Design and Implementation of Composite Protocols

by

Magesh Kannan

B.E. (Electronics and Communication Engineering),
Anna University, Chennai, India, 1997

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Professor in charge

Committee Members

Date thesis accepted

To My Parents

Acknowledgements

I wish to express my sincere gratitude to Dr. Gary Minden, my faculty advisor and thesis committee chairperson, for his encouragement and guidance throughout my work on this topic. His ideas and advice have helped me conduct better research. I thank Dr. Joseph Evans and Dr. Victor Frost for serving on my committee and for reviewing my thesis. Special thanks to Ed Komp for his professional supervision of the Composite Protocols project, for the numerous thought-provoking discussions I had with him, for many of his insights that shaped this thesis and for reviewing my thesis document. Many thanks to my teammates: Sandeep, Shyang, Srujana and Steve for their cooperation and help during my work.

I thank all my friends in KU, who made my stay here memorable and enjoyable. My thanks also go to my teammates in SPCNL and my friends in Anna University, whose support helped me pursue a graduate degree.

My heartfelt thanks to my parents, my brother and sister-in-law and others in the family. To them, I owe all the good things that have happened to me.

Abstract

A composite protocol is a collection of single-function components arranged in an orderly fashion to implement a network communications capability. This thesis presents a template for the design of protocol components. Components are specified in terms of finite state machines, memory objects used and formal properties provided to packets. The linear stacking approach is used as the composition technique to build composite protocols from components. The specification methodology for components has been developed to facilitate formal reasoning of logical correctness of protocols, as is the choice of the composition technique. Implementing protocols as a collection of components has other benefits: better scope for reuse, quick development of variants of protocols and customization of protocols to application requirements.

Ensemble, a group communication system, has been chosen as the basis for implementing a framework for composite protocols. The Ensemble system has been adapted for the purposes of a general protocol framework and a state machine executor has been implemented. The functional components of IP and UDP have been identified, specified and implemented as per the design methodology. A functional equivalent of FTP has also been specified and implemented as an example of the application of a control interface to create smaller components. The performance of a UDP-like composite protocol has been measured and compared with that of the Linux TCP/IP implementation.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation for Composite Protocols

Conventional network protocols are designed by following the layering principle suggested by the OSI model [1]. Protocol functionality is divided into layers with well-defined interfaces between them. Multiple layers are then arranged in a stack with each layer providing a more sophisticated abstraction of a communication channel than the layer below it. The principle of layering has facilitated the solution to the problem of network communication by breaking down the problem into smaller and more manageable sub-problems. However, the layering principle has only been used as a guideline for the design of network protocols. Implementations often ignore this principle, because strictly layered implementations perform poorly compared to monolithic alternatives [2]. Efficiency of protocol processing has long been a major concern for protocol developers. This concern has resulted in implementations in which modularity is traded for efficiency. With the advent of active networks [3], priorities of protocol development have changed drastically. Though efficiency is always important, the notion of correctness of network protocols is gaining equal or better priority. It is inherently challenging to design and implement protocols correctly due to the asynchronous nature of communication between physically distributed endpoints. Monolithic implementations of protocols aggravate this problem by their very structure. It is difficult to make any assertion about the behavior of protocol processing when implemented in a monolithic fashion.

Each layer in the OSI model contains more than one protocol function. For example, the network layer includes fragmentation and re-assembly, routing, bit error detection and routing loop mitigation. With a conventional layered approach, a network application has few choices in selecting a protocol. This limitation is evident when applications have to choose a transport protocol. There is a wide gap between the capabilities offered by UDP [4] and TCP [5]. Many application designers tend to opt for TCP, even when the application does not need all of TCP's functionality. If protocol software were organized in terms of protocol functions, applications would have more choices in configuring the protocol they need. An application may be able to choose the exact set of protocol functions that it needs and build a protocol out of them.

Reuse of existing code in new software reduces the time and effort required to develop new software. Reuse is considered beneficial in the Software Engineering community and many techniques have been used to promote software reuse. Development of libraries and object-oriented software design have code reuse as one of their important goals. Unfortunately, reuse of code is not prevalent in protocol implementations. Implementations may reuse common utility functions like lookup tables but protocol functions are rarely reused. Even when a monolithic implementation is structured in a modular fashion, reusing parts of this code-base in other protocols requires careful consideration by a protocol developer. The major hurdle in reuse of protocol functions is the set of assumptions made by implementations about the environment in which they operate.

One of the major advantages of active networks is the relative ease of introduction of new services, without the need for universal agreement. Many new services require only minor changes to the protocols deployed. For example, a streaming multimedia application may decide to include an error-correction function when it can afford to utilize the additional bandwidth required by typical error correction schemes. A monolithic implementation is not flexible enough to quickly make such small changes and deploy a variant of an existing protocol. For a protocol implementation structured in terms of protocol functions, it may be as simple as swapping an error-detection component for a suitable error-correction component. The rest of the protocol used by the multimedia application would largely remain unaffected by this component swap.

Given the reasons mentioned above, it is necessary to rethink the approach to designing and implementing network protocol software. Taking the layering principle to its logical extreme, each protocol function shall be designed and implemented as a separate entity. Each entity shall implement one and only one protocol function. It shall only make a limited but standard set of assumptions about its operating environment. There shall be a method to build a protocol by combining these single function entities in an orderly fashion. The entities shall be designed and implemented in a manner that helps in validating that the specification of the entity is logically correct and in verifying that the implementation conforms to the specification. This thesis proposes a solution for designing protocol software that meets the requirements mentioned above.

1.2 Salient features of our approach

Each protocol function is designed and implemented as what is termed a "protocol component". Protocol components are arranged in an orderly fashion to form a "composite protocol". An application may choose the set of protocol components based on the protocol functions needed and configure a custom composite protocol. Many protocol components offer formal properties to the packets they process. Protocol components are specified in terms of a state machine, action functions associated with state transitions and classifications of memory used by the component. This section briefly discusses the salient features of our approach. The design is explained in detail in chapter 3.

1. This thesis proposes a methodology to precisely specify the functionality of a protocol component.
2. The functionality of a protocol component is specified in terms of a state machine, which facilitates many formal verification and static code-analysis tasks.
3. Given the association between protocol components and the formal properties that they provide and a library of components, it is possible to implement a

"Properties-in Protocol-out" configuration tool that automatically builds a composite protocol that provides the requested formal properties.

4. The thesis classifies memory objects used by components into categories based on their scope and extent. This classification makes many external dependencies of components explicit.
5. Since a component implements a single protocol function, formal verification of the logical correctness of its functionality is more manageable than that of its monolithic alternative.
6. Single function components are more likely to be reused in new protocols.
7. Constructing new and innovative variants of an existing composite protocol is relatively easy, given a library of single function components.

1.3 Thesis organization

The rest of this document is organized as follows. Chapter 2 surveys the literature on the design of modular protocol systems. The salient features of each approach are described, followed by a discussion of their relevance to composite protocols. The design template of a protocol component is explained in Chapter 3. This chapter explains the elements of a component, describes the infrastructure supported by the protocol framework and discusses some methods of composing protocols from components. Chapter 4 is dedicated to the implementation of composite protocols using the group communication system Ensemble. The realization of a state machine executor over Ensemble is covered in this chapter. Chapter 5 discusses issues pertaining to the design of individual components. The design of two components, fragmentation and re-assembly and forwarding is explained and some scenarios of building protocols with different arrangements of these components are discussed. The application of the control interface, a means of inter-component and intra-protocol communication, to the design of a file-transfer application is described in Chapter 6. Chapter 7 describes the tests conducted to evaluate the performance of a UDP-like composite protocol and discusses the results. Chapter 8 summarizes the work and suggests topics for further improvement.

2. Related Work

This chapter surveys the state of the art in the design of modular protocol software systems. Each approach is briefly described followed by a discussion of its main advantages and shortcomings.

2.1 x-kernel

2.1.1 Features

x-kernel [6][7] provides an architecture for implementing and composing network protocols. It is an operating system that provides explicit support for developing protocols. Apart from multiple address spaces and lightweight processes, it supports three protocol-specific classes: protocols, sessions and messages. Protocol objects implement protocol-global functions like managing allocation of port numbers and demultiplex received messages to appropriate sessions. A session is an abstraction of a live connection. Sessions interpret messages and maintain state pertaining to individual connections. A message is an active object that moves between protocol and session objects.

Protocol and session objects add headers to messages before transmission and strip headers off received messages. The relationships between protocol objects are not static. At kernel configuration time, protocol objects are given capabilities to their neighbors in the protocol graph. For example, if a particular configuration of the kernel supports the protocol graph UDP-IP-Ethernet, the IP protocol object is given capabilities to access the protocol objects of UDP and Ethernet. Protocol and session objects offer a uniform interface. Protocol objects expose functions to create sessions. Sessions can be created via active opens (a protocol stack initiating a connection to another) or passive opens (a stack expressing its willingness to accept incoming connections). Session objects provide functions to accept messages from their neighbor protocol objects. Message objects are optimized to accept addition of headers in last-in first-out (LIFO) order and to minimize copying of payloads.

Protocol objects are created and initialized at kernel bootstrap time and they exist as long as the kernel is running. A session object is created when an application initiates a connection and exists for the duration of the connection. Message objects are created for every message sent or received. In the common case, the operating system is designed to send and receive messages without any context switches. For sending messages, this is achieved by letting a user process execute with the privileges of a kernel process upon making a system call. On message reception, a lightweight process is assigned to shepherd the received message through a series of protocol and session objects. x-kernel also provides efficient implementations to manage buffers, maps and alarms.

The x-kernel architecture has been applied to implement configurable protocols. Conventional protocols have the following features when represented as a graph. The topology of the graph is simple and is determined at protocol standardization time. Individual protocols that form the nodes of the graph include complex functionality. The goal of composing protocols with x-kernel is to follow the complement of this architecture. Protocol graphs are complex and are configured based on application requirements. Nodes of the graph implement single functions. These single function entities are called micro-protocols. Another innovation of this architecture is the notion of a virtual protocol. If the path taken by a message through a protocol graph is likened to control flow in a program, virtual protocols are analogous to conditional statements. They direct messages to appropriate micro-protocols. Virtual protocols are so termed, as they may not have peers and do not add headers to messages.

2.1.2 Discussion

x-kernel was one of the first attempts to structure the implementation of protocols in terms of layers. Micro-protocols promote reuse. Applications can tailor the protocol graph to their needs, though this configuration is possible only when the kernel is built. Experiments with composed protocols have shown that x-kernel protocols perform as well or even better than protocols implemented on less structured systems. The ease of configuration of protocol graphs helps in adjusting to changes in physical network technologies.

Formal verification of micro-protocols was not one of the design goals of x-kernel. The architecture is implemented in the C programming language to make the implementation as efficient as conventional operating systems. This makes the task of formal verification daunting. The control operations permitted by micro-protocols break their uniform interface. x-kernel avoided the problem of composing incompatible micro-protocols by doing the composition at kernel compilation time. Additional care may have to be taken to avoid this problem when micro-protocols are composed dynamically.

2.2 Cactus

2.2.1 Features

Cactus [8] is a framework for constructing highly configurable network services. It supports a two-level composition model. Single function protocol entities called micro-protocols are composed to form composite protocols and composite protocols may be composed with other composite protocols. The inner composition of micro-protocols to form composite protocols is non-linear. The outer composition of composite protocols is hierarchical, similar to x-kernel. A composite protocol may be one unit of an x-kernel protocol graph. Multiple prototype implementations of Cactus in different programming languages and platforms are available.

A micro-protocol is structured as a set of event handlers. Cactus supports a rich set of events and micro-protocols may define their own event types. The framework provides operations for binding a handler to an event, to raise an event, to delete a binding, to order the execution of multiple event handlers bound to the same event etc. When an event occurs, the handlers bound to the event are executed in order. Typically, the order of invocation of event handlers should not affect the function of micro-protocols. However, the event handlers of some micro-protocols may need to be executed before that of others. Such micro-protocols may insert their event handlers at the head of the execution sequence. Event handlers are executed atomically. No other event handler is started while one is executing. Micro-protocols may share data with other components of the same composite protocol. The same micro-protocol may process the same message multiple times depending on the number of events related to or raised by the message.

To avoid introducing an explicit order into the processing of a message and yet assure that all interested micro-protocols of a composite protocol have processed a message, Cactus provides a coordination mechanism using a facility called 'send bits'. Send bits are confirmation flags set by micro-protocols interested in processing a message. A message can be passed to the next higher-level or lower-level composite protocol, only if all send bits appropriate for the message are set. Send bits minimize the assumptions that micro-protocols need to make about other constituents of the same composite protocol.

2.2.2 Discussion

Cactus supports a rich composition model. It does not enforce a strict hierarchical order among micro-protocols. This flexibility helps in cases where the composed micro-protocols are logically at the same level or are totally independent of each other. This composition model allows for complex interaction between micro-protocols wherever they are required. Experiments with prototype implementations have shown that the structured approach of Cactus does not suffer a significant performance penalty compared to conventional implementations.

Formal verification of protocol correctness was not one of the design goals of Cactus. Non-hierarchical composition, the essential feature of Cactus is also the major obstacle to formal verification of Cactus micro-protocols. The dynamic event mechanism coupled with complex interactions between micro-protocols make the task of formal analysis quite challenging.

2.3 CHANNELS

2.3.1 Features

CHANNELS [9] is a run-time system dedicated for network protocol processing. It enables dynamic configuration of protocols out of components. It also supports the development of protocols with real-time and parallel processing requirements. It provides facilities like buffers, timers and mapping tables that simplify common protocol operations like encapsulation, blocking, segmentation etc. It also provides a standard framework for inter-component communication. CHANNELS is implemented in the C++ programming language and is used in the Transport and Internetworking Package (TIP) project.

Protocol components in CHANNELS are called 'modules'. All modules have a uniform interface but may implement different functionality. Instances of modules are created as C++ objects. New module classes can be derived from existing ones using the C++ inheritance feature. The type, number and arrangement of module instances are configurable at run time. Though most arrangements of modules tend to be linear stacks, this structure is not imposed by the run-time system.

Modules communicate with one another indirectly through the ‘channel server’, which is an instance of the run-time system. The ‘channel server’ enables applications to communicate through a ‘channel’. A ‘channel’ is a collection of modules that process and forward all messages exchanged between applications. Information is exchanged between modules in terms of objects called messages. CHANNELS provides standard operations to manipulate message objects. When message objects are passed between modules, their access rights are also passed from the caller to the callee. Operations invoked at the same module are executed sequentially. Operations of different modules may be executed concurrently.

The logical higher-level module for any given module is termed its 'predecessor' and the logical lower-level module is termed its 'successor'. A module communicates with its predecessor using an 'upcall' and with its successor using a 'downcall'. At run time, any module can create other modules, which are termed its 'child' modules. The parent module communicates with its children using 'control' calls. A module may also receive 'events' from the channel server. These four methods along with the creation and deletion operations form the uniform interface for a module.

CHANNELS provides a basic class library that supports the most common protocol operations. This library includes classes for messages, timers and mapping tables. The channel server maintains pre-allocated message pools and timer pools. Modules request messages out of this pool and use them to communicate with neighbor modules. When message objects are deleted, they are returned to the message pool. This minimizes expensive dynamic memory allocation operations. Mapping tables facilitate insertion and lookup of key-value pairs. Different modules may maintain shared state using mapping tables.

2.3.2 Discussion

CHANNELS supports dynamic configuration of protocols. New modules can be inserted into or deleted from a channel at run time. The inter-module communication framework does not enforce any particular arrangement of modules and allows arbitrary arrangements of modules. The basic class library provides utility classes that ease protocol development.

CHANNELS uses the C++ inheritance feature to create new module classes. The module class hierarchy is single-rooted. Though the framework does not prohibit multiple inheritance, none of the modules in the quoted examples derives from more than one super-class. For deep inheritance trees, this limits the reusability of module classes. Though the framework is independent of the granularity of modules, published examples have simple arrangements and the modules used are relatively coarse-grained.

2.4 CogPiT

2.4.1 Features

CogPiT (Configuration of Protocols in TIP) defines a dynamic configuration methodology to build composite protocols from components (modules) [10]. Given a library of protocol components and their formal specifications, the configuration algorithm (COFAL) chooses appropriate protocol components and generates a 'channel template'. The channel template is a description of a list of selected components and the interconnection among them. The CHANNELS run-time system [9] uses the channel template to create instances of selected protocol components and interconnects the components to setup an application-tailored protocol.

A protocol developer formally specifies the functionality of a protocol component and the constraints that must be satisfied for this component to be included in a configured protocol. A custom language called the Module Description Language (MDL) is used for this purpose. The protocol developer also specifies semantically correct protocol adjacencies. This adjacency specification is termed the ‘worldgraph’ and is expressed in MDL as well. The constraints of a protocol component are expressed in terms of symbolic variables for QoS parameters (e.g. maximum latency, acceptable packet loss rate etc.) and Network and System Resources (e.g. total local memory available, maximum network delay etc.).

