226 1 0 11 8 5

3.out: planetlab3.xeno.cl.cam.ac.uk 3 203 230 1 0 12 9 6

4.out: planetlab1.iis.sinica.edu.tw 4 4 4 2 1 0 0 0

5.out: planetlab2.iis.sinica.edu.tw 5 5 5 2 2 0 0 0

6.out: planetlab-01.bu.edu 6 6 6 2 3 0 0 0

7.out: planetlab-02.bu.edu 7 7 7 2 1 0 0 0

8.out: PLANETLAB-1.CMCL.CS.CMU.EDU 8 8 8 2 2 0 0 0

9.out: PLANETLAB-2.CMCL.CS.CMU.EDU 9 9 9 2 3 0 0 0

10.out: PLANETLAB-3.CMCL.CS.CMU.EDU 10 10 10 2 1 0 0 0

11.out: planetlab1.comet.columbia.edu 11 11 11 2 2 0 0 0

12.out: planetlab2.comet.columbia.edu 12 12 12 2 3 0 0 0

Appendix B

Functional Tests: Functional Testing System Output

B.1 Functional Test 1: FT1.out

```
Functional Test Passed!
------------------------------------
Expected GP=4878, at highest level
Actual   GP=4878, at highest level
------------------------------------
ID= 1, SP=201, GP=1626, level= 1 MST=N/A SAL[1]=85 SAL[2]=88 SAL[3]=91
ID= 2, SP=202, GP=1657, level= 1 MST=N/A SAL[1]=86 SAL[2]=89 SAL[3]=92
ID= 3, SP=203, GP=1595, level= 1 MST=N/A SAL[1]=84 SAL[2]=87 SAL[3]=90
ID= 4, SP= 4, GP= 4, level= 4 MST= 58
ID= 5, SP= 5, GP= 5, level= 4 MST= 59
ID= 6, SP= 6, GP= 6, level= 4 MST= 60
ID= 7, SP= 7, GP= 7, level= 4 MST= 61
ID= 8, SP= 8, GP= 8, level= 4 MST= 62
ID= 9, SP= 9, GP= 9, level= 4 MST= 63
ID=10, SP= 10, GP= 10, level= 4 MST= 64
ID=11, SP= 11, GP= 11, level= 4 MST= 65
ID=12, SP= 12, GP= 12, level= 4 MST= 66
ID=13, SP= 13, GP= 13, level= 4 MST= 67
ID=14, SP= 14, GP= 14, level= 4 MST= 68
ID=15, SP= 15, GP= 15, level= 4 MST= 69
ID=16, SP= 16, GP= 16, level= 4 MST= 70
ID=17, SP= 17, GP= 17, level= 4 MST= 71
ID=18, SP= 18, GP= 18, level= 4 MST= 72
ID=19, SP= 19, GP= 19, level= 4 MST= 73
ID=20, SP= 20, GP= 20, level= 4 MST= 74
ID=21, SP= 21, GP= 21, level= 4 MST= 75
ID=22, SP= 22, GP= 22, level= 4 MST= 76
ID=23, SP= 23, GP= 23, level= 4 MST= 77
ID=24, SP= 24, GP= 24, level= 4 MST= 78
ID=25, SP= 25, GP= 25, level= 4 MST= 79
ID=26, SP= 26, GP= 26, level= 4 MST= 80
ID=27, SP= 27, GP= 27, level= 4 MST= 81
ID=28, SP= 28, GP= 28, level= 4 MST= 82
ID=29, SP= 29, GP= 29, level= 4 MST= 83
ID=30, SP= 30, GP= 30, level= 4 MST= 57
ID=31, SP= 31, GP= 31, level= 4 MST= 58
ID=32, SP= 32, GP= 32, level= 4 MST= 59
```

```
ID=33, SP= 33, GP= 33, level= 4 MST= 60
ID=34, SP= 34, GP= 34, level= 4 MST= 61
ID=35, SP= 35, GP= 35, level= 4 MST= 62                                    40
ID=36, SP= 36, GP= 36, level= 4 MST= 63
ID=37, SP= 37, GP= 37, level= 4 MST= 64
ID=38, SP= 38, GP= 38, level= 4 MST= 65
ID=39, SP= 39, GP= 39, level= 4 MST= 66
ID=40, SP= 40, GP= 40, level= 4 MST= 67
ID=41, SP= 41, GP= 41, level= 4 MST= 68
ID=42, SP= 42, GP= 42, level= 4 MST= 69
ID=43, SP= 43, GP= 43, level= 4 MST= 70
ID=44, SP= 44, GP= 44, level= 4 MST= 71
ID=45, SP= 45, GP= 45, level= 4 MST= 72                                    50
