INTERACTIVE PROTOCOLS FOR EXTENSIBLE NETWORKING, 2000-2005

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Raid Y. Zaghal

August 2005

Dissertation written by
Raid Y. Zaghal
B.Sc., Yarmouk University, Jordan, 1993
M.S., American University, Washington DC, 1996
Ph.D., Kent State University, Kent OH, 2005

Approved by

_____, Chair, Doctoral Dissertation Committee
_____, Members, Doctoral Dissertation Committee
_____
_____
_____

Accepted by

_____, Chair, Department of Computer Science
_____, Dean, College of Arts and Sciences

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

Dedication

To Iman, Raghad and Omar,

To my mother, Zakia and the memory of my father Yousef Zaghal.

Acknowledgements

First, I praise Allah Almighty for enabling me to complete my degree and for all the bounties that He had bestowed on me and my family.

I would like to express my deepest gratitude to my advisor, Dr. Javed I. Khan, for his invaluable guidance and support throughout the past four years. He taught me how to do research, encouraged me to publish and travel, and inspired me with his endless bright ideas. I was also honored to have Professors Kenneth Batcher, Hassan Peyravi, and Mohammad K. Khan on my dissertation committee. My sincere thanks are due to all of them for their constructive comments, suggestions, and inspiration. I extend my gratitude to the faculty and staff of the Computer Science Department at Kent State University for all their efforts and support which made my academic experience at Kent State University fruitful and successful.

I owe this work to my beloved wife, Iman, whom without her help and support I would not have been able to finish. I thank her for standing besides me during the most difficult times and for her patience and understanding. I thank my lovely children Raghad and Omar for bringing light and liveliness into my life.

Last, but not least, I thank my mother Zakiah and my late father Yousef for their continues love, care, encouragement, and sacrifice, and for their strong commitment to give me the best education despite all the hardships.

CHAPTER 1

Introduction

1.1 Motivation

The layered organization of the classic *OSI reference model* has been used as a framework for designing almost every network system known today. The OSI model divides the complex task of host-to-host networking into layers, where each layer provides a specific communication service, and the collective effort of all layers ultimately provides the high level communication between the application end-points. The semantics of the OSI model emphasize the independency and separation of these layers, and thus, it draws a clear interface between these layers that allow them to exchange data and control messages in a relatively strict manner.

Another key principle that has also influenced the classical design of the Internet suggests moving specialized application-oriented functionalities up into the upper network layers and out of the core of the network. The core—which includes lower layers and covers the backbone and routers—should be kept as simple and generic as possible, and should only provide general-purpose data transfer services that can be used by all kinds of network applications. This principle is referred to in the literature as the *end-to-end arguments* [Sal84]. These classical principles are supported by the following arguments: (1) reducing the complexity of the core network which also increases its robustness, (2) increasing the generality of the network by allowing new applications to use the same core services without change, and (3) increasing the reliability of the network—if specialized application-oriented functions were built inside the core of the network, then applications will have to depend on their successful implementation and operation in the network.

It is believed that these fundamental principles, which have served as the architectural model for the Internet, are mainly responsible for the successful operation and stability of the Internet during the past 30 years. However, over the last decade, as applications became more sophisticated (streaming audio/video, e-commerce) and their communication needs have increased (more bandwidth, more security, mobility support), new requirements have emerged which are challenging these principles; on one hand, these requirements are demanding that new mechanisms and services should be added to the core of the network, and on the other hand, the current organization of network software layers seems to be too rigid for such modifications to be practically realized.

1.2 New Requirements

The emerging requirements for the Internet are mainly due to its explosive growth in terms of size, speed, number of connected users, and the diversity of applications. Here we show few such examples to demonstrate the need for new services.

1. Streaming Applications: the 'best effort' communication service that the Internet provides for any particular application does not give any guarantees regarding the quality of service (e.g., throughput and bandwidth). While some applications can tolerate variations in transmission rate or even disconnections, like FTP and e-mail, a newer type of streaming applications (e.g., audio and video) demand specific service guarantees, for example, providing a certain throughput. This has created a need to design creative solutions for the Internet to provide acceptable streaming services for such demanding applications, and at the same time to ensure that Internet resources are being used fairly by all types of applications—known in the literature as *transport-friendly*.