The application that needs a custom-configured protocol expresses its requirements by specifying desired values for the QoS symbolic variables. The configuration algorithm queries the channel server to determine current values for the 'network and system resources' symbolic variables. Given this information, the configuration algorithm generates a protocol configuration by following an eight-step process. The configuration task makes a distinction between protocol functions (e.g. flow control, error detection etc.) and protocol mechanisms (e.g. window-based flow control, rate-based flow control etc. for the flow control protocol function). Given a request from the application, the required protocol functions are selected first and appropriate mechanisms that implement these functions are chosen next.

2.4.2 Discussion

The concept of formal specification of protocol component functionality and valid component adjacencies and its use in dynamically configuring protocols is most relevant for work on composite protocols. This project provides a language for communication between application developers and protocol developers. It also helps in better design of protocol components, as the protocol developer needs to think of the semantic interface with other components that are valid predecessors or successors of the designed component. The use of symbolic variables to represent QoS parameters and network and system resources is powerful in that it serves as a means for expressing application requirements in precise terms and in choosing the required protocol functions.

The number of QoS parameters is unlikely to remain static. The effect of adding a new QoS parameter has a ripple effect, not just on the configuration algorithm but also on existing specifications. NodeOS support will be required to query current values for the network and system resources. Unfortunately, there are no published examples of protocols configured using this algorithm.

2.5 Netscript

2.5.1 Features

Netscript [11] is an environment and a coordination language for developing composite protocols for active networks. It uses the data flow programming model to structure software on an active node. An active node is represented as a processor of packet streams. It consists of one or more Netscript engines. A Netscript engine is a set of computational elements called boxes, which are interconnected through their input and output ports. A box consists of sequential event-handling code. When a message arrives at an input port of a box, the event handler associated with the input port is executed. Typically, this processing leads to the message being sent out of one (or more) of the output ports of the box. The computation in a box is reactive to the arrival of messages. Messages flow from box to box as they are processed by a Netscript engine. New boxes can be added or removed dynamically from a Netscript engine.

Netscript also defines a coordination language (also called Netscript) to specify the interconnection between boxes. Scripts written in this language describe the constituents of a composite protocol. These scripts are compiled into a set of Java classes that extend the functionality of the Netscript environment called the Netscript Toolkit. The classes in the environment implement the constructs of the Netscript language. The compiled bytecodes of these classes can be deployed in a Netscript engine at an active node, to dynamically extend its functionality. Netscript is not a general-purpose programming language. It provides only a limited set of constructs to represent computational elements and the interconnections between them.

The box construct is the unit of protocol composition. A box consists of the declaration of its input and output ports and the definition of event handling functions for its input ports. A box may be composed of other boxes. In this case, the interconnections between the component boxes are also specified in the outer box construct. The ports of a box are strongly typed. Two ports can be connected only if they have the same signature (same number of arguments and identical type for each argument). One output port can be connected to more than one input port of other boxes. There is no global state or state shared between different boxes. So different boxes may execute concurrently.

2.5.2 Discussion

The data flow programming model is an alternative methodology to structure packet processing software. A rich topology of protocol components is possible with this methodology. Boxes do not export a uniform procedural interface. The Netscript environment checks semantic compatibility of neighbor boxes, by checking the signature of ports of neighbor boxes. Only the Netscript toolkit is currently implemented and a compiler for the Netscript language is not ready at this time. This limits the utility of the Netscript toolkit, because one has to extend the predefined classes in the Netscript toolkit by hand to define new protocols, a task that would have been automated by a compiler.

3. Design of Composite Protocols

Before describing the design of Composite Protocols in detail, let us define some of the terms used in the discussion.

A **protocol component** is an entity that implements a single protocol function. Components typically cannot be used alone. A protocol component may or may not have a uniform interface to its environment, though the exposure of a uniform interface will enable arbitrary composition of components into protocols. Examples of components include fragmentation and re-assembly and reliable delivery. The same protocol function may be implemented by different components using different mechanisms. For example, a reliable delivery component may employ retransmission with a negative acknowledgement scheme and another component may use selective acknowledgements.

A **composite protocol** is a collection of protocol components arranged in an orderly fashion and implements a network communications capability. It can be used as a unit to enable communication between two or more endpoints. It can be tailored to the needs of an application by including only those components essential for that application. Conventional TCP/IP protocol stacks are in fact composite protocols in the sense that they are collections of layers of protocols. However, conventional protocol stacks differ from our definition of composite protocols in terms of the granularity of protocol components (or layers) and in the degree of customizability of protocol functionality. Examples of composite protocols include file transfer and web document retrieval.

A **network service** is a collection of cooperating composite protocols that offer a larger communication service. All composite protocols of a service need not be present and functional at all endpoints of communication. Examples of network services include multicast and web caching. A multicast service comprises of a routing protocol to discover and propagate routes to reach multicast group members, a group membership protocol to keep track of members of a multicast group and forwarding protocols to ensure reliable and secure delivery of multicast content. Each of these protocols can be a composite protocol built from components. A web caching service may use different protocols between a web browser and a proxy server and between the proxy server and a web server.

This thesis only discusses the design and implementation of composite protocols using components. Composition of network services from composite protocols is beyond the scope of this thesis.

3.1 Elements of a protocol component

A protocol component has the following essential elements: a state machine that captures the functionality of the component, classification of memory used by the component into different categories and the properties offered by the component to processed packets.

3.1.1 State machine

Representing a protocol component as a state machine helps in better comprehension of its functionality. The asynchrony of message receptions and a protocol's inherent reactionary nature to messages map naturally to the notion of a state machine. State machines have already been used to model the behavior of conventional protocols. The canonical example is that of the TCP state machine [5]. Other examples of protocols specified using state machines are the Point-to-Point Protocol (PPP) [12] and the Label Distribution Protocol (LDP) [13]. A state machine representation of a protocol component also facilitates automatic validation of specifications and conformance testing of an implementation with respect to its specification.

We use an extended state machine model to represent protocol components. The basic notion of a finite state machine is augmented to include the following three extensions. The extensions were inspired by the concepts of Statecharts [14] and Abstract State Machines [15].

1. Any protocol component is likely to use local memory to maintain state across transitions. For example, a reliable delivery component may save the number of attempted retransmissions of a packet in its local memory so that retransmissions are limited to a maximum. Including the states of local memory objects as part of

a state machine is certain to increase the total number of states of the machine. In our approach, local memory objects are represented as such: variables of a given type. Though each local memory object can be visualized as a state machine on its own, we feel that there are no major benefits in doing so. Our state machine model only includes 'control states': states that indicate the current operational status of a protocol component.

2. State transitions are subject to the evaluation of guard expressions. Guards are predicates with terms from packet headers and local memory objects that should hold true for the corresponding transition to be executable. The introduction of guard expressions minimizes the number of control states (compared to a basic state-machine model in which only one input signal is allowed to change). However, guard expressions may result in the specification of incomplete state machines (no transition is executable for a given set of input conditions). Some measure of discipline in specifying component functionality and automated tools to check for completeness of state machines can alleviate this problem.
3. Code fragments with control structures like branches and loops can be expanded to a state machine in which the branch and loop conditions become guard expressions and transition actions become simple and linear code fragments. However, these state transitions are not triggered by external events but by the entry of the machine into that state. These are called 'synchronous transitions'. Though the addition of synchronous transitions increases the number of control

states, the increase is offset by the benefits of relative ease of formal reasoning of simpler transition actions.

3.1.1.1 Motivation for using Augmented FSMs

We chose to use the concept of an augmented finite state machine for the following reasons.

1. Sequential abstract state machines can simulate any sequential algorithm at their natural abstraction level [16].
2. It is possible to connect the specification of a protocol component to its implementation in a transparent manner by using a set of stepwise refinements [17].
3. Abstract state machines facilitate machine verification of the logical correctness of the specification of a protocol component.
4. They also enable the testing of the conformance of an implementation to its specification.

3.1.1.2 Protocol component as a state machine

The functionality of a protocol component is represented as an augmented finite state machine. The state machine consists of a finite set of states and a finite number of transitions from each state. A state transition has the following elements: current state, next state, event type, guard expression, action function and local memory update function. The terms current state and next state are self-explanatory. The event type denotes the kind of event that triggers the said transition. An augmented state machine reacts to a fixed set of events. Events are asynchronous and are delivered to the state machine by the composite protocol framework (to be discussed in section 3.2). A guard expression is a condition that should hold true for a transition to be executable. It is a predicate with terms from memory maintained by the component and the framework and that from any packets received off the network. The guard expression shall be purely functional and not lead to any side effects. The action function signifies the response of a state machine to the event that triggered the transition. Actions typically result in packets being transmitted over the network or being delivered to other components in the same composite protocol. Action functions use framework primitives (to be discussed in section 3.2.3) to transmit packets on the wire, to deliver packets to higher-level components etc. The local memory update function saves changes to local memory objects. Action and local memory update functions clearly separate the functional and imperative parts of the response of a state machine to an event.

A synchronous transition is activated by the entry of a state machine to a synchronous state. In this case, the state machine does not wait for the occurrence of an event and immediately executes one of the outgoing transitions from this state. All transitions from a synchronous state shall be synchronous. It is recommended that synchronous transitions be used to keep the body of action functions as simple and non-branching fragments of code.

Of all the transitions from a state, the guard expression of one and only one of the transitions shall hold for any occurrence of an event. If none of the guard expressions evaluates to true, the state machine is under-specified. If more than one guard expression evaluates to true, the state machine is ambiguous. Non-deterministic transitions are not allowed. Given these restrictions, the truth table of each of the guard expressions shall be disjoint and the union of that of all expressions shall be exhaustive.

The execution of a state transition is atomic i.e. an ongoing state transition is not interrupted by the arrival of another event. Events meant for a protocol component busy executing a transition are queued and delivered to the component in First-In First-Out (FIFO) order once it is ready. Each protocol component can be thought of being driven by two event queues: one for events due to arrival of packets from the network or a lower-level component and the other for events due to arrival of packets from the application or a higher-level component. The action functions associated with a state transition may lead to a new event being deposited in one of two output queues: one connected to the lower-level component and the other to the higher-level component.

The functionality of some protocol components involves two distinct tasks: processing packets from the component below and from above. If these two tasks are relatively independent of one another, the component may be specified as a pair of state machines. The Transmit State Machine (TSM) exclusively processes packets passed from the component above and the Receive State Machine (RSM) exclusively processes packets delivered from the component below. The two state machines will share the same set of local memory objects. If a component has two state machines, only one of them can be active at any instant of time. Data plane components like reliable delivery and fragmentation and re-assembly are likely to be specified as a pair of complementary state machines.

3.1.1.3 State machine execution mechanics

The steps involved in the execution of a state transition are described in this section. Figure 1 illustrates the flowchart for state machine execution. The diamond symbol labeled 'event type?' stands for a selection based on the event type. The state machine in this figure handles four event types. For every additional event type handled by a state machine, a separate instance of the shaded block labeled 'Transition block' will be part of the flowchart, driven by the selection based on event type.

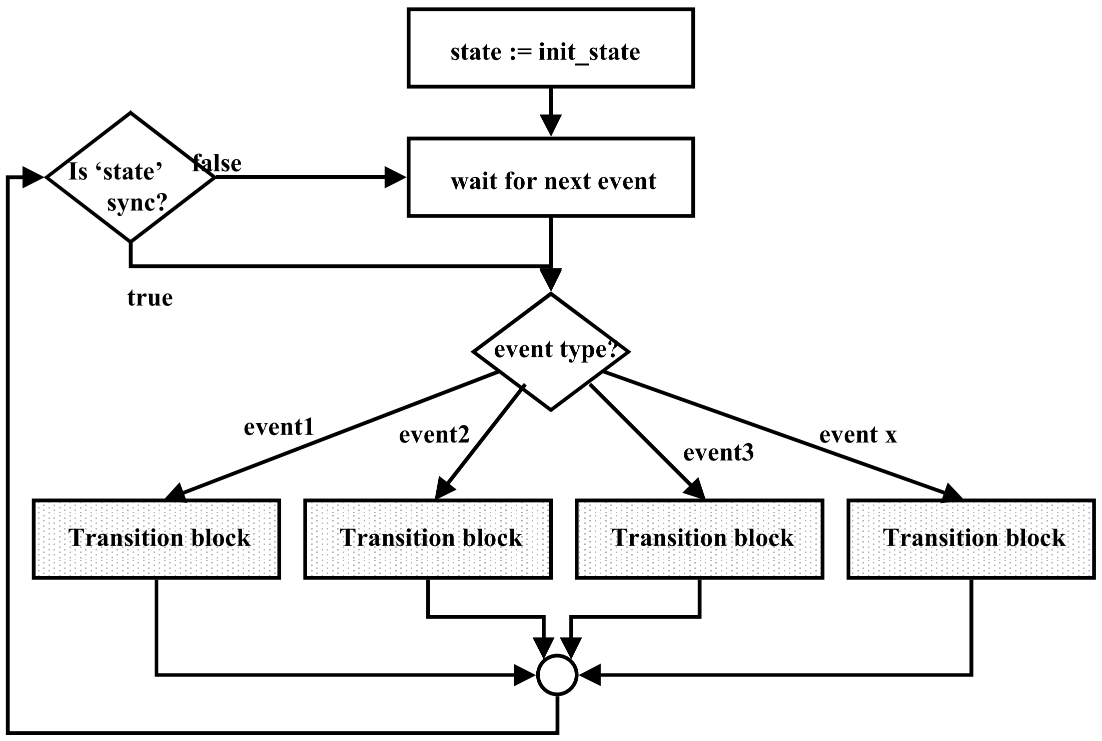

Figure 1: Flow chart of state machine execution

Figure 2 shows the internal structure of the block labeled 'Transition block' in Figure 1. This block shows the sequence of operations that the state machine executor performs for every event that is delivered to a state machine.

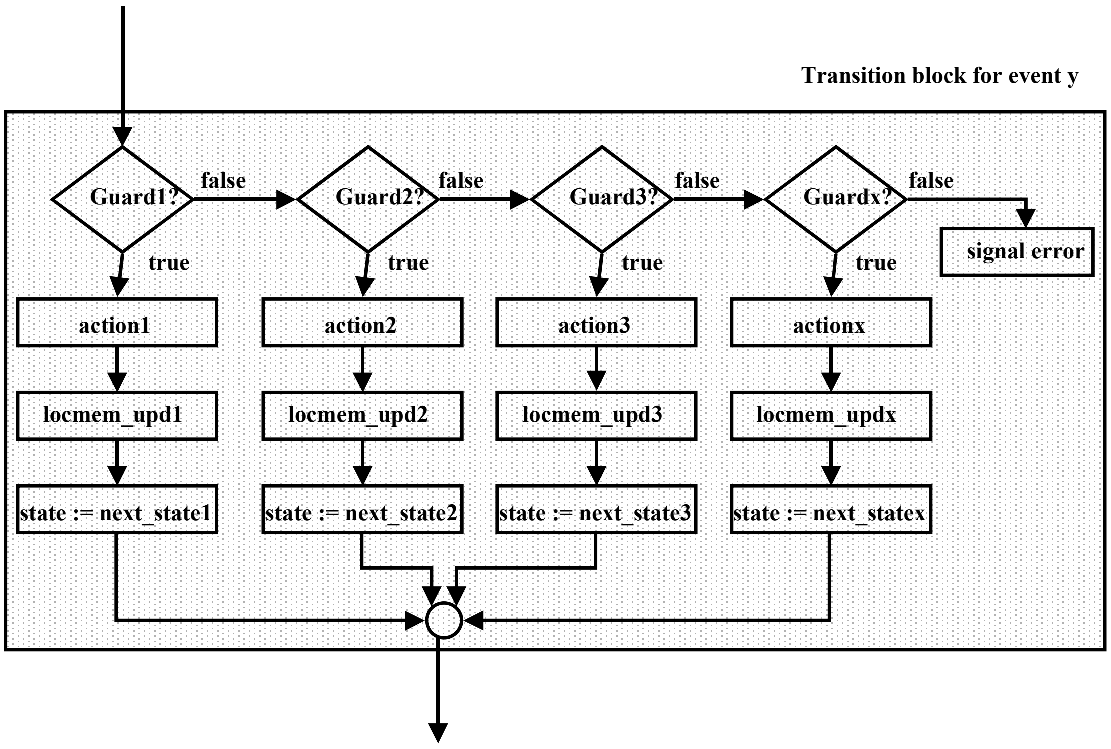

Figure 2: Internal structure of a transition block

The state machine starts with its current state set to the initial state and local memory objects initialized appropriately. When an event occurs, the following steps are carried out.

1. From the list of outgoing transitions for the current state, only those transitions whose event type matches the current event are selected.
2. The guard expression of each of the short-listed transitions is evaluated.
3. The transition whose guard expression holds true is chosen for execution.
4. The action function associated with this chosen transition is executed followed by the local memory update function.