ID=46, SP= 46, GP= 46, level= 4 MST= 73
ID=47, SP= 47, GP= 47, level= 4 MST= 74
ID=48, SP= 48, GP= 48, level= 4 MST= 75
ID=49, SP= 49, GP= 49, level= 4 MST= 76
ID=50, SP= 50, GP= 50, level= 4 MST= 77
ID=51, SP= 51, GP= 51, level= 4 MST= 78
ID=52, SP= 52, GP= 52, level= 4 MST= 79
ID=53, SP= 53, GP= 53, level= 4 MST= 80
ID=54, SP= 54, GP= 54, level= 4 MST= 81
ID=55, SP= 55, GP= 55, level= 4 MST= 82                                    60
ID=56, SP= 56, GP= 56, level= 4 MST= 83
ID=57, SP= 57, GP= 87, level= 3 MST= 84 SAL[1]=30
ID=58, SP= 58, GP= 93, level= 3 MST= 85 SAL[1]= 4 SAL[2]=31
ID=59, SP= 59, GP= 96, level= 3 MST= 86 SAL[1]= 5 SAL[2]=32
ID=60, SP= 60, GP= 99, level= 3 MST= 87 SAL[1]= 6 SAL[2]=33
ID=61, SP= 61, GP= 102, level= 3 MST= 88 SAL[1]= 7 SAL[2]=34
ID=62, SP= 62, GP= 105, level= 3 MST= 89 SAL[1]= 8 SAL[2]=35
ID=63, SP= 63, GP= 108, level= 3 MST= 90 SAL[1]= 9 SAL[2]=36
ID=64, SP= 64, GP= 111, level= 3 MST= 91 SAL[1]=10 SAL[2]=37
ID=65, SP= 65, GP= 114, level= 3 MST= 92 SAL[1]=11 SAL[2]=38               70
ID=66, SP= 66, GP= 117, level= 3 MST= 84 SAL[1]=12 SAL[2]=39
ID=67, SP= 67, GP= 120, level= 3 MST= 85 SAL[1]=13 SAL[2]=40
ID=68, SP= 68, GP= 123, level= 3 MST= 86 SAL[1]=14 SAL[2]=41
ID=69, SP= 69, GP= 126, level= 3 MST= 87 SAL[1]=15 SAL[2]=42
ID=70, SP= 70, GP= 129, level= 3 MST= 88 SAL[1]=16 SAL[2]=43
ID=71, SP= 71, GP= 132, level= 3 MST= 89 SAL[1]=17 SAL[2]=44
ID=72, SP= 72, GP= 135, level= 3 MST= 90 SAL[1]=18 SAL[2]=45
ID=73, SP= 73, GP= 138, level= 3 MST= 91 SAL[1]=19 SAL[2]=46
ID=74, SP= 74, GP= 141, level= 3 MST= 92 SAL[1]=20 SAL[2]=47
ID=75, SP= 75, GP= 144, level= 3 MST= 84 SAL[1]=21 SAL[2]=48               80
ID=76, SP= 76, GP= 147, level= 3 MST= 85 SAL[1]=22 SAL[2]=49
ID=77, SP= 77, GP= 150, level= 3 MST= 86 SAL[1]=23 SAL[2]=50
ID=78, SP= 78, GP= 153, level= 3 MST= 87 SAL[1]=24 SAL[2]=51
ID=79, SP= 79, GP= 156, level= 3 MST= 88 SAL[1]=25 SAL[2]=52
ID=80, SP= 80, GP= 159, level= 3 MST= 89 SAL[1]=26 SAL[2]=53
ID=81, SP= 81, GP= 162, level= 3 MST= 90 SAL[1]=27 SAL[2]=54
ID=82, SP= 82, GP= 165, level= 3 MST= 91 SAL[1]=28 SAL[2]=55
ID=83, SP= 83, GP= 168, level= 3 MST= 92 SAL[1]=29 SAL[2]=56
ID=84, SP= 84, GP= 432, level= 2 MST= 3 SAL[1]=57 SAL[2]=66 SAL[3]=75
ID=85, SP= 85, GP= 445, level= 2 MST= 1 SAL[1]=58 SAL[2]=67 SAL[3]=76      90
ID=86, SP= 86, GP= 455, level= 2 MST= 2 SAL[1]=59 SAL[2]=68 SAL[3]=77
ID=87, SP= 87, GP= 465, level= 2 MST= 3 SAL[1]=60 SAL[2]=69 SAL[3]=78
ID=88, SP= 88, GP= 475, level= 2 MST= 1 SAL[1]=61 SAL[2]=70 SAL[3]=79
```

```
ID=89, SP= 89, GP= 485, level= 2 MST= 2 SAL[1]=62 SAL[2]=71 SAL[3]=80
ID=90, SP= 90, GP= 495, level= 2 MST= 3 SAL[1]=63 SAL[2]=72 SAL[3]=81
ID=91, SP= 91, GP= 505, level= 2 MST= 1 SAL[1]=64 SAL[2]=73 SAL[3]=82
ID=92, SP= 92, GP= 515, level= 2 MST= 2 SAL[1]=65 SAL[2]=74 SAL[3]=83
```