2. Security Needs: the growing numbers of Internet users have a wide range of motivations which may eventually lead to misuses and abuses. In addition, many newer applications that communicate highly sensitive information over the Internet (e.g., banking, e-commerce, and medical applications) need to protect their communication channels and backend servers. But, since end-points cannot be trusted

anymore, newer protection mechanisms must to be installed to deal with all kinds of security threats and attacks, and even to block undesirable forms of interaction like spam e-mails.
3. Mobile and Wireless Networks: all protocols in the classic core network were originally designed with wire-line networks in mind. In the last decade, we have seen the advent of wireless technology and the tremendous growth of wireless devices and services. New protocols/services were added to the core network (e.g., 802.11, Mobile-IP) to cope with these changes. But still, many issues are still open and need to be resolved, such as security, performance, and handoffs.

1.3 End-to-End vs. Direct Modification

Most solutions that were proposed to deal with these emerging requirements can be classified into two main approaches: (1) the *end-to-end* approach: implements the solution in the upper layers while trying to adapt to whatever 'best-effort' service the core network can provide, and (2) the *direct modification* approach: applies custom modifications (or enhancements) to the core of the network by direct implementation or by injecting customization programs.

Since the 'end-to-end' approach tries to stick to the principle of keeping the core simple and generic, it treats the core of the network as a 'black box' which cannot be altered or accessed except through the standard API. But, usually, a solution that implements a network adaptation strategy or a service extension should be aware of certain events and states within the network that cannot be 'seen' via the standard API. Networking solutions that are based on this approach usually try to compensate this limitation by employing application-level functions to estimate an approximation of these states. Unfortunately, the accuracy and timeliness of such estimations are often questionable, and sometimes they resort to redundant means that are naturally being used in the core network anyway. But the key advantage of this approach is the deployment of the solution at the upper layers. This is much easier and practical to implement and deploy—even on a large scale—since it does not require modifications in the software layers of the core network. On the other hand, the 'direct modification' approach seems to be more effective since the solution is manually implemented right inside the core of the network—so accessing the network state is not a problem. The difficulty here comes in practice; since these enhancements require customized changes within lower network layers, they are often difficult, time-consuming, and impractical. Many smart solutions that have been shown to achieve significant improvements could not make it beyond the experimental phase and were only tested in the lab or through simulation. Besides, the 'direct modification' approach clearly violates the end-to-end model, and therefore, is raising concerns among experts who want to preserve the benefits of the original Internet design.

The paradigm of *active* and *programmable* networks attempted to simplify the deployment of new services in the core of the network. They provide means to inject customized programs (or methods) into the network which in essence enables the user to 'program' the communication channel between the two end-points to fit the application's needs. In a way, this approach can be seen as diametrically opposite to the 'end-to-end' approach. Action codes are installed right into the core network where all the events (triggers) and state information are readily available. Unfortunately, active networks are still facing another set of challenges. Typically, the network system space has not been designed for multi-user execution environment, and thus, issues like resource sharing, scalability, and security have remained unresolved.

1.4 Methodology

In this work we propose a third approach which may be able to keep the best of both approaches by creating a decoupling mechanism between the *information trigger* needed to initiate adaptation (or service extension), and the actual *action code* that implements the customization. The 'direct modification' approach—as well as active networks—kept both inside the core network, while the 'end-to-end' approach kept both at the upper layers.

The fundamental idea of our approach is to perform a simple, light-weight re-organization (or *meta-engineering*) on the protocols of the core network to make them *interactive* and *transparent*. These protocols become (interactive) since they can provide *event notification* to service subscribers, and they become (transparent) since they also allow *controlled access* to their internal state information. Actual protocol extensions (or customizations) can then be performed at the application space by programmable modules called *transientware modules*. We call this mechanism *Interactive Transparent Networking*

(*InTraN*) and we label the re-organized network protocol as *InTraN*-enabled. The proposed methodology has three types of components:

1- ***InTraN*-enabled Protocol**: A meta-engineered protocol with added handles for event notification and state information exchange. The protocol designer who performs this meta-engineering designates a subset of the protocol's events (i.e., state transitions) to be subscribeable, and a subset of its state information (i.e., internal variables) to be accessible.

2- **Transientware Module**: A user-level program specifically written to provide the protocol extension or to implement adaptation. It is triggered at the application layer by event signals from the underlying *InTraN*-enable protocol, and it is provided by means to access protocol's state information.