5. The current state of the machine is now changed to the next state of the chosen transition.
6. If the next state is a synchronous state, i.e. a state with only synchronous transitions, then all outgoing transitions are selected.
7. Steps 2 till 5 are repeated until the current state of the machine is not a synchronous state.
8. The state machine now awaits the occurrence of the next event.
9. Steps 1 through 8 are executed atomically and are not interrupted by the occurrence of another event.

If none of the guard expressions of a list of outgoing transitions holds, an exception is raised. An implementation may also choose to evaluate the guard expressions of all outgoing transitions to insure that only one of them holds true and that there are no ambiguous transitions. If the specification of a component was statically checked for ambiguous transitions, the implementation may simply choose the first transition whose guard evaluates to true and execute its associated action and local memory update functions.

3.1.1.4 State machine events

The state machine of a protocol component deals with a limited set of events. The number of distinct event types is restricted to a minimum to ease the task of defining complete state machines: state machines that handle all scenarios of events and guard conditions. There are four possible event types.

1. PKT_TO_NET
2. PKT_FROM_NET
3. TIMEOUT
4. CONTROL

Each of these event types is described in the following sections.

3.1.1.4.1 PKT_TO_NET

The PKT_TO_NET event type corresponds to the arrival of a packet from a higher-level component. The arrival of the packet may have been due to the application or a higher-level component sending a message to its peer. When the application sends a message, a PKT_TO_NET event is generated for the topmost component in the composite protocol. The processing of this event by the topmost component typically leads to the generation of a PKT_TO_NET event for the second component in the sequence. The processing at the second component leads to the generation of the same event type for the next component and so on until a PKT_TO_NET has been generated for the bottommost component. The handling of the event at the bottommost component leads to the transmission of the packet on the wire. If a component sends a peer-to-peer message, a similar sequence of events as described above are generated but starting from the component that is next in the processing sequence.

The PKT_TO_NET event carries two additional pieces of data: the payload as sent by the application and a collection of packet memories added by higher-level components. As each component handles a PKT_TO_NET event, a new PKT_TO_NET event is generated for the next lower-level component, but the payload and packet memories from the previous event are handed over to the new event. If a component is specified as a pair of state machines, then a PKT_TO_NET event is exclusively delivered to the TSM.

3.1.1.4.2 PKT_FROM_NET

As the name suggests, the PKT_FROM_NET event type is the complement of the PKT_TO_NET event type. It corresponds to the arrival of a packet from a lower-level component. The arrival is ultimately due to the reception of a packet off the network. When a packet is received from the network, a PKT_FROM_NET event is generated for the bottommost component. The processing of this event recursively leads to the generation of the same event type for every component in the sequence bottom-up. PKT_FROM_NET and PKT_TO_NET are two event types that are most often delivered to a typical state machine. If a component has a pair of state machines, the PKT_FROM_NET is always delivered to the RSM. A PKT_FROM_NET carries the same two pieces of data as the PKT_TO_NET event.

3.1.1.4.3 TIMEOUT

A TIMEOUT event signifies the expiration of a preset timer. Framework primitives are provided to set and cancel timers. When the preset timer goes off, a TIMEOUT event is generated and delivered only to the state machine that requested the timeout. Every TIMEOUT event carries an integer identifier. The semantics of the identifier is decided by the component that uses timeouts. The value of the identifier is set when a timer is started and is transparently returned as part of the TIMEOUT event. A reliable delivery component may use the sequence number of an unacknowledged packet as the timer identifier, so that the appropriate packet can be retransmitted when the TIMEOUT event is generated. Periodic transmission of 'Hello' packets by a neighbor discovery component is another example of a component using timeouts.

3.1.1.4.4 CONTROL

Control events are used as a means of inter-component communication and are explained in detail in section 3.1.5.

3.1.2 Memory classifications

Protocol components use memory objects to save state across transitions. Strictly speaking, the state of a protocol component includes the state of the memory objects as well. However, memory objects are treated as disjoint from the control states of the state machine. This separation reduces the number of states of the machine and eases the task of formal reasoning. The memory objects used by a component are classified based on their scope and extent into four types.

1. Packet Memory
2. Local Memory
3. Stack-local Packet Memory
4. Global Memory

The classification of memory based on their scope and extent is one of the highlights of this design. Specification of components using this classification makes their internal and external dependencies explicit.

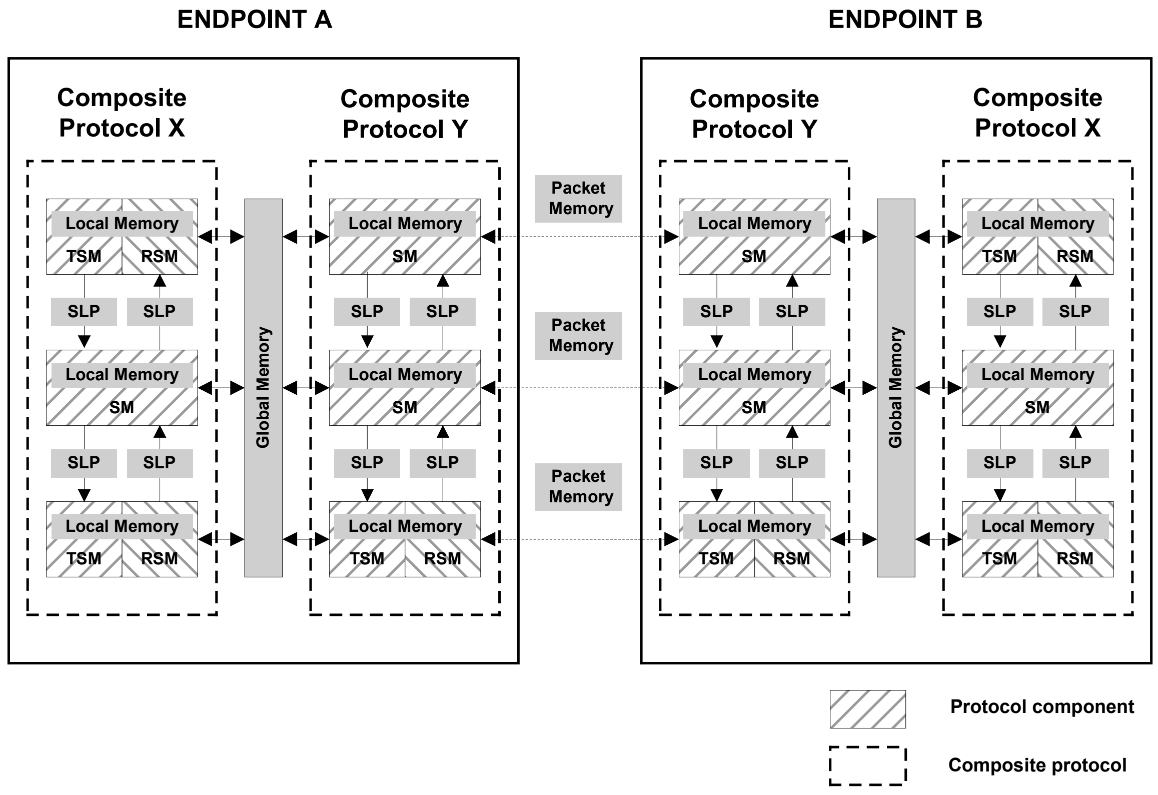

Figure 3: Classification of memory used by a component

Figure 3 illustrates the four types of memory objects used by a protocol component. The figure shows a network service built from two composite protocols X and Y, operating at endpoints A and B. Only the packet memories used by components of composite protocol Y are shown, though components of the other protocol may also use packet memory. The figure also hints at the possibility of some components being specified as a single state machine and some other as a pair of state machines.

3.1.2.1 Packet memory

Packet memory is the term used to indicate header fields attached to packets. It is categorized as a kind of memory, because it transfers state information across peer components. The type of packet memory is defined by the specification of a component and is visible only to that component and its peers. The extent of packet memory is the same as that of the packet to which it is attached. There are no issues related to concurrent access because a packet is processed by only one component at a time and packet memory is private to the component that defined it. For some protocol functions like encryption and error detection, it may be necessary to access packet memories attached by other components of the same composite protocol. The set of packet memories attached to a packet is available as a read-only array of bytes to meet this requirement. The internal structure of packet memories is transparent to other components. Examples of packet memory include sequence numbers tagged to packets by reliable delivery and in-order delivery components, checksum computed by an error-detection component and fragment numbers added by the fragmentation and re-assembly component.

Components are allowed to attach different types of packet memories to different packets. For example, the fragmentation component may tag a single boolean identifier to an unfragmented packet to indicate that it is not fragmented and may add additional information about fragment numbers and payload identifiers to packet fragments. This requirement implies that the format of packet memory is not fixed for all packets and the type of packet memory shall be carried as part of it.

3.1.2.2 Local memory

Local memory is the set of memory objects that are limited to an instance of a protocol component. The type of local memory is defined by the component's specification. Local memory objects exist as long as the component instance that defined them. Access to local memory objects is limited to the instance of the component that defined them. If a component is specified as a pair of state machines, the same instance of local memory objects is shared by the transmit and receive state machines. To avoid issues with concurrent access, only one of the transmit and receive state machines can be busy executing a transition at any instant of time. Each state transition includes a dedicated function in which the status of local memory objects are updated. Examples of local memory include unacknowledged packets buffered by the reliable delivery component, fragments of payloads to be reassembled saved by the fragmentation component and identities of neighbors maintained by a neighbor discovery component.

3.1.2.3 Stack-local packet memory

Unlike packet memory that is a means of communication between peer components in different composite protocol instances, stack-local packet memory (SLPM) is used to enable communication between components in the same instance of a composite protocol. SLPM is a set of attributes attached to a packet as it traverses a sequence of components in a composite protocol. It is primarily used by components to influence the processing of a packet by another component in the same protocol. It may also be used as a convenient placeholder for some information that is later used by the same component. Access to stack-local packet memory is limited to the component that is currently processing the associated packet. Since a packet is processed by only one component at a time, there is no scope for simultaneous access. All components of a composite protocol can access stack-local packet memory associated with a packet, but only one component at a time. SLPM is a means of communication within a composite protocol and is never transmitted on the wire.

Stack-local packet memory is carried with the packet as an association list of <name, value> pairs. Framework primitives are provided to set and get the value of a named SLPM field. SLPM being a means of communication between components, it is possible to erroneously configure a composite protocol that includes a component that reads a named SLPM field but does not include a component that sets the same field. Static analysis of components at composition time is suggested to avoid configuring incomplete protocols.

An example of an SLPM field is the next hop network address tagged to a packet. Forwarding a packet towards its destination involves two distinct tasks: finding the network address of the next hop to the destination and sending the packet to the next hop through appropriate physical media. The first step is a lookup of the destination network address in the forwarding table to find the network address of the next hop. The second step includes finding the physical address of the chosen next hop and sending the packet to the next hop. The first step is independent of the physical media used between this endpoint and the next hop whereas the second step is not. Separating these two tasks as distinct components facilitates the reuse of the 'next hop lookup component' in any composite protocol regardless of the physical media used. When the forwarding function is thus split into a 'next hop lookup' component and a 'network address to physical address mapping' component, it is essential to tag the network address of the next hop to every packet being forwarded. An SLPM field named NEXT_HOP_ADDR is used for this purpose. This SLPM field is set by the 'next hop lookup' component and read by the 'address mapping' component.

Another example of usage of an SLPM field is to carry the Time-To-Live (TTL) value of a received packet. When a 'forwarding loop mitigation' component of a composite protocol at a router receives a packet with a TTL value in packet memory, the memory is stripped and the packet is delivered to a higher-level component. When the same packet is to be forwarded to the next hop, the new TTL value to be added shall be one less than the received TTL value. This requires that the received TTL value in packet memory be saved, though the packet memory instance itself is removed. SLPM serves as a convenient placeholder for this information. The received

TTL value is tagged to the packet as an SLPM field before it is delivered to a higher-level component so that the TTL is correctly decremented, when the packet is forwarded.

The facility of SLPM shall be used with care. SLPM shall only be used in scenarios where there is a one-to-one correspondence between an SLPM writer and an SLPM reader. This restriction, though not currently enforced, eases the task of insuring that composite protocols are always configured completely. For example, a component that sends a peer-to-peer message using the primitive 'NewPktSend' may add an SLPM field named DO_NOT_ENCRYPT which is a hint to the Encryption component not to encrypt a packet with this SLPM field set. This component assumes that there will be an encryption component in the same protocol and that the encryption component would have been designed to recognize this SLPM field. This is a scenario where there is no clear one-to-one correspondence between the writer and reader of an SLPM field. If one has to satisfy this restriction, this component may only be used in protocols that also include an encryption component, thus limiting its applicability.

3.1.2.4 Global memory

Global memory is memory shared by more than one composite protocol operating at an endpoint. Any component of any composite protocol operating at the endpoint is allowed access to global memory objects. Definition of global memory objects requires agreement between components of different composite protocols. The specification of a component shall make all dependencies on global memory explicit. The extent of global memory objects is not limited to the lifetime of individual composite protocols. These objects exist as long as the endpoint is operational. This requires that every endpoint have a dedicated subsystem to manage all global memory objects. Access to global memory objects is provided through a functional interface. This interface is also responsible for resolving issues of simultaneous access. The interface insulates the dependent components from changes in the actual format of global memory objects.

Introduction of global memory raises many challenges for the formal verification of composite protocols. It is impossible to check at component development time or protocol composition time if there are any inconsistencies in configured protocols. The protocol that writes to a global memory object may not even be operational when the protocol that reads the same object is started. Further work needs to be done to address the issues of inconsistencies in composite protocols, due to dependencies on global memory objects.

An example of global memory is the routing table maintained on routers. A composite protocol responsible for routing creates and maintains the routing table and a forwarding composite protocol looks up destinations in the routing table. Another example is the multicast group membership table maintained on leaf routers. A group membership protocol creates and maintains this table. A multicast routing protocol may read the entries in this table to send prune messages to inhibit the transmission of unwanted multicast data packets.

3.1.3 Parameters

Parameters are used to tune the operations of a protocol component. Parameters are passed to a component at initialization time and may be used by the component to pre-allocate appropriate amounts of memory, choose a particular algorithm, set limits on specific protocol operations etc. Examples of component parameters include the maximum number of retransmissions to be attempted by a reliable delivery component, the maximum degree of disorder to be tolerated by an in-order delivery component, the maximum amount of buffer space to be used by a fragmentation and re-assembly component to save fragments etc. Some of the component parameters may also be specified at the command line of the application using a composite protocol when it is invoked.

Parameters increase the complexity of configuration of protocols. It is possible that peer components of a protocol are initialized with different values for their parameters. This may lead to incompatibilities in their behavior. It is recommended that parameters be used judiciously.

3.1.4 Properties

Protocol components may provide formal properties to the packets they process. Properties are formal assertions about a particular condition being true for a packet or for a sequence of packets. An example of a property for a single packet is the guarantee of bit-error free transmission of packets offered by an error-correction component. The guarantee of in-order reception of packets is a property that pertains to a sequence of packets. The formal properties offered by a component are not absolute and are often subject to restrictions due to practical reasons. For example, a reliable delivery component may guarantee reliable delivery of packets subject to the condition that no packet will be retransmitted more than the maximum number of retransmissions and packet losses of greater magnitude will not be compensated.

A component may require that a property hold at the interface between itself and a lower-level component to guarantee its correct operation. A simple example for a requirement is the error-free reception of packet memory fields. No component may operate as specified if the received packet memory fields are in error. A component may provide a property at the interface between itself and a higher-level component. Guarantees of error-free reception of packets and in-order delivery of packets are examples of properties provided by components. A component may guarantee that it does not alter a particular property of a packet or a sequence of packets. For example, a decryption component may preserve the property of orderly delivery of packets. A property is said to be invariant across a component if the component preserves the property for all packets that it processes.

Properties offered by a component are usually specified in a formal language like the Temporal Logic of Actions (TLA) [18]. One of the main purposes of specifying protocol functions as modular components is to make the task of formal reasoning manageable. Given the specification of a component, it shall be possible to make assertions about the properties that it offers. As a further improvement, it shall also be possible to automatically configure a composite protocol that includes components providing the properties requested by a user. The task of formal reasoning about the properties offered by components is beyond the scope of this thesis.

3.1.5 Control Interface

3.1.5.1 Motivation for Control Interface

Protocol components as defined in section 3.1.1 are said to offer a "Packet-in Packet-out" (PIPO) interface. The component gets a packet from a higher-level component, processes it, adds an instance of packet memory and passes it to a lower-level component. On receiving a packet from a lower-level component, the instance of packet memory attached to the packet is removed, the packet is processed and is delivered to a higher-level component. The PIPO interface is uniform and completely serves the needs of data plane components like fragmentation and re-assembly and reliable delivery. Data plane components usually form the lower levels of a composite protocol.

Implementing higher-level protocol functions as components requires an interface with richer semantics than what is possible with the PIPO interface. Consider the case of realizing file transfer as a component. Implementing file transfer as a component instead of as an application promotes its reuse in other composite protocols like web caching that also need to perform file transfer tasks. A file transfer utility similar to the Unix 'ftp' tool can be built using a composite protocol that includes a file transfer component. This application will implement the minimal functionality of parsing command-line arguments; creating and sending control commands to the file transfer component. All file transfer tasks will be delegated to the component. If the same component is included in a web caching composite protocol, the web-caching component shall create the appropriate control commands and pass them to the file transfer component. This scenario requires a means of control communication between the application and a component or between two components in the same instance of a composite protocol.