B.2 Functional Test 2: FT2.out

```
Functional Test Passed!
---------------------------------
Expected GP=678, at highest level
Actual   GP=678, at highest level
---------------------------------
ID= 1, SP=201, GP= 222, level= 1 MST=N/A SAL[1]=10 SAL[2]= 7 SAL[3]= 4
ID= 2, SP=202, GP= 226, level= 1 MST=N/A SAL[1]=11 SAL[2]= 8 SAL[3]= 5
ID= 3, SP=203, GP= 230, level= 1 MST=N/A SAL[1]=12 SAL[2]= 9 SAL[3]= 6
ID= 4, SP= 4, GP= 4, level= 2 MST= 1
ID= 5, SP= 5, GP= 5, level= 2 MST= 2                                    10
ID= 6, SP= 6, GP= 6, level= 2 MST= 3
ID= 7, SP= 7, GP= 7, level= 2 MST= 1
ID= 8, SP= 8, GP= 8, level= 2 MST= 2
ID= 9, SP= 9, GP= 9, level= 2 MST= 3
ID=10, SP= 10, GP= 10, level= 2 MST= 1
ID=11, SP= 11, GP= 11, level= 2 MST= 2
ID=12, SP= 12, GP= 12, level= 2 MST= 3
```

Appendix C

Stress Tests: Ethereal Output

C.1 Stress Test 1

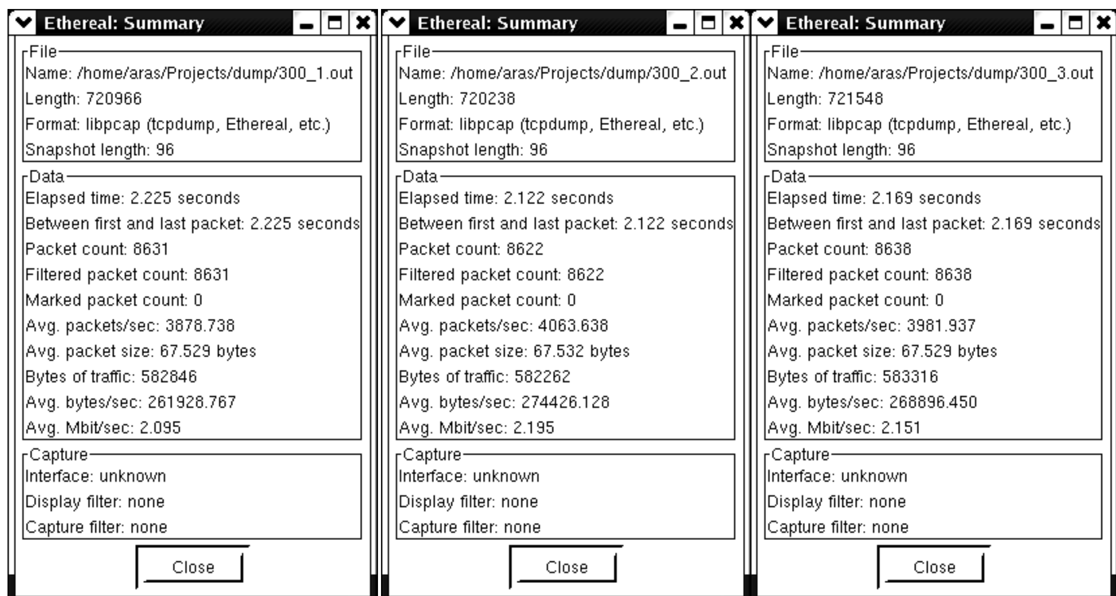


Figure C.1: ***Test Case 1.***

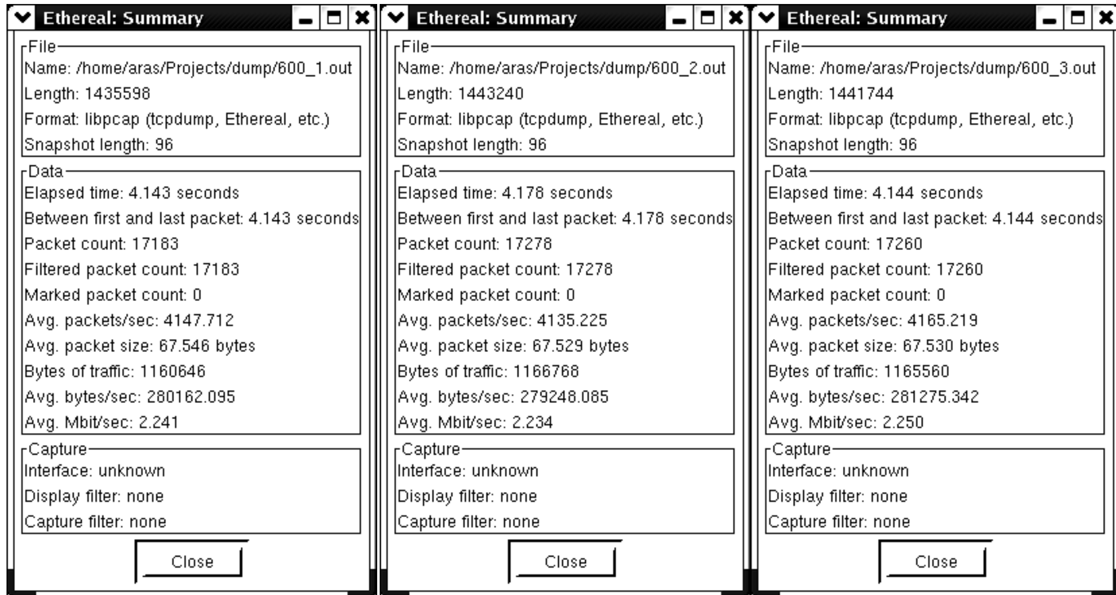


Figure C.2: ***Test Case 2.***

C.2 Stress Test 2

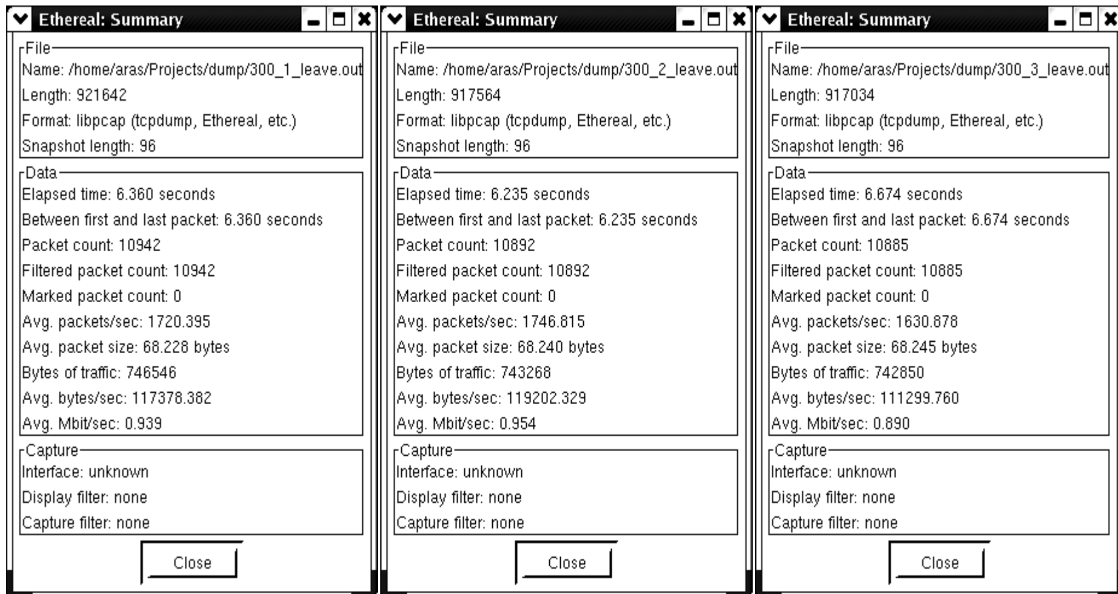


Figure C.3: ***Test Case 1.***

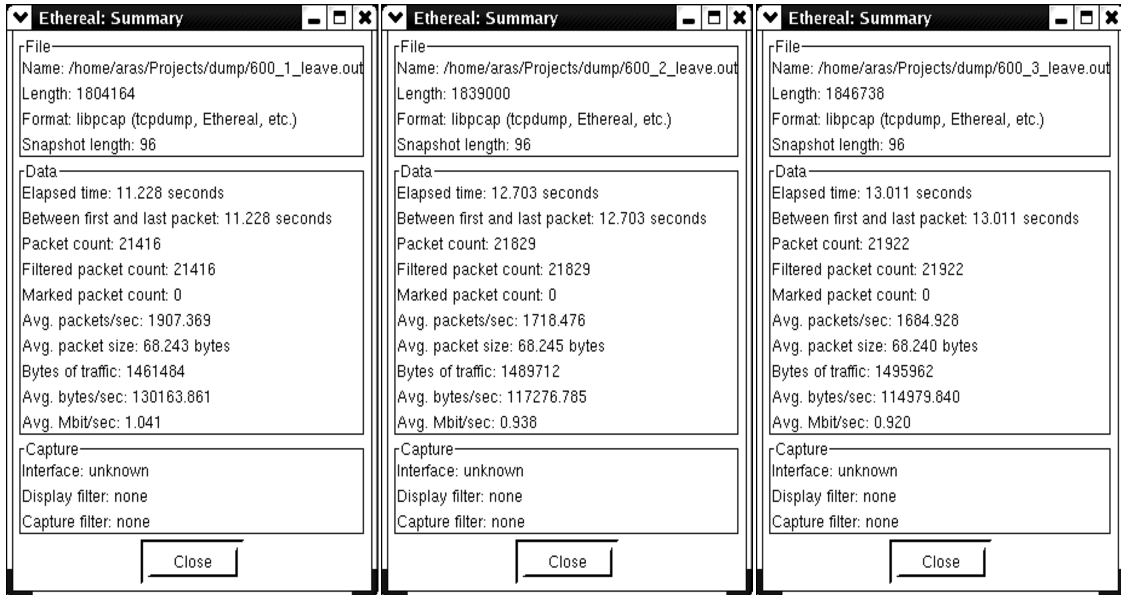


Figure C.4: ***Test Case 2.***

C.3 Stress Test 3

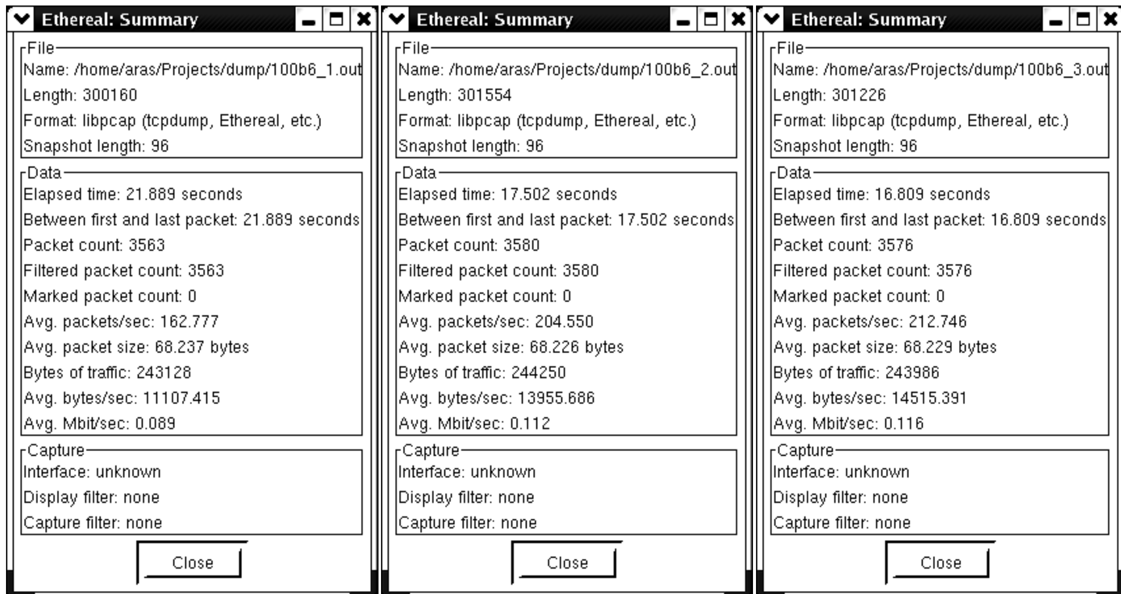


Figure C.5: ***Test Case 1.1***

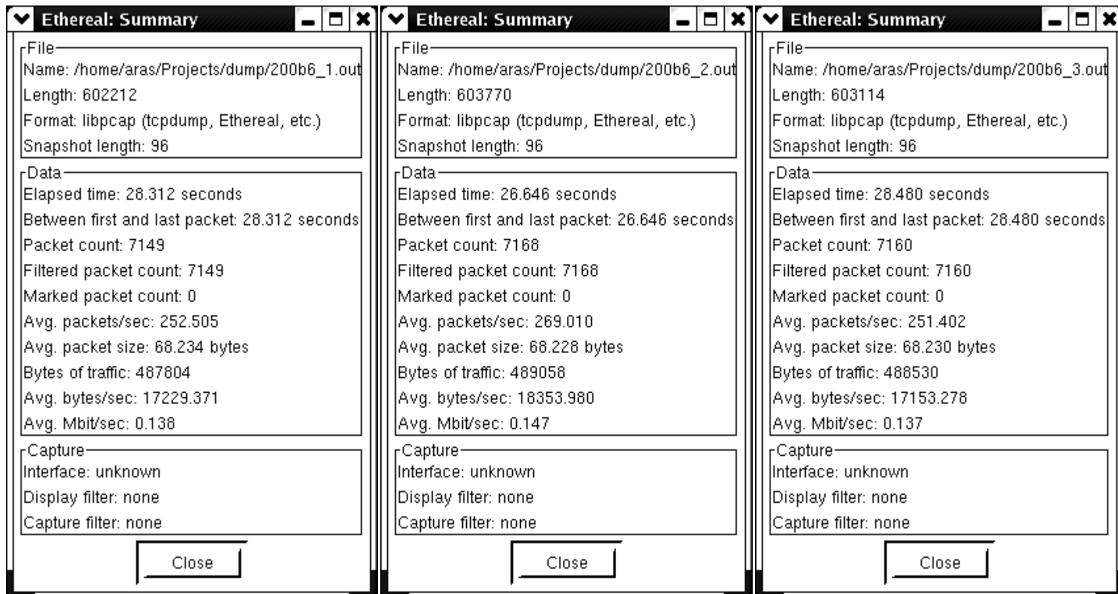


Figure C.6: ***Test Case 1.2***

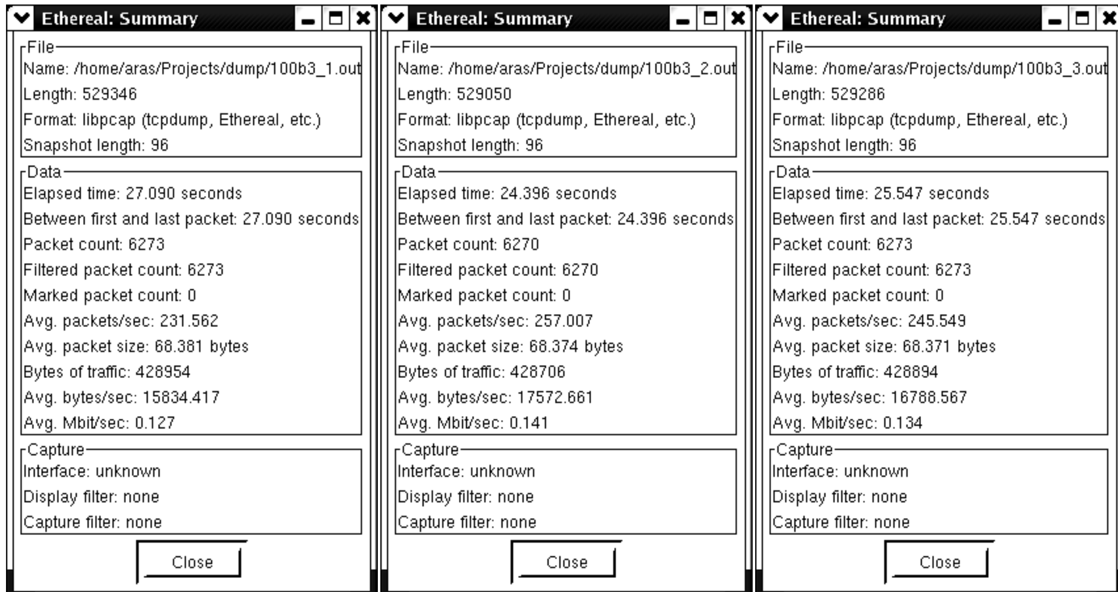


Figure C.7: ***Test Case 2.1***

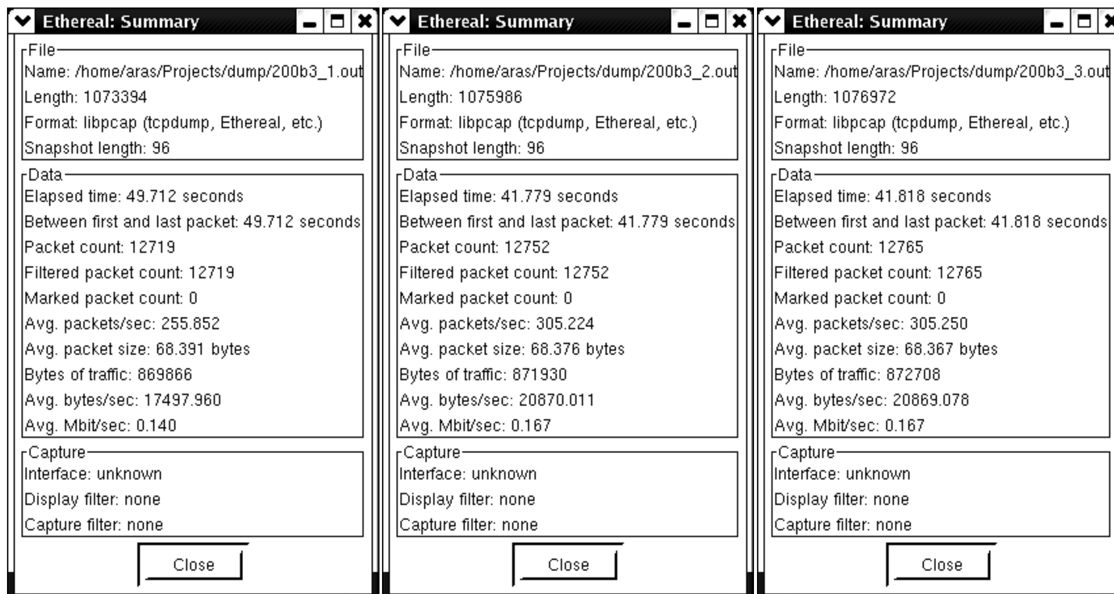


Figure C.8: ***Test Case 2.2***

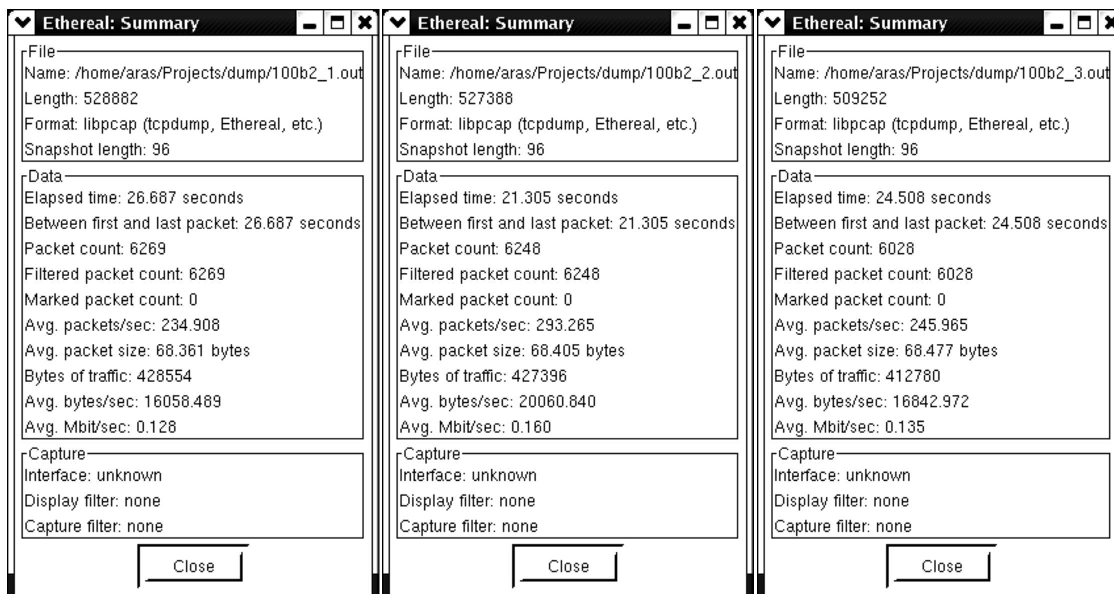


Figure C.9: ***Test Case 3.1***

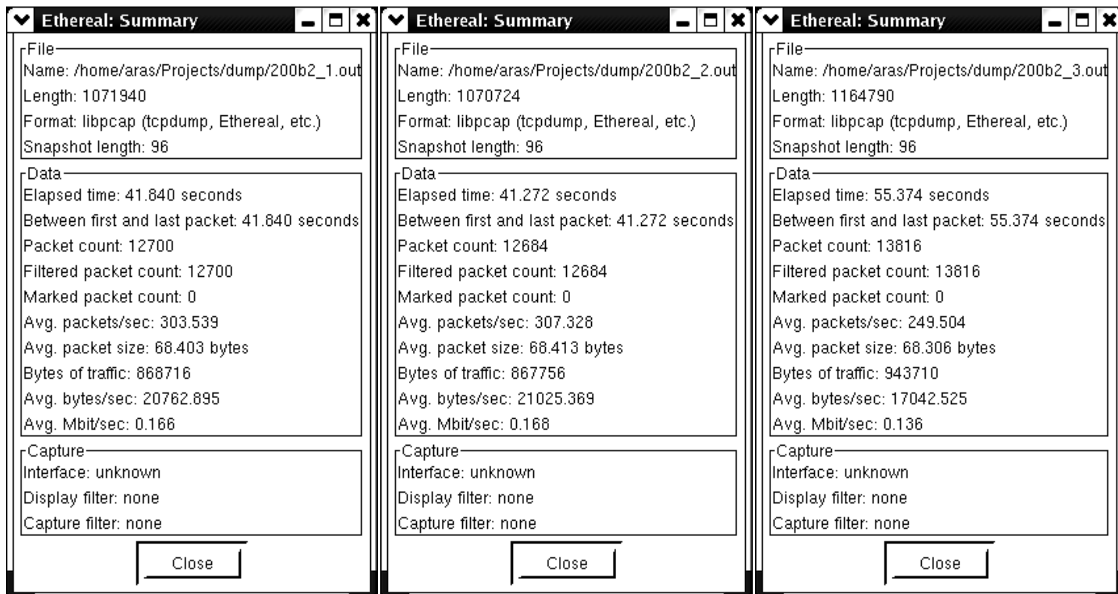


Figure C.10: ***Test Case 3.2***

Appendix D

Timing Tests Output

D.1 Timing Test 1

D.1.1 Test Case 1

1_endjoin.out :

started : Tue Jun 3 08:24:12 2003
finished: Tue Jun 3 08:24:13 2003
elapsed : 1.00 seconds

2_endjoin.out :

started : Tue Jun 3 08:24:13 2003
finished: Tue Jun 3 08:24:13 2003
elapsed : 0.00 seconds

3_endjoin.out :

started : Tue Jun 3 08:24:14 2003
finished: Tue Jun 3 08:24:14 2003
elapsed : 0.00 seconds

4_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:16 2003
elapsed : 0.00 seconds

5_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:17 2003
elapsed : 1.00 seconds

6_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:18 2003
elapsed : 2.00 seconds

7_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:20 2003
elapsed : 4.00 seconds

8_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:19 2003
elapsed : 3.00 seconds

9_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:25 2003
elapsed : 9.00 seconds

10_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:18 2003
elapsed : 2.00 seconds

11_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:27 2003
elapsed : 11.00 seconds

12_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:29 2003
elapsed : 13.00 seconds

13_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:18 2003
elapsed : 2.00 seconds

14_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:17 2003
elapsed : 0.00 seconds

15_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:19 2003
elapsed : 3.00 seconds

16_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:29 2003

elapsed : 13.00 seconds

17_endjoin.out : N/A (Failed)