It is possible to overload the semantics of the PIPO interface to achieve this form of control communication. The application may create a fake packet that includes control commands and pass it to the controlled component. The component may then parse the contents of the fake packet and respond accordingly. This extension of the

PIPO interface drastically alters the definition of a packet and essentially hides a semantically peculiar interface behind a syntactically uniform interface. Extra care may have to be taken to insure that application data is never misinterpreted as a control command. Providing a dedicated control channel for communication between components leads to a more elegant design.

3.1.5.2 Design of control interface

Control communication is modeled as an exchange of messages between a pair of components or between the application and a component. The component being controlled offers a service (not to be confused with a network service built from multiple composite protocols). The controlling component or the application utilizes this service by sending commands to the controlled component.

The control communication consists of the following major parts.

1. Framework support for invoking services offered by components.
2. Specification of a service by the controlled component.
3. Invocation of a service by the application or another controlling component.

3.1.5.2.1 Framework support

A new event type, namely CONTROL event is introduced to carry commands from the source (controlling) component to the destination (controlled) component. The CONTROL event is delivered to the state machine of the destination component in the same manner as other events like PKT_TO_NET and PKT_FROM_NET. Control events are passed between components using the same FIFO queues used for sending other events. This maintains the relative order between control events and other events generated by the same component. Control events are a means of communication within a composite protocol and are never transmitted on the wire. Framework primitives are provided to create a control event and to send a control event up or down the composite protocol. An instance of a control event has a placeholder for a service request (to be introduced in the next section).

The introduction of a new event type increases the complexity of state machine definitions. Each state should have at least one transition for a control event. However, some optimizations are possible in reducing the number of transitions. Components that do not participate in control communication need not define transitions for control events. Even components that do provide a service only have to take care of handling all commands that are part of the offered service and not define transitions for other services.

3.1.5.2.2 Service specification

The specification of a service offered by the controlled component has the following elements.

1. Service interface.
2. Service implementation.

The service interface defines the syntax and semantics of legal commands that are part of the service. This definition includes the following information.

1. Service name.
2. List of names of commands.
3. Additional parameters for each of the commands.

The service name uniquely identifies a service. Though more than one component in a library of components may implement the same service, only one of these components shall be included in a composite protocol. Services that are offered by components in a particular composite protocol shall be distinct. In turn, each service may have many possible commands. At any time, the controlled component shall be ready to accept and act upon any one of the available commands for a given service. The namespace of commands is limited to a service and need not be unique across services. Each command may optionally carry additional parameters. The specification of the service interface defines the number and type of additional parameters for each command supported by the service.

The service implementation, being part of the functionality of the controlled component, is realized as a state machine. A service request is delivered to the destination component as part of a control event. The controlled component shall have defined transitions for all possible commands of the offered service. The response of the component to a service request is codified as part of the action and local memory update functions associated with state transitions that handle control events.

3.1.5.2.3 Service invocation

When a controlling component or the application needs to use a service, it creates an instance of the corresponding service request. One of the commands is selected and additional parameters for this command, if any, are initialized. A control event is created and the instance of the service request is tagged to the control event. A framework function is then invoked to send the control event up or down the composite protocol. The framework then performs the necessary steps to deliver this control event to the state machine of the controlled component.

3.1.5.3 Application of Control Interface

This section illustrates the application of the control interface to create smaller protocol components. A file transfer application has been specified and implemented by leveraging this control interface. File transfer involves two distinct functions: Control of file transfer and the actual transfer of files. The specification of file transfer as a single component will lump these two functions together in ways that would make reuse of only one of these functions difficult. Instead, the file transfer application has been structured as a lean application that uses two new protocol components, FTP control and FTP data. The application is lean in the sense that it serves only as a parser of command line arguments and has no file transfer functionality of its own. The FTP control component is responsible for accepting FTP commands from the application and managing the file transfer process by talking to its peer component at the other endpoint. The FTP data component is responsible for reading files from the local file system and sending them on the wire and vice versa.

This design opens many avenues for reuse. The FTP data component alone can be reused in other composite protocols used for data logging and streaming audio and video. The FTP control and data components together can be used by other protocol components like web caching to delegate their file transfer tasks. More innovative variants of protocols are now possible. For example, an encryption component can be placed between FTP control and FTP data components in a composite protocol to insure the privacy of only the FTP control commands while leaving the contents of transferred files unprotected.

3.2 Framework support

Components of a composite protocol are held together by a framework. The framework provides the infrastructure for composition and proper operation of protocol components. It drives the state machines of individual components by delivering events, manages multiple events destined for the same state machine by queuing them and provides a rich set of primitives for components to invoke its services.

3.2.1 State machine execution

Reactive systems like finite state machines require a driver to execute them. The composite protocol framework includes a state machine executor. The executor has the following responsibilities.

1. Convert packet transmissions by an application and receptions from a network into appropriate events for the state machine.
2. Trigger and execute a state transition. This includes the evaluation of all necessary guard expressions and invocation of action and local memory update functions.
3. Provide support for the notion of synchronous transitions by repeatedly executing transitions until the state machine reaches a non-synchronous state.
4. Keep track of the current control state and the current state of local memory objects of a component.

3.2.2 Event queue management

A state transition is guaranteed to be executed atomically (i.e. not interrupted or pre-empted by the occurrence of another event). Events destined for a state machine busy executing a transition are queued and processed in FIFO order. The framework manages the FIFO event queues. Conceptually, event queues are infinite in length, though practical limitations may restrict the number of elements in the queue. The FIFO property of event queues is crucial for proper operation of many protocol components. For example, an in-order delivery component cannot provide the in-order delivery property, unless packets delivered by this component reach the next higher-level component in the same order. If control commands from the application or another component do not reach the controlled component in the same order, the controlling component may not receive the exact service that it had expected. To avoid the buildup of a backlog of events, action and local memory update functions of state transitions shall be simple and short execution sequences. The idea of synchronous transitions shall be judiciously used to avoid long or infinite processing loops.

3.2.3 Framework primitives

The composite protocol framework provides a rich set of functions for components to invoke its basic services. These functions are broadly classified into the following categories.

1. Packet transfer
2. Timer management
3. Buffer management
4. SLPM access
5. Control communication

The functions in each of these categories are described in the following sections. Unless otherwise noted, these primitives can be invoked from the action functions of any state transition and for components specified as a pair of state machines, from either the TSM or the RSM.

The primitives as defined in this section are the specifications of functions supported by the framework. When the primitives are implemented, the signatures of these functions may have to be extended or modified to accommodate implementation considerations. For example, polymorphic SLPM access functions may have to be replaced with concrete functions with fixed type signatures, one for every SLPM field.

3.2.3.1 Packet transfer

procedure PktSend(PktMemory)

The invocation of this function leads to the generation of a PKT_TO_NET event for the next lower-level component. This primitive can only be invoked from the action function of a state transition that handles a PKT_TO_NET event. When a component is specified as a pair of state machines, this can happen only in the TSM. The payload and packet memories associated with the current PKT_TO_NET event are handed over without modification to the newly generated event and an instance of the packet memory for this component is also tagged to the new event. Invocation of this function by the lowest-level component of a composite protocol leads to the packet being transmitted on the wire.

procedure PktDeliver()

This is the complement of the PktSend function. It leads to the generation of a PKT_FROM_NET event for the next higher-level component. When this function is invoked by the highest-level component in a protocol, it is delivered to the application. The packet payload is passed along without any modification to the new event. This function can only be invoked from the action function of a state transition that handles a PKT_FROM_NET event. For components with a TSM and a RSM, this can happen only in the RSM.

procedure NewPktSend(PktMemory)

This primitive creates a new peer-to-peer message and triggers the generation of a PKT_TO_NET event for the next lower-level component. It is remarkable that there is no payload attached to this message, because the message is created by a component and not by the application. The message contains only an instance of the packet memory of the invoking component.

procedure NewPktDeliver(PktPayload)

This function delivers a new PKT_FROM_NET event to the next higher-level component along with the newly reconstructed packet payload. This primitive shall only be used if the sender side of the invoking component modifies the payload. For example, fragmentation and encryption components modify their payloads on transmission. It is important to note that the primitives PktSend and PktDeliver cannot be used to carry altered payloads. A payload once modified by a component on the sender side (for example, divided into fragments) does not make sense as such to the higher-level components of its peer. It is understood only by the peer component. It is suggested that modified payloads be carried as packet memory attached to a peer-to-peer message sent from the component that performs the modification. On reception of this message, the peer component may reconstruct the original payload and use NewPktDeliver to deliver the same to its higher-level component.

3.2.3.2 Timer management

procedure SetTimeout(timerid : int, timeout : long)

This function is used to start a timer that will expire 'timeout' seconds from current time. When the timer expires, a TIMEOUT event is generated and delivered to the state machine that invoked this primitive. The timer ID is a convenient parameter that is returned transparently as part of a TIMEOUT event. This identifier can be used by components to associate a particular timeout with a packet. For example, the sequence number of an unacknowledged packet buffered by a reliable delivery component may be passed as timer ID. When the timer expires, the sequence number as returned by the TIMEOUT event can be used to retransmit the appropriate packet.

procedure CancelTimeout(timerid: int)

The function requests that the timer identified by 'timerid' be cancelled.

3.2.3.3 Buffer management

The buffer management primitives offer convenient mechanisms for components to buffer packets. These primitives are heavily used by the reliable delivery and in-order delivery components.

function KeepPacket() : PktRef

This function buffers the packet whose arrival caused the current state transition and returns an opaque object that represents the buffered packet. This function shall only be invoked from action functions of state transitions triggered by PKT_TO_NET and PKT_FROM_NET events.

function KeepPacket(NewPktMemory) : PktRef

This function buffers the packet whose arrival caused the current transition after adding or replacing the current packet memory with the passed instance of packet memory. This function is useful to buffer packets along with the instance of packet memory that would have been tagged to the packet whenever it is transmitted.

procedure DeliverKeptPacket(thePkt : PktRef)

This function passes the buffered packet identified by 'thePkt' to the next higher-level component. After the invocation of this function, the calling component shall not use the same PktRef value again.

procedure TransmitKeptPacket (thePkt :PktRef)

This function passes the packet identified by 'thePkt' to the next lower-level component. The packet is still buffered by the framework and hence the value of 'thePkt' is still valid. This function is used to retransmit unacknowledged packets by the reliable delivery component.

procedure DropKeptPacket (thePkt : PktRef)

This function drops the buffered packet identified by 'thePkt'. The memory allocated to the buffered packet is freed and the reference 'thePkt' is no longer valid.

3.2.3.4 SLPM access

Fields of Stack-local packet memory are conceptually represented as an association list of <name, value> pairs. The following functions are used to set and get the value of an SLPM field. These functions shall only be invoked from the action functions of state transitions that handle PKT_TO_NET and PKT_FROM_NET events.

function getSLP(name) : value

This function returns the value of the SLPM field tagged to the current PKT_FROM_NET or PKT_TO_NET event, whose name is passed as a parameter. The name of an SLPM field is an enumerated constant and the type of its value depends on the definition of the field.

procedure putSLP(name, value)

This function sets the value of the SLPM field, attached to the current event, whose name and desired value are passed as arguments.

3.2.3.5 Control communication

procedure SendUpControlEvent(event)

procedure SendDownControlEvent(event)

These two functions send the passed control event to the next higher-level and the next lower-level component respectively. The control event passed as a parameter shall be initialized with exactly one service name and one command name for that service and any additional parameters appropriate for the chosen command.

3.3 Identification of components

One of the best methods to identify protocol components is to analyze the functionality of existing protocols. In this regard, it is instructive to look at protocol header fields. Though not all functions of a protocol can be identified from header fields, this method offers a good starting point. For example, header fields of the Internet Protocol (IP) [19] were grouped based on their functions. This grouping led to the identification of the functions of fragmentation and re-assembly, routing loop mitigation, error detection etc. However, one shall be careful in inferring protocol functions from header fields because in some cases the same header field may serve the purpose of more than one protocol function. The sequence number in the TCP header is an example of a multi-purpose header field. The same sequence number is used for reliable delivery and in-order delivery. The specifications of protocols shall also be analyzed to find distinct protocol functions.

One needs to make a distinction between a protocol function and mechanisms used to implement a function. An error-detection component may be implemented using parity bits, checksum calculation or Cyclic Redundancy Codes (CRC). It is suggested that each mechanism be implemented as a distinct component instead of a single error-detection component with all mechanisms. This keeps the functionality of a component simple. Moreover, configuring a protocol requires choosing only the appropriate mechanism and not selecting options of a chosen protocol component. This minimizes the complexity of the configuration step. The suggestion to avoid component options even applies to implementing the same mechanism for different usage scenarios. A fragmentation and re-assembly component that operates over high data rate, high-bandwidth links may have to use larger numberspaces for packet memory fields and may require more memory to buffer fragments than a component that works over low-speed links. If the algorithms used to implement the two cases are substantially different, it is better to implement them as two distinct components. The appropriate component can then be chosen based on expected usage.

It is not our aim to specify components so that composite protocols built out of them will inter-operate with conventional networking protocols. The goal is to specify and implement components with similar functions as conventional protocols. The components may use different packet memory fields and hence may not inter-operate with implementations of current protocols.

3.4 Methods of composition

Given a library of protocol components and a list of desired functions, there are many methods to arrange components in an orderly fashion and form a composite protocol. This section discusses two major approaches to composition of composite protocols and justifies the selection of one of them.

3.4.1 Type-Length-Value (TLV) approach

The format and arrangement of the header fields of conventional protocols are static and are defined at specification time. A given header field is identified by its offset from the start of the header bits and its length. This static arrangement diminishes the customizability of a protocol. To enable arbitrary composition of components, the header fields of components shall identify themselves. The encoding scheme proposed by the Basic Encoding Rules (BER) [20] of the Abstract Syntax Notation One (ASN.1) shall be used for every header field. A header field is represented as a tuple of <type, length, value>. Each header (or packet memory) field shall be given a distinguished name. The length of the header field in bytes and the actual value of the header field form the other members of the tuple. The benefits of arbitrary composition of components outweigh the extra bits required to represent header fields in TLV notation.

Before a packet is transmitted, each protocol component shall add one or more TLV tuples to the packet payload. There is no explicit sequence in which a packet is processed by components. TLV tuples that are added to a payload by a protocol component are visible to all other components that process the same payload. This simplifies the implementation of protocol functions like error detection, encryption etc. Two independent components may even be allowed to process the same packet in parallel.

On receiving a packet from the network, the protocol components that need to process this packet are identified from the list of TLV tuples carried by the packet. Each required protocol component is then allowed to process the packet payload along with the TLV tuples for which it is responsible. Parallel processing of the same payload by independent components is also allowed.

This approach offers a flexible solution to represent packet memory fields. However, many protocol functions require that a packet payload be processed in a particular sequence. For example, the fragmentation component cannot process a packet until all other components have added their TLVs to the payload. Otherwise, the resultant packet may still exceed the Maximum Transmission Unit (MTU) of the physical medium. The same restriction also applies to an error-detection component that protects all header fields. Specifying the sequence of expected protocol-processing steps requires more information than a set of TLVs found in a packet. Moreover, the task of formal verification is complicated by the possibility of parallel processing of a payload.

3.4.2 Linear stacking approach

As per this approach, protocol components are arranged in a linear sequence to configure a composite protocol. This is an extension of the well-known encapsulation method. Messages originated by an application are processed by a fixed sequence of components. Each component is allowed to access the message payload. Components may add an instance of their packet memory to the processed payload. Components cannot modify the payload or packet memories added by other components. The packet memories added by components that have already processed this payload are available as a read-only sequence of bytes with no visibility of their internal structure. Access to packet memories of other components caters to protocol functions like fragmentation and re-assembly, error detection, encryption etc. After the message is processed by the last component in the sequence, it is transmitted on the wire.

When a message is received off the network, it is processed by the same sequence of components but the order of processing is the reverse of that on the sender endpoint.

Each component is allowed access to the payload and the instance of packet memory added by its peer component. When the component finishes processing the packet, the instance of packet memory is removed and the rest of the packet is delivered to the next component in sequence. When the last component in the sequence completes processing the packet, it is delivered to the application.

All messages from the application are processed by the same sequence of components. This implies the absence of multiplexing capabilities in the composite protocol. The absence of multiplexing does not have major disadvantages because of the customizability of composite protocols. Each application can configure a composite protocol with exactly the functions that it needs. Moreover, an application can use more than one composite protocol for communication. On reception of packets, demultiplexing is limited to finding the appropriate instance of the composite protocol to which the packet shall be delivered for processing.

The sequence of components that form a composite protocol is decided at protocol-configuration time. A composite protocol is configured when the application that needs to use one starts execution. Once configured, the sequence remains unchanged during the operation of the protocol.

Each component has two adjacent components, one on a higher level and the other on a lower level. The event queues of adjacent components are connected during composition of the protocol so that packet transmission from a higher-level component results in a PKT_TO_NET event being sent to the lower-level component and vice versa. A component may not be interested in processing every packet from the application. In that case, the component may simply pass the packet onwards to the next component in the processing sequence, after adding an empty header.

Due to the high degree of customizability of composite protocols, care shall be taken to ensure that communicating endpoints use protocols with identical sequence of components. The name of a protocol is derived from the sequence of its constituent components. Two communicating endpoints shall have the same names for their composite protocols. The need to match names of composite protocols has interesting ramifications when routers are part of the communication. If a packet is forwarded by a composite protocol running on intermediate routers, it is necessary that the sequence of components in that protocol shall also match that in the endpoints. However, this restriction can be relaxed to an extent. The forwarding component in a composite protocol is at the highest level that a packet being forwarded will reach. It is sufficient that the sequence of components below (and including) the forwarder be identical at router hosts and at endpoints.

The need to match component sequences also dictates that asymmetric protocol functionality be implemented as a single component. For example, the functionality of file transfer is asymmetric at the client and server endpoints. If an endpoint is always configured as a server, it is sufficient to include only the server functions of the asymmetric component. This implies that the sequence of components is different at the server and client. Though this difference in the processing sequence is acceptable, it is difficult to distinguish such a case from that of incorrectly configured composite protocols with mismatched component sequences. Hence, it is suggested that asymmetric functions be realized as a single component.

The simple linear arrangement of components along with FIFO event queues between them offers many advantages for formal reasoning of protocol correctness. The fixed sequence of processing a payload by components minimizes the number of possible execution sequences that a payload may be subjected. It is sufficient to prove that properties hold for the only possible execution sequence. The composition method also minimizes the number of assumptions made by a component about its neighbors. Dependencies on the presence of other components in the same protocol are brought out explicitly in the specification.

4. Implementation of Composite Protocols

Ensemble [21], the group communication system developed at Cornell University was chosen as the basis for implementing composite protocols. This section explains the reasons for selecting Ensemble, briefly describes its relevant features and discusses the implementation of a finite state machine executor over Ensemble.

4.1 Reasons for choosing Ensemble

Ensemble has several features that make it a good choice for implementing composite protocols. It is implemented in OCaml [22], which is a dialect of the functional programming language ML. Use of a functional language improves the chances of formal reasoning of the written code. In fact, some results have already been reported on the formal analysis performed on Ensemble, using the verification tool Nuprl [21].

Ensemble supports linear stacking of protocol components, the same composition operator that we chose for building protocols out of components. All messages are processed by a fixed sequence of protocol layers. Each layer has exactly one higher-level layer and one lower-level neighbor. Ensemble event handlers are executed atomically. Ensemble also implements unbounded event queues between adjacent components. The uniform interface exposed by Ensemble layers, in the form of up and down event handlers, enables arbitrary composition of layers to form protocols.

These desirable features make Ensemble the best possible choice for the composite protocol framework, other than implementing one from scratch.

Ensemble was primarily designed as a group communication system and hence includes many features that are not required to implement general-purpose network protocols. The Ensemble system was modified to remove some of the features related to group communication. The modifications have resulted in a smaller code-base for the composite protocol framework but the essential features have been retained.

4.2 Overview of the design of Ensemble

Protocol stacks in Ensemble are composed of single function 'layers'. A process can create multiple protocol stacks. A protocol stack is always part of a group at any instant of time. Upon creation, a stack is part of a singleton group. In course of time, singleton groups merge to form larger groups. A stack may send a point-to-point message (Send) to another member in the group or may send a message (Cast) to all members of the group. Each stack has a local data structure called the view, which holds the names and addresses of all members of the group. Most Ensemble layers have been designed to work with a single immutable view data structure. Whenever a new member joins a group, all members of the group configure a new instance of their protocol stacks that uses an updated view data structure. This principle simplifies the design of individual layers.

Events are the only means of communication between layers of a protocol stack. When an application transmits a point-to-point or a group message, a 'Send' or 'Cast' event is created respectively. The application message is included as a member of the event data structure and is processed by all layers of the stack in a fixed sequence. Each layer may add its own header fields to the message. When the bottommost layer finishes processing the event, the contents of the message and the included headers are marshaled and transmitted on the wire. Upon reception of a packet off the wire, a 'Send' or 'Cast' event is created based on information from the received packet. This event is then processed by layers of the receiver stack in reverse order. Each layer removes the header fields added by its peer. After the event is processed by the topmost layer in the stack, it is delivered to the receiver application.

Events are also used for other purposes. Timeouts requested by layers are sent in the form of events. Layers responsible for maintaining group membership may exchange information about unreachable members using specific event types. Special event types are used to initialize and tear down layers of the stack. The variant data type in OCaml enables Ensemble to use the event data structure for multiple purposes without wasting memory for unused fields. Other than the application message and header fields attached to 'Send' or 'Cast' events, none of the event fields is transmitted on the wire.

An Ensemble layer has the following three elements.

1. A data structure to hold local state information.
2. A format for headers to be added to messages.
3. A set of event handlers to process events.

Each layer is free to define the structure of its local state and the format of header fields to fit its needs. The local state is only accessible by event handlers of the same layer and the header fields are only accessible by event handlers of peer layers. Event handlers implement the functionality of the protocol layer. The processing of an event by a handler eventually leads to the event being passed up or down the protocol stack. Event handlers are divided into 'up' and 'down' handlers. 'up' handlers process events that move from the bottommost to the topmost layer of the stack. 'up' events are usually created following the reception of a packet off the wire. 'down' handlers process events going in the other direction. 'down' events are typically due to the application sending messages to other stacks. Each layer processes one event at a time. The Ensemble framework implements FIFO event queues to buffer events destined for a layer busy handling a prior event.

When a protocol stack is composed from layers, event handlers are so wired that events are passed between adjacent layers appropriately. Each layer defines a function that is given a set of handlers for passing events to adjacent layers and returns a set of handlers that other layers can use to pass events to this layer. Repeated invocation of this composition function results in a complete protocol stack. Conceptually, a layer can be thought of as an automaton that takes an event off one of two possible input queues, processes it and deposits one or more events in two output queues. The two input and output queues correspond to events moving bottom-up and top-down in the stack.

4.3 State machine executor over Ensemble

With Ensemble chosen as the basis for the Composite Protocol framework, components are implemented inside Ensemble layers. The function of an Ensemble layer is carried out in its event handlers. The event handlers have no notion of a state machine. However, protocol components are specified as finite state machines. Their function is implemented in the action and local memory update functions associated with state transitions. To implement a component as per its state machine specification in the Ensemble framework requires additional glue code to map Ensemble events to Composite Protocol framework events and to drive the state machines of components. The design of a state machine executor over Ensemble is explained in this section.

Just as in specification, state machines are represented as a list of states and a list of outgoing transitions for each state. A state transition is defined as an OCaml record parameterized with type variables. The transition record contains the following elements.

1. Enumerated names for current state and next state of the transition
2. Enumerated name for the event type that triggers the transition
3. Guard function
4. Action function
5. Local memory update function

The type variables are used to parameterize the types of local memory, packet memory and the name of the state. This generic OCaml record serves as a template for defining state transitions for all components, irrespective of the actual types of their local memory, packet memory and namespace for states. There are three members of the state transition record that are higher-order values (functions), namely the guard, the action and local memory update functions. Each protocol component also has an OCaml record to save the current control state of the state machines and current values for local memory objects.

The execution of a state machine is independent of its structure and the functionality embedded in state transitions. The common functions to drive state transitions are implemented as a separate module. When an Ensemble event is passed onto a layer, its appropriate event handler is invoked. Not all Ensemble events are of interest to protocol components. If the event is state-machine related, the state-machine executor function, 'event_handler' is invoked. This recursive function implements the state-machine execution algorithm as described in section 3.1.1.3. All synchronous state transitions are carried out until the next state is non-synchronous. The next state and the value of local memory objects are saved and control is returned to the Ensemble framework.

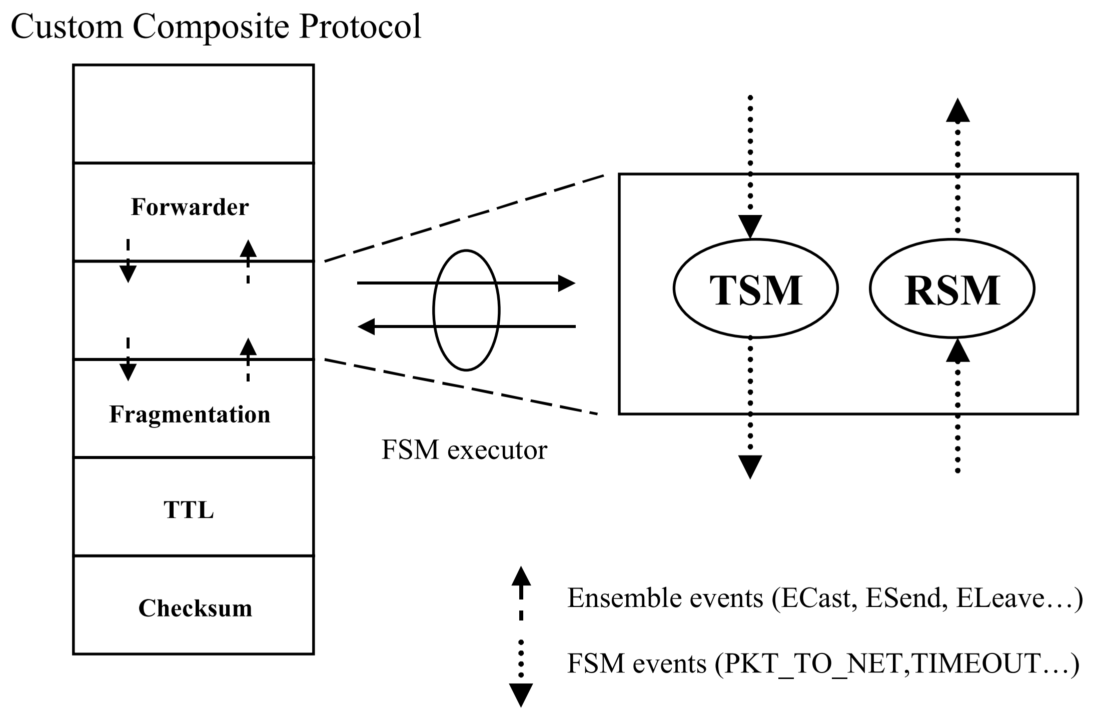
Custom Composite Protocol
Forwarder
Fragmentation
TTL
Checksum
FSM executor
TSM
RSM
Ensemble events (ECast, ESend, ELeave…)
FSM events (PKT_TO_NET,TIMEOUT…)

Figure 4: State machine executor over Ensemble

A schematic of the state machine executor's placement in a composite protocol is shown in Figure 4. The figure shows how Ensemble events are mapped by the executor function onto FSM events. The figure illustrates the case of a component specified as a pair of state machines. Though only one component is shown being driven by the FSM executor, all components in a composite protocol have identical arrangements.

The composite protocol framework supports a rich set of primitives for common protocol operations. These primitives are mapped onto corresponding Ensemble event handlers. For example, the framework function 'PktSend' is a thin wrapper over Ensemble's outgoing down event handler. The set of primitives is defined as an OCaml record and is passed as an argument to action functions. Many of Ensemble's event types are specific to the domain of group communication and hence have been removed while adapting Ensemble to be the basis for the composite protocol framework. PKT_TO_NET and PKT_FROM_NET, the two most common event types are mapped to the Ensemble event type 'ESend'. The TIMEOUT event type is mapped to 'ETimer'.

The specification of components assumes that the application payload and packet memories attached by other components are encapsulated as the payload is processed by each component in the protocol. Ensemble encapsulates only the headers added by layers and the application payload is stored as a separate member of the event data structure. These two entities are grouped into an OCaml record named 'pktpayld'. An instance of this record is made available to the guard, action and local memory update functions.

5. Design of Protocol Components

Design of single-function protocol components requires that a component make as few assumptions about its environment as possible and not assume the presence of other protocol functions in the same composite protocol. In some cases, the design of a component is simplified, if another function is available in the same protocol. Such dependencies shall be made explicit. This chapter discusses some of the issues that have influenced the design of all components. The design of two components is dealt with in detail and some issues that arise when building a protocol using these components are explored.

5.1 Common design issues

Functionality offered by a component shall be as simple as possible yet be useful for a variety of scenarios. For example, the transmit state machine of the fragmentation component handles the fragmentation of one payload at a time. After all fragments of this payload have been transmitted, the next payload is taken up for fragmentation. Though this may increase the delay in fragmenting payloads, the algorithm to buffer and transmit fragments is simplified.

The design of many components assumes that a composite protocol using the component will handle a single flow. A flow is a logical grouping of packets classified based on different criteria. The most common notion of a flow is that of traffic exchanged between two specific application instances. The assumption of handling a single flow and hence packets from a single source simplifies the design of many components. For example, the reliable delivery and fragmentation components are designed based on this assumption and the design of the RSMs of these components is simplified. However, the same assumption does not benefit components that operate on a connectionless packet-by-packet basis. For example, the TTL and forwarding components can handle multiple flows with no significant increase in complexity. Components that handle packets from a single flow are by no means less useful. Another instance of the same component can be used in another composite protocol to handle a different flow. The flow-based component design lets the application choose the exact protocol functions and mechanisms that best fits its needs.

5.2 Fragmentation and Re-assembly

The fragmentation component insures that none of the transmitted packets exceeds a maximum length. If a payload exceeds the specified maximum length, it is divided into fragments and the fragments are individually transmitted. The length of none of the fragments exceeds the maximum prescribed length. The received fragments are buffered until all fragments of a payload are available. Then the payload is reassembled and delivered to the next higher-level component on the receiver's side.

For maximum applicability, this component does not assume the presence of reliable and in-order delivery components in the same composite protocol. Therefore, delivery of reassembled payloads is not guaranteed. When the first fragment of a new payload is received, a timer is started. If all fragments of this payload are received before the expiration of the timer, the reassembled payload is delivered. Otherwise, all received fragments of the payload are dropped when the timer expires.

Each payload is tagged with a unique identifier and each fragment of a payload is consecutively numbered starting from one. The packet memory attached by this component to a fragment includes the payload identifier, fragment number, a flag that indicates if this is the last fragment and fragment data. The most peculiar field of packet memory is that of fragment data. Conventional protocols keep the fragment data as part of their payload and their header fields include only bookkeeping information. This component includes the fragment data as part of its packet memory. When a payload is fragmented, the individual fragments do not make any sense to the higher-level component of the fragmentation component. The fragment data is in fact recognized only by the peer fragmentation component as being a fragment. Since all peer-to-peer communication is accomplished through packet memory and fragment data is not really a payload in its strictest sense, this component includes the fragment data as part of its packet memory. Consequently, all fragments are transmitted using the peer-to-peer communication primitive 'NewPktSend'.

This component is specified as a pair of state machines, because fragmentation is a data plane component and the transmit and receive functions are independent. The TSM is limited to fragment one payload at a time. All fragments of a payload are transmitted (in effect, passed onto the next lower-level component) before the next payload is processed. The RSM does not impose any restrictions on the order of received fragments and payloads. The component only handles packets belonging to a single flow. It also assumes that payload identifiers do not wrap around fast enough to confuse the RSM with fragments with payload identifiers of identical value but of different incarnations.

The component assumes that packet memory fields are received without transmission errors and all packets belong to a single flow. These two are the *required* properties. It *provides* the property that none of the transmitted fragments exceeds the prescribed maximum length. The maximum length of fragments is not inferred from the Maximum Transmission Unit (MTU) of the underlying physical network but is accepted as an initialization-time parameter. It is the responsibility of the stack composer to pass an appropriate maximum length to this component based on the MTU of the physical network used and the number of bytes of packet memory added by components placed lower than fragmentation in the same protocol.

5.3 Packet Forwarding

The forwarding component is responsible for forwarding packets towards their ultimate destination. If a packet is destined for the host on which the composite protocol is running, then it is delivered to the higher-level component. If not, the destination address of the packet is looked up in the routing table. The address of the next hop to reach this destination is found from the routing table. The packet is then forwarded to the next hop. This component shall be part of composite protocols operating at endpoints as well as routers. At endpoints, the component shall be part of a protocol that handles a single flow. At routers, the component can be part of a protocol that handles multiple flows.

The responsibility of this component is limited to determining if this protocol instance is the destination for a packet and otherwise looking up the destination address in the routing table to find the network address of the next hop. The task of sending the packet using appropriate physical addressing mechanisms is the responsibility of another component.

This component is remarkable for the different kinds of memory that it makes use of. The network address of the next hop is assigned to an SLPM field NEXT_HOP_ADDR. An 'AddressTranslator' component is responsible for mapping the network address set in this SLPM field to the physical address appropriate for the network that connects this host to the next hop. The routing table, stored as global memory, is accessed through a functional interface to find next hop addresses. The network address of the source and destination of each packet is carried as packet memory.