18_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:29 2003
elapsed : 13.00 seconds

19_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:26 2003
elapsed : 10.00 seconds

20_endjoin.out :

started : Tue Jun 3 08:24:16 2003
finished: Tue Jun 3 08:24:20 2003
elapsed : 4.00 seconds

21_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:28 2003
elapsed : 11.00 seconds

22_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:23 2003
elapsed : 6.00 seconds

23_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:31 2003
elapsed : 14.00 seconds

24_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:26 2003
elapsed : 9.00 seconds

25_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:19 2003
elapsed : 2.00 seconds

26_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:28 2003
elapsed : 11.00 seconds

27_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:28 2003
elapsed : 11.00 seconds

28_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:29 2003
elapsed : 12.00 seconds

29_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:32 2003
elapsed : 15.00 seconds

30_endjoin.out : N/A (Failed)

31_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:28 2003
elapsed : 11.00 seconds

32_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:20 2003
elapsed : 3.00 seconds

33_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:34 2003
elapsed : 17.00 seconds

34_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:21 2003
elapsed : 4.00 seconds

35_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:21 2003
elapsed : 4.00 seconds

36_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:21 2003
elapsed : 4.00 seconds

37_endjoin.out :

started : Tue Jun 3 08:24:17 2003
finished: Tue Jun 3 08:24:23 2003
elapsed : 6.00 seconds

38_endjoin.out :

started : Tue Jun 3 08:24:18 2003
finished: Tue Jun 3 08:24:23 2003
elapsed : 5.00 seconds

39_endjoin.out :

started : Tue Jun 3 08:24:19 2003
finished: Tue Jun 3 08:24:25 2003
elapsed : 6.00 seconds

40_endjoin.out :

started : Tue Jun 3 08:24:19 2003
finished: Tue Jun 3 08:24:25 2003
elapsed : 6.00 seconds

D.1.2 Test Case 2

1_endleave.out : N/A (Killed)

2_endleave.out : N/A (Killed)

3_endleave.out : N/A (Killed)

4_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:17 2003
- **elapsed :** 0.00 seconds

5_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:20 2003
- **elapsed :** 3.00 seconds

6_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 2.00 seconds

7_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:19 2003
- **elapsed :** 2.00 seconds

8_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:17 2003
- **elapsed :** 0.00 seconds

9_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 1.00 seconds

10_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 1.00 seconds

11_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 1.00 seconds

12_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:19 2003
- **elapsed :** 2.00 seconds

13_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:17 2003
- **elapsed :** 0.00 seconds

14_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:17 2003
- **elapsed :** 0.00 seconds

15_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 1.00 seconds

16_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:17 2003
- **elapsed :** 0.00 seconds

17_endleave.out : N/A

18_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:17 2003
- **elapsed :** 0.00 seconds

19_endleave.out :

- **started :** Tue Jun 3 08:25:18 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 0.00 seconds

20_endleave.out :

- **started :** Tue Jun 3 08:25:18 2003
- **finished:** Tue Jun 3 08:25:19 2003
- **elapsed :** 1.00 seconds

21_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 1.00 seconds

22_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:19 2003
- **elapsed :** 2.00 seconds

23_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:18 2003
- **elapsed :** 1.00 seconds

24_endleave.out :

- **started :** Tue Jun 3 08:25:17 2003
- **finished:** Tue Jun 3 08:25:17 2003
- **elapsed :** 0.00 seconds

25_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:18 2003

elapsed : 0.00 seconds

26_endleave.out :

started : Tue Jun 3 08:25:17 2003

finished: Tue Jun 3 08:25:17 2003

elapsed : 0.00 seconds

27_endleave.out :

started : Tue Jun 3 08:25:17 2003

finished: Tue Jun 3 08:25:17 2003

elapsed : 0.00 seconds

28_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:19 2003

elapsed : 1.00 seconds

29_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:20 2003

elapsed : 2.00 seconds

30_endleave.out : N/A

31_endleave.out :

started : Tue Jun 3 08:25:17 2003

finished: Tue Jun 3 08:25:17 2003

elapsed : 0.00 seconds

32_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:18 2003

elapsed : 0.00 seconds

33_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:18 2003

elapsed : 0.00 seconds

34_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:19 2003

elapsed : 1.00 seconds

35_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:18 2003

elapsed : 0.00 seconds

36_endleave.out :

started : Tue Jun 3 08:25:17 2003

finished: Tue Jun 3 08:25:18 2003

elapsed : 1.00 seconds

37_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:19 2003

elapsed : 1.00 seconds

38_endleave.out :

started : Tue Jun 3 08:25:18 2003

finished: Tue Jun 3 08:25:19 2003

elapsed : 1.00 seconds

39_endleave.out :

started : Tue Jun 3 08:25:19 2003

finished: Tue Jun 3 08:25:21 2003

elapsed : 2.00 seconds

40_endleave.out :

started : Tue Jun 3 08:25:19 2003

finished: Tue Jun 3 08:25:19 2003

elapsed : 0.00 seconds