At first glance, it may seem that addressing is a separate component. It is expected that a transition from IP version 4 to version 6 shall change only the format of addresses and other components in the same protocol shall remain unchanged. However, the source and destination addresses carried by packets are in fact information for the forwarding component to route a packet to its destination. The design of the forwarding component depends on the addressing scheme used. If a different addressing scheme is desired, then the protocol shall include a variant of the forwarding component designed to work with the new addressing scheme.

The source and destination addresses carried as packet memory are passed to the forwarding component as parameters and are fixed for the lifetime of the component. This signifies the fact that the composite protocol including this component caters to a single flow. However, these parameters only affect packets originated by or destined for this host and not the forwarding behavior of the component. A packet is forwarded solely based on information carried in its packet memory and the contents of the routing table.

The need to match component sequences, as discussed in section 3.4.2, requires that the order of components in protocols operating at endpoints and routers be compatible. In particular, the order of components below and including the forwarder shall be identical at endpoints and routers.

5.4 Sample composite protocols with IP components

When protocol functions are implemented as components and not as a monolith, interesting variants of composite protocols can be configured. This section discusses the different possibilities in using the four IP components: fragmentation, forwarding, TTL and checksum to configure protocols.

The first restriction in the relative order of these four functions is that the TTL component shall always be placed below the forwarding component. This is because TTL has to be a hop-by-hop function to be useful in mitigating the effects of a forwarding loop. Any component placed below the forwarding component will process packets at every hop, whereas components above the forwarding component will only process packets at endpoints. Another restriction is that forwarding and TTL shall always be used in pairs because the TTL component uses an SLPM field named SOURCE_ADDR, which is set to the source address of a packet by the forwarding component. This SLPM field allows the TTL component to distinguish packets originating from this instance of the composite protocol from those being forwarded.

If the endpoints are in the same local network with a very small bit-error rate and if the application can tolerate infrequent bit errors, one can leave the checksum component out of the configured protocol. On the other hand, if the application is sensitive to dropped packets and if bit errors are the main reason for dropped packets, then the checksum component can be replaced with an error-correction component. The checksum component may also be placed above the forwarding component if end-to-end error detection is sufficient.

The placement of fragmentation component relative to the forwarding component has interesting consequences. If the path taken by packets includes multiple heterogeneous physical links with different MTUs, placing fragmentation below forwarding insures that the length of transmitted packets will always be less than or equal to the MTU of the chosen physical link. The fragmentation component at each intermediate hop can be configured with a different value of maximum length to cater to physical links with different MTUs.

Fragmentation can be placed above the forwarding component in the following two cases. If the network has homogenous physical links with same values for the MTU, a packet once fragmented may never need to be fragmented again. If the participating endpoints have undertaken 'Path MTU discovery' to find the least MTU of all physical links and if this least MTU is passed as a parameter to the fragmentation component at the endpoints, then again a packet may not be fragmented a second time.

Given the arguments above, the following component sequences may be used by applications based on their needs. The component sequences are listed top to bottom. Other components may also be part of these composite protocols.

1. FORWARD: TTL
2. FRAGMENT: FORWARD: TTL
3. FORWARD: FRAGMENT: TTL
4. FORWARD: FRAGMENT: TTL: CHECKSUM
5. FORWARD: TTL: ERROR CORRECTION
6. CHECKSUM: FORWARD: FRAGMENT: TTL

The choices shown above exemplify the flexibility offered by composite protocols.

6. Implementation of Control Interface

The CONTROL event serves as a medium of communication between components. It carries an instance of a service request added by the controlling component. The definition of the format of a service request is part of the service specification. In our implementation of the control interface in OCaml, service requests are represented as variant types. The names of commands form the data constructors for the variant type and additional parameters for a command are declared as arguments to the data constructors. The representation of a service request as an OCaml variant type captures the notion that any instance of a service request holds one valid command at a time and associated parameters for the chosen command.

The controlled component is required to define transitions to handle every possible command of the offered service in every non-synchronous state. The specification of a component includes a predicate that holds TRUE only for the names of services offered by the component. This predicate is evaluated for every control event fetched from the FIFO event queues. Only events that satisfy the predicate are delivered to the state machine. Other events are transparently transferred to the next component in the processing sequence. This shortcut implements the optimization that allows components to define transitions for only those services provided by them.

6.1 Realization of file transfer using the control interface

As briefly discussed in section 3.1.5.3, the functionality of file transfer is implemented as a pair of components, named FTP control and FTP data. The FTP control component manages the transfer of files by checking access permissions, verifying the existence of requested files and authenticating users. The FTP data component performs the actual file transfer by reading from files in the local file system and sending the data on the wire and vice versa. This section explains the details of the application of the control interface to enable communication between the two file transfer components.

The FTP control component defines a service called 'FTPControlCM'. This service has five possible commands: USER, LIST, GET, PUT and QUIT. The USER command is meant to start a file transfer session by authenticating the client to the server. The LIST command is used to retrieve a listing of files in a particular directory on the server's file system. The GET command is used to retrieve a file from the server, the PUT command to save a file on the server and the QUIT command to terminate the file transfer session. Most of the five commands have additional parameters. The USER command takes the username and password for authentication. The LIST command requires the name of the remote directory, the GET and PUT commands take the names of the local (on the client) and remote (on the server) files.

At the client, the file transfer application is a command-line utility similar to the Unix 'ftp' tool. The client application uses a composite protocol with one instance each of the FTP control and FTP data components along with other components for reliable and in-order delivery of packets. The peer endpoint is the file server, which also has the identical composite protocol, but the FTP control component has been configured to act as a server. The client application serves as a simple command-line parser. It constructs a service request of type 'FTPControlCM' for every correct command entered by a human user. One of the five possible commands of the service 'FTPControlCM' is chosen based on user input. The application creates an instance of the control event, attaches the service request and passes the control event down the protocol. The event bypasses every component between the application and the FTP control component and is delivered to the FTP control component.

The client FTP control component talks to its peer at the server to authenticate the user; to verify the existence of requested files on the server's file system and to check access permissions. This is accomplished by sending peer-to-peer messages using the primitive 'NewPktSend'. If the current command from the application is LIST, GET or PUT and if sanity checks succeed, the FTP control components at the client and server delegate the file transfer task to their companion FTP data components.

For this purpose, the FTP data component defines a service named 'FTPDataCM'. It includes three commands: SEND_FILE, RECV_FILE and STOP. The commands SEND_FILE and RECV_FILE take the name of a file in the local file system as a parameter. The command SEND_FILE instructs the component to read the specified file from the local file system, packetize it and send it on the wire. The command RECV_FILE is the complement of SEND_FILE. The command STOP terminates the ongoing file transfer task and completes bookkeeping operations. The FTP control components at the client and server each create a service request of type 'FTPDataCM'. Depending on the direction of data transfer, one of the peer FTP data components is sent the command SEND_FILE while the other is given the command RECV_FILE. This starts the actual transfer of files between the client and the server.

6.1.1 Message exchanges for the GET command

Figure 5 illustrates the exchange of messages between the composite protocol at the client and server and control events between the application, FTP control and FTP data components when the user at the client initiates a GET command. The numbers in bold on the arrows indicate the temporal order of events. The horizontal arrows indicate transmission of packets between the client and the server and the vertical arrows signify control events within a composite protocol.

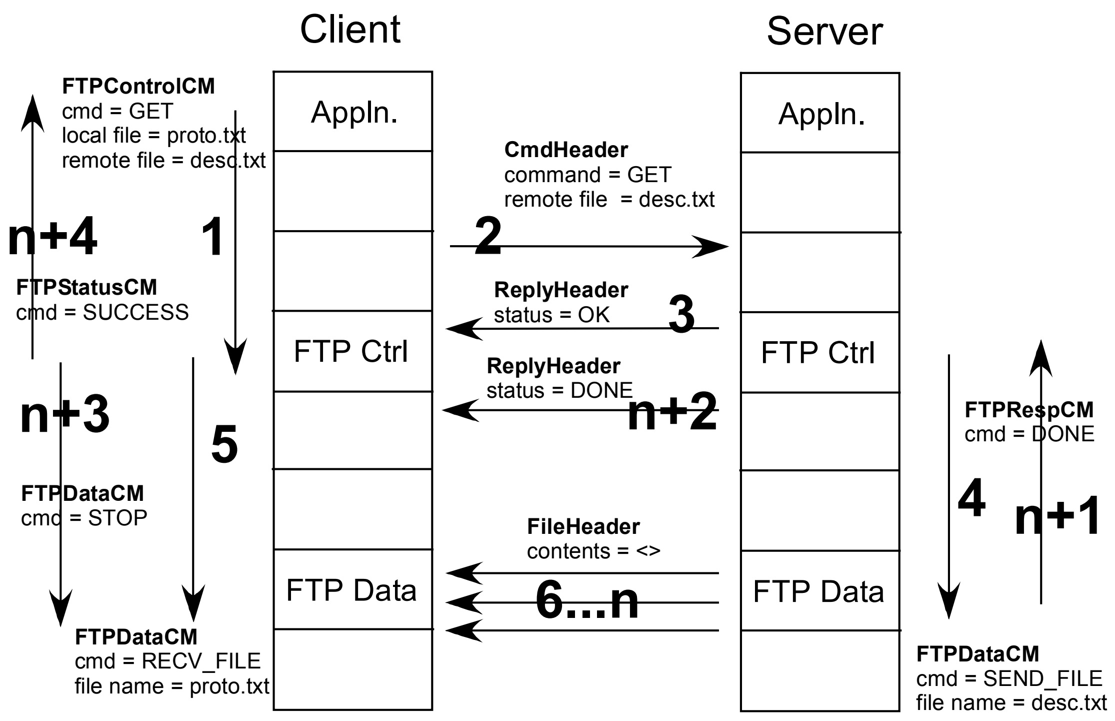

Figure 5: Component Interactions for the GET command

When the user enters the GET command at its prompt, the client application creates an instance of the service request ’FTPControlCM’. The GET command of this service is chosen and the parameters of this command; local file name and remote file name are set. This service request instance is attached to a newly created control event and the event is passed down the protocol (Message 1). This control event is delivered to the FTP control component by the framework. The FTP control component talks to its peer at the server to check if the remote file exists on the server machine and if the user is permitted to read the file (Message 2). On receiving confirmation from its peer component (Message 3), the FTP control component at the client creates an instance of the service request ’FTPDataCM’. It chooses the

RECV_FILE command and sets the name of the local file as its parameter. The control component creates a new control event, attaches the service request ’FTPDataCM’ and sends the event down the protocol stack (Message 5). Meanwhile, the FTP control component at the server also makes a ’FTPDataCM’ service request, but chooses the SEND_FILE command and sets the name of the remote file as its parameter (Message 4).

The two FTP data components spring into action at this time. The data component at the server reads the remote file and sends multiple packets of file contents on the wire. Its peer at the client receives these packets and writes their payload into the local file (Messages 6 through n). When the data component at the server is done sending the contents of the remote file, it signals its companion FTP control component (Message n+1). This signal is again realized as a service request from the data component to the control component. The FTP control component at the server then sends an end-of-file message to its peer (Message n+2).

On receiving the end-of-file message, the FTP control component at the client creates a service request instance of ’FTPDataCM’ and chooses the STOP command. This service request is passed to the data component via another control event (Message n+3). The FTP data component at the client then closes the local file. The FTP control component also signals the successful completion of the GET command to the application (Message n+4).

7. Performance Evaluation

Benchmarking tests were conducted to evaluate the performance of an Ensemble application that utilizes composite protocols for communication. The performance of the in-kernel implementation of the TCP/IP protocol suite in Linux was chosen as the benchmark. Test applications were written on Ensemble to mimic the functionality of two Linux network testing tools, *ping* and *ttcp*. This chapter describes the test setup, tabulates the results of tests performed and discusses the major observations.

7.1 Test setup

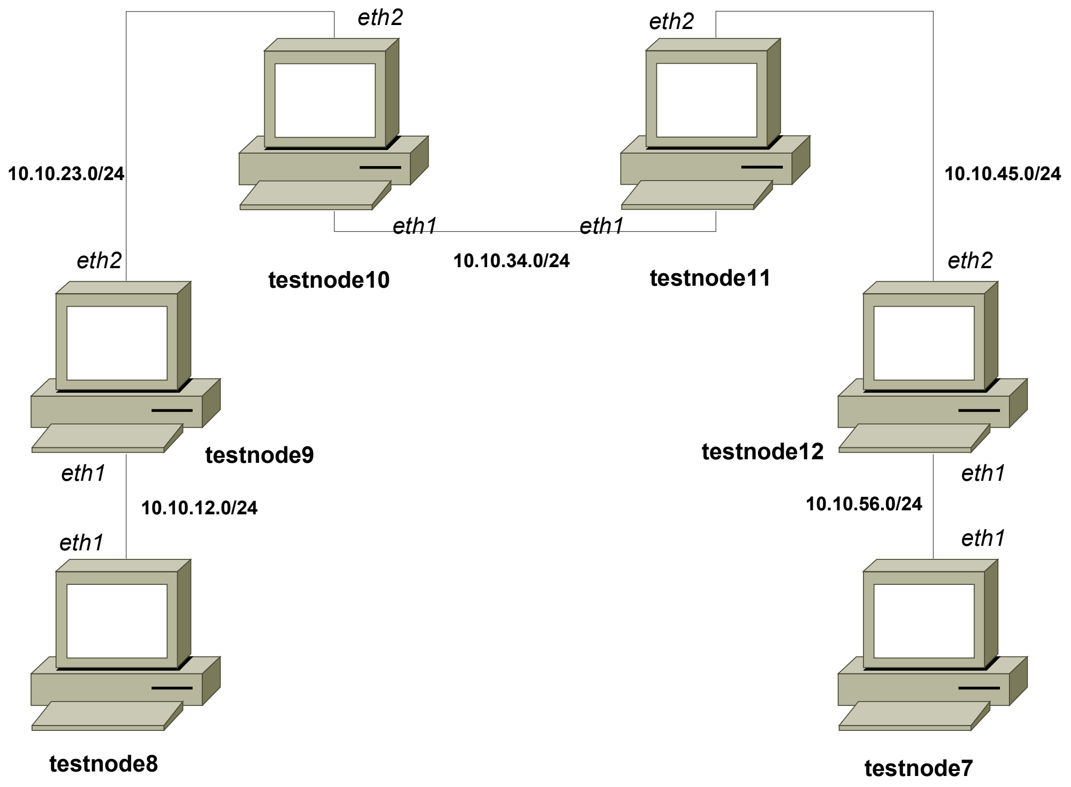

Figure 6: Setup for running performance tests

Figure 6 illustrates the topology of nodes used to run performance tests. Six PCs from the *IANS bluebox*, a compact network testbed of 13 nodes, were used for executing the tests. All machines used had identical hardware and software configurations. Each PC is equipped with a 533MHz Pentium III processor, 128 MB of RAM, 20GB of hard disk space, a built-in Fast Ethernet (100 Mbps) NIC and one to four additional Fast Ethernet NICs. The machines were installed with RedHat v7.1 distribution of Linux, including the kernel version 2.4.3-12. Ensemble test applications were compiled using OCaml v3.06 compiler suite to native code (Linux ELF format). The NICs were directly connected using crossover Ethernet cables. The primary objectives of the tests were to measure the round-trip latency and one-way throughput of a composite protocol with respect to several metrics, namely message size, number of intermediate hops, protocol components used in the composite protocol etc.

Ensemble, being a user-level program, uses the BSD socket interface to send messages to its peers. The current adapted implementation of Ensemble supports communication over UDP sockets as well as Packet sockets (a Linux-specific interface that allows packet transmission directly over data-link layer frames). Unless otherwise specified, the performance tests used UDP sockets as the means of communication between Ensemble applications. It may seem counter-intuitive to run a composite protocol on top of the TCP/IP protocol suite (as is the case when UDP sockets are used), but this arrangement simplifies protocol development. The ubiquity of the socket interface and the fact that they can be used by non-privileged users of a Linux workstation eases the requirements for a protocol development environment. However, tests were also conducted using Packet sockets to quantify the cost of running composite protocols over UDP sockets.

All tests were performed using a composite protocol with UDP-like functionality, unless explicitly stated otherwise. The default composite protocol was composed of the following four components in that order top-down: FORWARD, TTL, FRAGMENT and CHECKSUM. The design of the FORWARD and FRAGMENT components is described in sections 5.2 and 5.3 respectively. The 'max_frag_len' parameter of the FRAGMENT component was set to 1400 bytes to avoid IP fragmentation, when composite protocol messages are transmitted using UDP sockets over an Ethernet network. The TTL component adds a time-to-live field to its packet memory at the sender endpoint. When a packet is received at an intermediate router or at its destination, the time-to-live field is stripped from packet memory, decremented and copied to SLPM. If the packet is turned around by the FORWARD component (as would happen in an intermediate router), the new value of TTL is copied from SLPM and sent as the new instance of packet memory. The CHECKSUM component computes a checksum over its entire payload, which also includes the packet memory fields of components above itself in the composite protocol, using the Internet Checksum algorithm [23].

For some of the tests, a TCP-like composite protocol was used. This protocol was composed of the following components in that order top-down: RELIABLE, FORWARD, TTL, FRAGMENT and CHECKSUM. The RELIABLE component implements reliable and in-order delivery of packets and employs positive acknowledgements and a go-back-n retransmission mechanism.

The tests were run with modified OCaml garbage collection parameters, the implications of which are discussed in section 7.5

7.2 Results of round-trip latency tests

The round-trip latency of sending a message using a composite protocol with UDP-like functionality was measured as a function of message size and number of intermediate hops. The tests involved 11 trials of sending 1000 messages each trial to a receiver that simply echoed the messages back to the sender. The average round-trip latency and variance of latency were collected from each trial. The sender application employed the stop-and-wait technique and only one message was in flight at any time during the test. For measuring latency as a function of message size, the sender and receiver applications were directly connected (one hop). For measuring the variation of latency with number of hops, minimum-size messages were used. The minimum message size for Ensemble test applications is 1 byte. Equivalent tests were also conducted using the Linux utility *ping,* whose minimum message size is 8 bytes (*ping* sends ICMP messages that carry a timestamp).

Message Size (in bytes)	Average round-trip latency (in microseconds)		Normalized standard deviation	
	Composite Protocols	Linux Networking	Composite Protocols	Linux Networking
1(8)	340.582	75.818	0.3360	0.2969
200	412.298	111.182	0.1531	0.2040
400	477.884	146.818	0.2418	0.4697
800	640.312	217.455	0.2241	0.3356
1300	796.937	304.727	0.1635	0.3930
1600	1228.090	377.727	0.1507	0.3105

Table 1: Variation of round-trip latency with message size

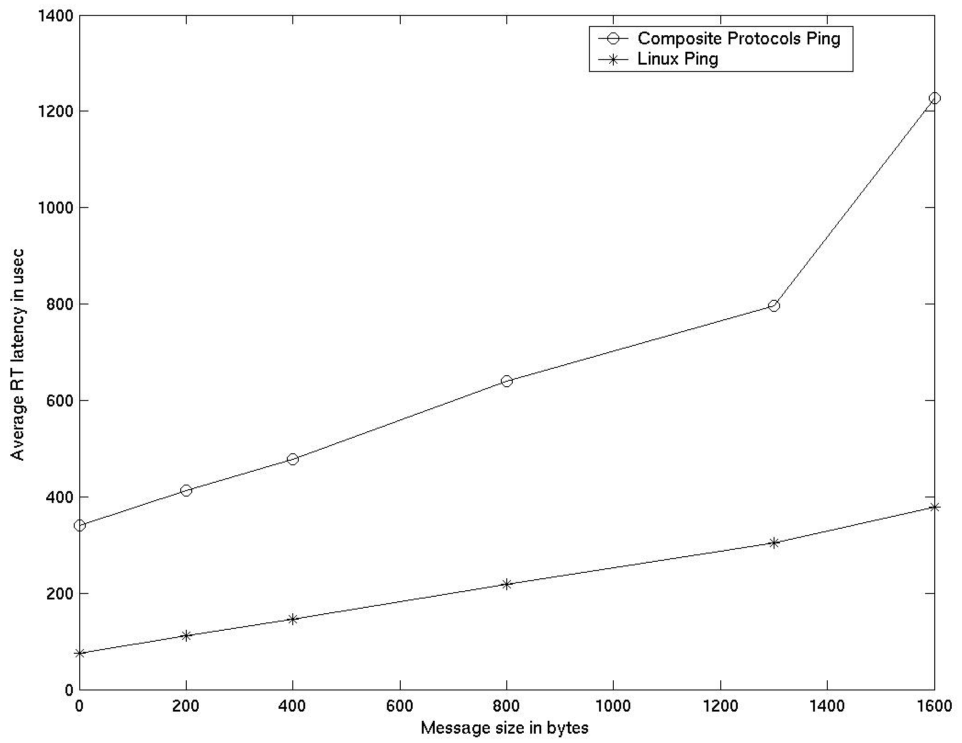

Figure 7: Variation of round-trip latency with message size

The variation of round-trip latency with respect to message size is tabulated in Table 1 and plotted in Figure 7. The average round-trip latency is calculated by taking the average of the average latencies reported by individual trials. The normalized standard deviation is the ratio of pooled standard deviation to average round-trip latency. The pooled standard deviation is determined by finding the square root of the average of variances reported by individual trials.

Latency increases with increase in message size, as expected. The performance of composite protocol ping is worse than that of Linux ping, by a factor of 2 to 4. Given the constraints imposed by the specification methodology and limitations of the current implementation, this is a reasonable performance penalty to pay. Executing a component's state machine incurs a non-trivial amount of overhead, which the in-kernel implementation in Linux does not. There are no well-defined boundaries between layers in the Linux implementation with respect to memory access and all layers operate on a common instance of a socket buffer. Linux protocol software can afford to perform pointer arithmetic on socket buffers and minimize memory copies. The strict layering enforced by the composite protocol framework makes it impossible to access the local memory of another component. Moreover, Ensemble is a user-level program and hence incurs further overhead in sending and receiving messages compared to the Linux in-kernel implementation. Finally, the Linux implementation has matured over many years of use and improvement, whereas only limited time could be spent so far in optimizing the current implementation of composite protocols.

The sharp increase in the latency of composite protocol ping, when message size is increased from 1300 to 1600 bytes is attributed to the overhead of fragmentation at the FRAGMENT component. When message size is less than the 'max_frag_len' parameter, the FRAGMENT component does not copy the payload that it processes. When a payload has to be fragmented, new memory is allocated to hold individual fragments. It is not possible to avoid this memory copy in our current implementation, because each fragment is a packet of its own and the FRAGMENT component must give up access to fragment data when the fragment leaves its state machine. Moreover, the lifetime of individual fragments in the same instance of the composite protocol cannot be easily predicted, because it is possible to include other components below FRAGMENT that may buffer their payloads upon transmission.

Number of hops	Average round-trip latency (in microseconds)		Normalized standard deviation	
	Composite Protocols	Linux Networking	Composite Protocols	Linux Networking
1	340.582	75.818	0.3360	0.2969
2	666.503	139.091	0.3952	0.3267
3	986.262	199.455	0.1861	0.2385
4	1307.730	263.909	0.2063	0.5072
5	1635.820	321.000	0.2409	0.2142

Table 2: Variation of round-trip latency with number of hops

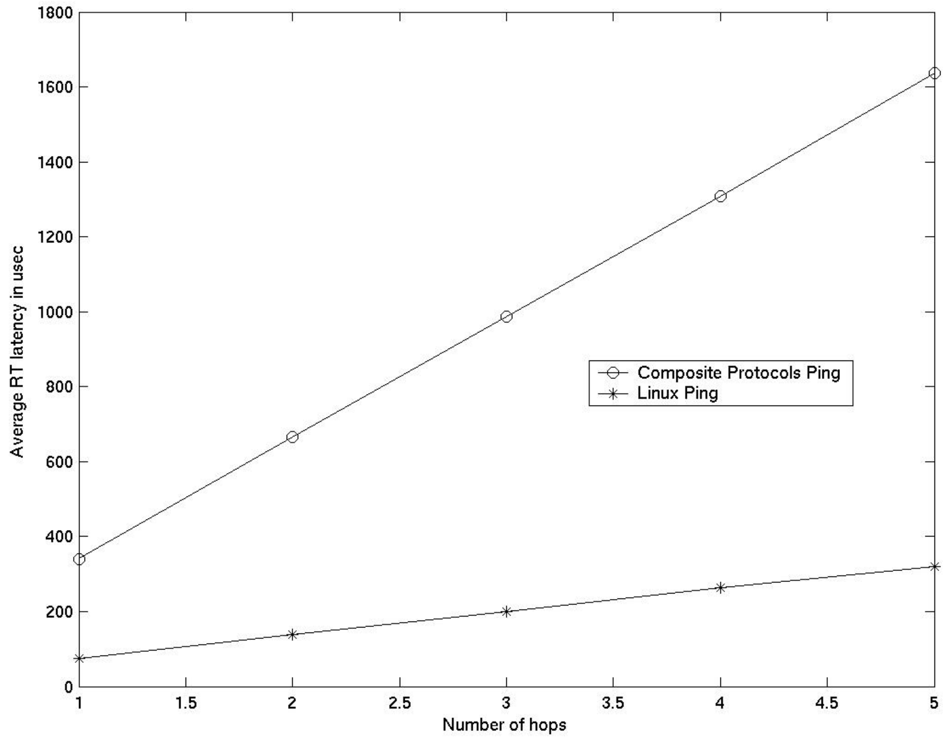

Figure 8: Variation of round-trip latency with number of hops

Table 2 and Figure 8 show the results of the test that measures the variation of latency with respect to number of hops. The rate of increase of latency for composite protocol ping, with respect to number of hops, is higher than that of Linux ping. Each additional hop adds a delay of about 320 microseconds to round-trip latency. This per-hop increment is only slightly smaller than the round-trip latency for a single hop (340 microseconds). This marginal variation is because the number and arrangement of components that process a packet at an intermediate router are the same as that at the source and destination endpoints for a UDP-like composite protocol. On a router, the highest component in the composite protocol that a packet can reach is FORWARD. The same sequence of components (FORWARD, TTL, FRAGMENT and CHECKSUM) also process packets on endpoints. The difference between round-trip latency for a single hop and the per-hop increment in latency may be attributed to the absence of application-level un-marshaling of packet payload at routers.

Transport used	**Average round-trip latency (in microseconds)**	**Normalized standard deviation**
UDP	340.582	0.3360
ETH	322.642	0.2929

Table 3: Round-trip latency for different transport options

The average round-trip latency of composite protocol ping between directly connected machines for a minimum-size message was also measured when UDP sockets and Packet sockets were used as the "transport" mechanism for composite protocol messages. The results of this test are shown in Table 3. The additional overhead of running composite protocols over UDP sockets is marginal compared to running them over Packet sockets. The UDP transport adds only 18 microseconds of delay to the latency measurements. The major portion of the delay is due to the composite protocol and not because of the transport mechanism chosen.

Composite Protocol used	**Average round-trip latency (in microseconds)**	**Normalized standard deviation**
UDP-like	340.582	0.3360
TCP-like	554.605	0.0842

Table 4: Round-trip latency for different composite protocol configurations

We also measured the average round-trip latency of a composite protocol with UDP-like functionality and another with TCP-like functionality. The test was conducted on directly connected machines and minimum-size messages were used. The results of this test are listed in Table 4. The TCP-like composite protocol adds a non-trivial amount of delay to the latency measurements, as much as 215 microseconds. This increase is attributed to the overhead introduced in the RELIABLE component due to buffering of packets before transmission and sending of positive acknowledgments.

7.3 Results of one-way throughput tests

The average throughput of an application using a UDP-like composite protocol was measured with respect to message size and number of hops. This series of tests involved sending messages from an Ensemble test application as fast as it can. It was observed that the sender's rate of transmission was far greater than what the receiver could handle. This resulted in UDP buffer overruns at the receiver. A mechanism was added to the application to decrease the rate of transmission of the sender so that packet loss is minimized at the receiver and a more stable estimate of the application's throughput can be obtained. This mechanism was calibrated to eliminate packet loss at the receiver for a run of 10000 messages over a single hop. The sender was slowed by a factor of 10. The same setting was used for measuring the variation of throughput with number of hops. In each trial, 10000 messages were sent so that the receiver could sustain the throughput for at least 1 second. The throughput as measured at the receiver was collected for each trial and samples were averaged over 11 trials. The normalized standard deviation is the ratio of standard deviation of throughput to average throughput.

Equivalent tests were run using the Linux tool *ttcp* over UDP sockets for comparison. No adjustments were made to the rate of transmission at the sender side of ttcp, because ttcp is designed to transmit as fast as possible and provides no interface to adjust its rate of transmission. It was observed that the receiver side of ttcp lost more packets when the packet size was small. For maximum-size packets (that avoid IP fragmentation over an Ethernet network), there was minimal packet loss at the receiving ttcp process. When run over UDP sockets, ttcp sends four-byte sentinel messages to indicate the start and end of transmission. Therefore, the minimum possible message size with ttcp over UDP is 5 bytes.

Message Size (in bytes)	Average receiver throughput (in Mbps)		Normalized standard deviation	
	Composite Protocols	Linux Networking	Composite Protocols	Linux Networking
1(5)	0.06281	2.054	0.0043	0.1611
100	6.112	31.602	0.0035	0.0779
200	11.914	58.976	0.0039	0.0087
400	22.639	79.089	0.0033	0.0013
800	41.171	88.048	0.0034	0
1300	60.022	90.730	0.0030	0
1600	27.654	85.939	0.0838	0.0268

Table 5: Variation of average throughput with message size

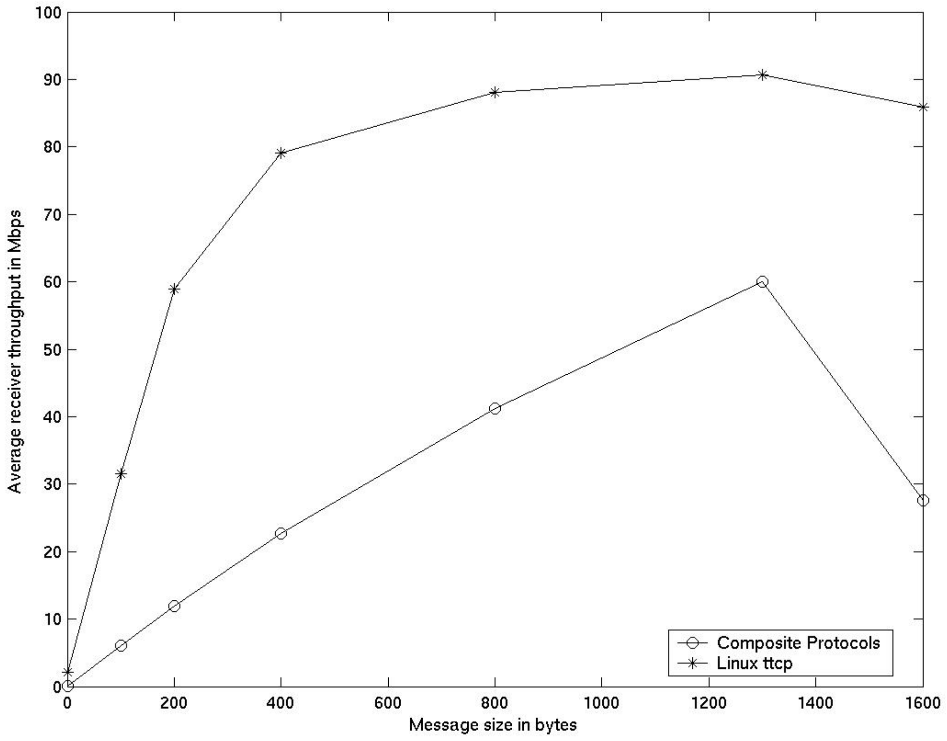

Figure 9: Variation of average throughput with message size

Table 5 and Figure 9 illustrate the variation of average throughput with message size. As expected, throughput increases with increase in message size, because the overhead of sending a packet is spread over a greater number of bytes in bigger messages. Maximum throughput of about 60 Mbps is achieved at a message size of 1300 bytes for the UDP-like composite protocol. The components of this composite protocol add about 90 bytes of packet memory and marshaling information. The Ensemble framework adds another 28 bytes of headers for holding the size of the transmitted packet and for identifying the appropriate instance of the composite protocol at the receiver. The maximum theoretical throughput of 88.04 Mbps can be achieved at a message size of 1354 bytes, given this information. The current implementation of composite protocols achieves about 68% of the theoretical maximum. This is due to inefficiencies in the implementation and the resultant need to decrease the rate of transmission of the sender.

The length of packet memory added by components remains constant at 90 bytes, irrespective of the message size. For small messages, this overhead is significant. This explains the low throughput values achieved by the composite protocol at smaller message sizes. The marked decrease in throughput for a message size of 1600 bytes is again due to the overhead of fragmentation at the FRAGMENT component, as explained in section 7.2. In comparison, the throughput of Linux ttcp running over UDP sockets approaches the maximum theoretical throughput of 95.7 Mbps for a message size of 1300 bytes. Moreover, ttcp's throughput is only marginally reduced when 1600-byte messages are subjected to IP fragmentation.

Number of hops	**Average receiver throughput (in Mbps)**		**Normalized standard deviation**	
	Composite Protocols	**Linux Networking**	**Composite Protocols**	**Linux Networking**
1	60.022	90.730	0.0030	0
2	39.570	90.626	0.0039	0.0015
3	39.509	90.399	0.0071	0.0040
4	39.559	89.877	0.0034	0.0083
5	38.977	90.176	0.0068	0.0036

Table 6: Variation of average throughput with number of hops

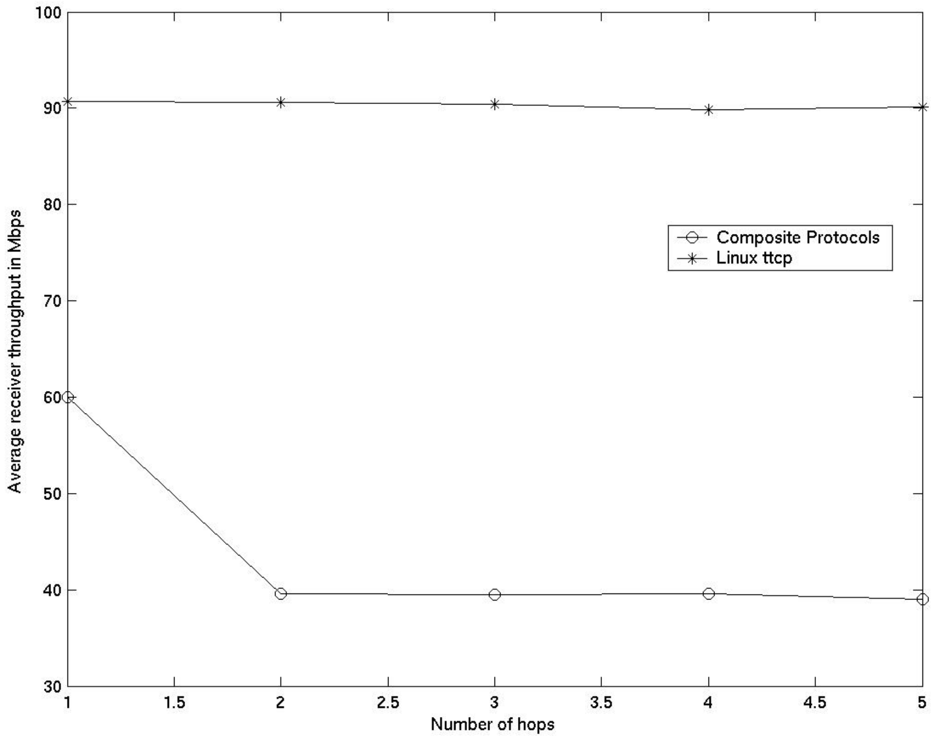

Figure 10: Variation of average throughput with number of hops

The results of the test to measure the effect of adding intermediate routers on average throughput are shown in Table 6 and Figure 10. The throughput of the composite protocol test application decreases by about 33% when one intermediate router is introduced. It remains almost constant when more routers are added along the path. To investigate this behavior, the number of packets forwarded by each router was counted. For a trial of 10000 messages, the first and subsequent routers along the path forwarded about 6800 packets. There was a 32% packet loss at the first router and no further loss at other routers. This statistic shows that a router is able to sustain a throughput of 39 Mbps without losing packets, irrespective of its placement along the path of packets. In contrast, throughput of Linux ttcp decreases only marginally when one or more intermediate routers are introduced.

Transport used	Average receiver throughput in Mbps	Normalized standard deviation
UDP	60.022	0.0030
ETH	62.558	0.0018

Table 7: Average throughput for different transport options

The average throughput of a UDP-like composite protocol was also measured when run over UDP sockets and Packet sockets. This test was run between two directly connected machines and used a message size of 1300 bytes. The results of this test are shown in Table 7. Skipping UDP and IP processing of the Linux TCP/IP implementation leads only to an increase of 2.5 Mbps in average throughput. This reinforces our earlier assertion that the overheads affecting performance are primarily due to the implementation of composite protocols and not due to the transport mechanism used.

Composite Protocol used	Average receiver throughput (in Mbps)		Normalized standard deviation	
	Composite Protocols	Linux Networking	Composite Protocols	Linux Networking
UDP-like	60.022	90.730	0.0030	0
TCP-like	20.202	89.626	0.0030	0.0002

Table 8: Average Throughput for different composite protocol configurations

We measured the average throughput of the Ensemble test application for a UDP-like composite protocol and a TCP-like composite protocol. The results of this test are shown in Table 8. The TCP-like composite protocol included a RELIABLE component that implements the sliding-window algorithm and employs a positive acknowledgement scheme and a go-back-N retransmission mechanism. There was no packet loss on the network used and the 'max_win_size' parameter of the RELIABLE component was set to 140 packets. The test application was blocked (prevented from sending any more messages) when the number of unacknowledged packets equaled 'max_win_size'. The application was unblocked upon reception of one or more acknowledgements. It was experimentally observed that the sender did not retransmit any of the packets. The average throughput of the test application using a TCP-like composite protocol is three times worse than that using a UDP-like protocol. On the other hand, Linux ttcp running over UDP and TCP sockets achieved almost equal throughput values.

It is remarkable that the normalized standard deviation for latency tests is significantly higher than that for throughput tests. The method used to determine standard deviation for latency tests is different from that used for throughput tests. Each trial in a latency test reports the average and variance of latency. However, each trial in a throughput test only reports a throughput sample value. The standard deviation of latency is high, because its calculation includes the variance of latency from individual trials.

7.4 Results of one-way throughput test of ftp

The set of tests described in the previous section measure throughput when messages are sent from an application that utilizes a composite protocol. It is also possible to send messages from one or more of the components that are part of a composite protocol. The test ftp application on Ensemble makes use of this feature. The client and server ftp test applications were configured to use a composite protocol with the following components: FTP_CONTROL, FTP_DATA, FORWARD, TTL, FRAGMENT and CHECKSUM. The design of the FTP_CONTROL and FTP_DATA components is explained in section 6.1. One of the FTP_DATA components at the server and the client start sending the contents of the file to be transferred, upon getting a GET or PUT request. Though this configuration is not useful in a real network because of the absence of a reliable delivery component, it enables the measurement of possible throughput from a component. Had the RELIABLE delivery component been included in this protocol, any throughput achieved would have been limited by the design of the RELIABLE component and not that of the FTP_DATA component.

The average throughput achieved in sending files between an FTP client and server was measured as a function of file size. It was again observed that the sending FTP_DATA component was overwhelming its peer with its rate of transmission. We incorporated a mechanism to decrease the rate of transmission of the sending FTP_DATA component by adding empty synchronous transitions to its state machine. We also calibrated this mechanism to avoid packet loss at the receiver, for transmission of files up to 20 MB. The sender FTP_DATA component was slowed by a factor of 12.

File size (in MB)	Average receiver throughput (in Mbps)	Normalized standard deviation
5	64.212	0.0034
10	63.811	0.0045
15	60.852	0.0630
20	41.983	0.0841

Table 9: Variation of throughput of FTP_DATA with file size

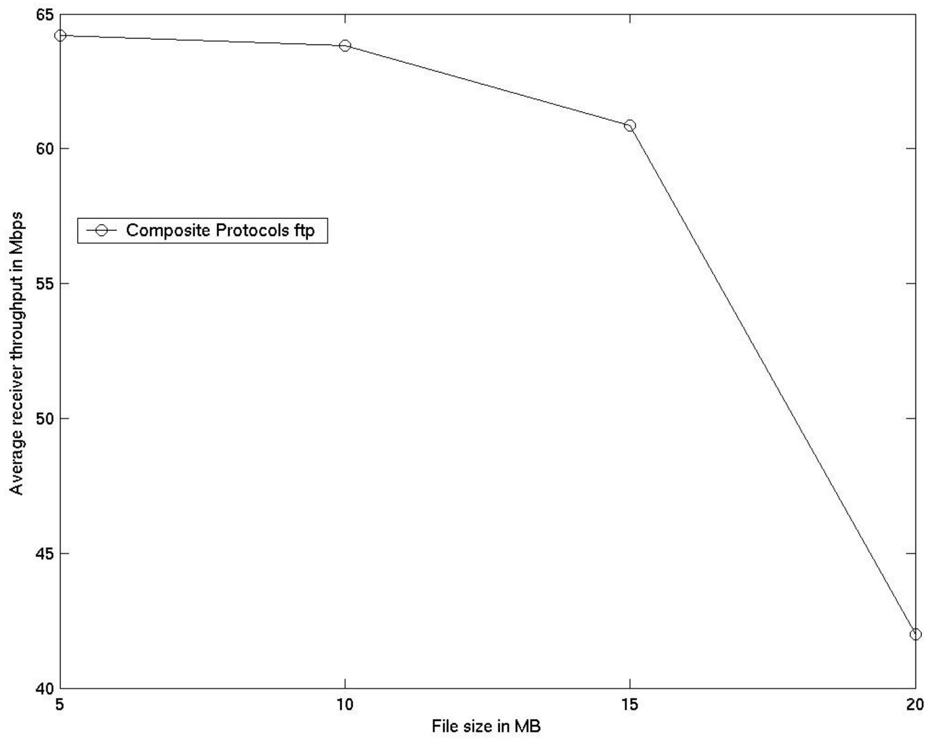

Figure 11: Variation of throughput of FTP_DATA with file size

The results of the test to measure the effect of size of file transferred on throughput are shown in Table 9 and Figure 11. The FTP_DATA component is able to achieve a throughput as high as 64 Mbps for a file size of 5 MB. This throughput is only slightly greater than the maximum throughput of 60 Mbps achieved by the test application. The overhead of sending a message from the application is significantly greater than that from a component, only if the component is placed very low in the composite protocol. The FTP_DATA component, which is placed above FORWARD, incurs almost the same overhead as an application using a UDP-like composite protocol. The drop in throughput for a 20-MB file is due to the saturation of the OCaml heap as more memory is allocated to transfer the larger file and the suppression of garbage collection for the duration of the test.

7.5 Effect of OCaml garbage collection on performance

OCaml supports automatic memory management and uses garbage collection (GC) to reclaim memory that is no longer used. The initial performance tests were run with default GC parameters. It was observed that garbage collection was triggered regularly when messages were being sent for the latency test. This led to regular spikes in round-trip latency sample values and resulted in a high standard deviation of latency for a single test run. It was decided to run the tests already described in sections 7.2 to 7.4 with modified GC parameters to minimize the effect of GC on the results obtained. The 'minor_heap_size' and 'major_heap_increment' parameters were increased to 64M words. 'space_overhead', a parameter that signifies the eagerness of garbage collection with higher values indicating less eager GC, was set to its highest allowed value of 100. The default values for these parameters were 32K words, 62K words and 80 respectively. With the modified settings for GC parameters, it was verified that there was no garbage collection during the execution of tests. This section presents the results of tests run with default GC parameters to better understand the effect of OCaml GC on the performance of composite protocols.

GC settings	Average round-trip latency (in microseconds)	Normalized standard deviation
default	824.013	0.8560
modified	796.937	0.1635

Table 10: Effect of GC parameters on round-trip latency

Table 10 shows the results of 11 trials of the round-trip latency test between directly connected machines for a message size of 1300 bytes. The normalized standard deviation of latency is as high as 0.85 with default GC parameters. This is due to the performance penalty incurred by regular garbage collection. With modified parameters that avoid GC, standard deviation is significantly lower than that for a test with default parameters. It is also remarkable that the improvement in average latency with modified GC parameters is only marginal. These two observations prompted the decision to use modified parameters consistently for all performance tests.

GC settings	Average receiver throughput (in Mbps)	Normalized standard deviation
default	61.487	0.0069
modified	60.022	0.0030

Table 11: Effect of GC parameters on average throughput

Tests were also conducted to measure the throughput of an application using a UDP-like composite protocol between directly connected machines for a message size of 1300 bytes, with default and modified GC parameters. The results are tabulated in Table 11. It was observed that regular GC did not affect the throughput achieved by the application. In fact, throughput was marginally better when default GC parameters were used.

8. Summary and Future Work

This thesis describes a new methodology to design protocol components. Component functionality is represented as a finite state machine and memory objects used by a component are classified into categories based on their scope and extent. Improving the prospects of formal analysis and verification of component and protocol functionality has been a major driving force for this methodology. Though the exposure of a uniform interface by a component facilitates arbitrary composition, the need for a peculiar control interface is also explained. The control interface is seamlessly integrated with the component's state machine. The responsibilities of an underlying composite protocol framework are discussed. Different methods of composition of protocols are explored and the linear stacking approach has been chosen based on its simplicity and for being conducive to formal analysis.

The Ensemble group communication system has been chosen as the basis for implementing the composite protocol framework. A state machine executor has been realized over Ensemble. The functional components of IP and UDP have been identified, specified using this methodology and implemented. A file transfer application has also been built out of reusable components using the notion of a control interface. The performance of a UDP-like composite protocol has been measured and evaluated.

8.1 Future Work

There are many possible areas of improvement in composite protocols. This section identifies some of the topics for improvements and extensions to the design of composite protocols.

1. The thesis suggests many restrictions that enhance the ease of performing formal analysis tasks. For example, an SLPM field shall always be so used that there is a single writer and a single reader. Another example is the completeness of guard expressions in state machines. Many of these restrictions are not currently enforced. Tools can be written that analyze component specifications and protocol compositions to enforce these restrictions.
2. Many formal verification tasks on specifications of components are carried out manually. For formal analysis to be widely applicable, using the verification tools shall be as simple as using a language compiler. Further work needs to be done to automate the verification process as much as possible.
3. Currently, component functionality is implemented manually as per specifications. Automatically generating executable code from specification is a possible area of improvement. Alternatively, tools may have to be written to formally prove the equivalence of hand-written code and the specification.
4. Given the association between protocol components and the properties they provide and a library of components, a configuration tool can be built to automatically compose a protocol that provides a set of desired properties. Such a "Properties-In Protocol-Out" tool can simplify the composition process. This

utility can also be coupled with other tools that enforce component and framework restrictions.

5. More protocol functions and network services can be specified and implemented using this methodology, thus demonstrating its wide applicability.
6. The current implementation of the composite protocol framework can be improved to minimize the disparity in packet-processing rates at the sender and receiver that adversely affects the throughput achieved by current applications.

Bibliography

[1] ISO, "Information Processing Systems - OSI Reference Model - The Basic Model", ISO/IEC 7498-1, 1994.

[2] David D. Clark, "Modularity and Efficiency in Protocol Implementation", RFC 817, IETF, July 1982.

[3] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden, "A Survey of Active Network Research", IEEE Communications Magazine, January 1997, pages 80-86.

[4] Jon Postel, "User Datagram Protocol", RFC 768, IETF, August 1980.

[5] Jon Postel, "Transmission Control Protocol", RFC 793, IETF, September 1981.

[6] Norman C. Hutchinson and Larry L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, Vol. 17(1), January 1991, pages 64-76.

[7] Sean W. O'Malley and Larry L. Peterson, "A Dynamic Network Architecture", *ACM Transactions on Computing Systems*, Vol. 10(2), 1992, pages 110-143.

[8] Gary T. Wong, Matti A. Hiltunen and Richard D. Schlichting, "A Configurable and Extensible Transport Protocol", *Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001)*, Anchorage, Alaska, April 2001, pages 319-328.

[9] Stefan Boecking, Vera Seidel and Per Vindeby, "CHANNELS: A Run-Time System for Multimedia Protocols", *International Conference on Computer*

Communications and Networks (ICCCN 1995), Las Vegas, Nevada, September 1995.

[10] Burkhard Stiller, "CogPiT - Configuration of Protocols in TIP", Computer Laboratory Technical Report TR368, University of Cambridge, June 1995.

[11] Sushil da Silva, Danilo Florissi and Yechiam Yemini, "Composing Active Services in Netscript", Position Paper, DARPA Active Networks Workshop, Tucson, Arizona, March 1998.

[12] William A. Simpson, "The Point-to-Point Protocol", RFC 1661, IETF, July 1994.

[13] Christophe Boscher, Pierrick Cheval, Liwen Wu, Eric Gray, "LDP State Machine", RFC 3215, IETF, January 2002.

[14] David Harel, "Statecharts: A visual formalism for complex systems", Science of Computer Programming 8, 1987, pages 231-274.

[15] Robert Stark, Joachim Schmid, Egon Borger, "Java and the Java Virtual Machine: Definition, Verification, Validation", Chapter 2, Springer, Berlin, 2001.

[16] Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", *ACM Transactions on Computational Logic*, Vol. 1, No. 1, July 2000, pages 77-111.

[17] Egon Borger, "Abstract State Machines at the Cusp of the Millenium", *Proceedings of International Workshop on Abstract State Machines 2000*, Springer, Berlin, 2000, pages 1-8.

[18] Leslie Lamport, "The Temporal Logic of Actions", *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pages 872-923.

[19] Jon Postel, "Internet Protocol", RFC 791, IETF, September 1981.

[20] ISO, "Information Technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1, 1998.

[21] Mark Hayden, "The Ensemble system", Ph. D. dissertation, Computer Science department, Cornell University, January 1998.

[22] Xavier Leroy, "The Objective Caml system, release 3.04", Documentation and user's manual, INRIA, France, December 2001.

[23] Robert Braden, David A. Borman, Craig Partridge, "Computing the Internet Checksum", RFC 1071, IETF, September 1988.