3- **Subscription Manager**: An interface between application layer components (i.e., subscriber applications and transientware modules) and network components (i.e., *InTraN*-enabled protocols). It handles subscription requests and state information exchange operations.

A complicated adaptive solution can now be formulated by designing one or more transientware modules and binding them with events from the *InTraN*-enabled kernel. These modules can then pull-up the protocol's state information needed for adaptation or service extension, perform the required action, and if needed push-down any results or state updates. The *Subscription Manager* manages all correspondence (subscription, signaling, read state, and write state) between the network kernel and the application-level components and imposes safety measures to ensure the stability and correctness of the system.

1.5 Main Contributions

The *InTraN* paradigm offers a number of unique features that can be considered as the main contributions of this work:

1- **Implementation path via application layer**: *InTraN* allows kernel-level enhancements (or modifications) to be performed at the application layer, which is especially important since it opens a more practical implementation path for such modifications to be realized—which otherwise would have been performed inside the core of the network. This relieves lower network layers from housing costly custom components, and thus, it preserves the benefits of the 'end-to-end' model by keeping the core simple and generic. Also, it becomes more effective to handle other complex issues like security and resource sharing. The attraction is that the application space has plenty of means to deal with these issues effectively—much of that can be reused.

2- **Light-weight core design**: Although *InTraN* still requires some re-organization of lower network protocols to facilitate event notification and state information exchange, but as we will show, it is much lighter than the re-organization needed to run the customized actions inside the network.

3- **Small inter-component communication overhead**: The *InTraN* paradigm still incurs some overhead in terms of signaling and state information exchange between the transientware and the kernel. Though, we will show by real measurements that this overhead is very small—even negligible in some cases. Therefore, the performance of the *InTraN* paradigm is expected to be no less than that of active networks, and even much faster than the 'end-to-end' approach since the state information can now be retrieved directly from the local end-point.

4- **Backward compatibility**: we have designed the *InTraN* paradigm to comply with the following three principles for backward compatibility: (i) the *InTraN*-enabled version of a protocol remains functionally compatible with legacy silent versions, (ii) the API is an extended set, and thus classical applications remains fully usable with the interactive versions of the end-point components, and (iii) the meta-engineering of a protocol does not change its original dynamics, and thus, the dynamics of the network.

1.6 Solution Classes

The distinguished features of the *InTraN* paradigm can support the following solution classes for general networking problem solving: (1) *application adaptation*, (2) *cross-layer optimization*, and (3) *protocol extension*. We have realized a FreeBSD implementation of *InTraN* and used it to design a novel solution for the first two types:

1- **Application adaptation**: We have designed a TCP-friendly, congestion management scheme for time-sensitive elastic traffic. The scheme allows a video transcoder to adjust its sending bit rate in real-time based on the feedback (loss events and TCP state updates) it receives from the *InTraN*-enabled TCP (or iTCP). The scheme exposes the overall benefits of application adaptation for time-sensitive traffic, and takes a different approach to achieve true TCP-friendly traffic where both the application and the network cooperate to recover from congestion.

2- **Cross-layer optimization**: We have designed a connection-oriented mobility scheme for IP networks. In this scheme, a smart employment of *InTraN* by three layers (namely: Link layer, IP, and TCP) was able to (i) freeze the TCP connection right after L2 handoff has started, (ii) perform handoff on the IP level directly by updating the actual IP addresses on both endpoints (i.e., mobile node and correspondent node), and (iii) resume the connection on the TCP level right after handoff has finished. This scheme offers a number of benefits over conventional Mobile-IP such as faster handoffs and direct triangulation-free routing.

3- **Protocol extension**: In the literature, many networking protocols have been manually extended (or modified) to cope with emerging communication needs. We have chosen two such modifications that were proposed to improve TCP performance over mobile and wireless networks, namely, Snoop [Bal95] and WTCP [SiV99], and then we illustrated how to transform them into an *InTraN*-enabled version where the protocol extension is implemented as application layer transientware.

1.7 Dissertation Outline

In chapter 2, we preview related works. In chapter 3, we present a formal EFSM-based framework for the proposed meta-engineering and relevant issues like interfacing and security. In chapter 4 we illustrate the principles of *InTraN* meta-engineering by showing a real example based on the TCP protocol; first, we discuss the congestion control model of classic TCP, and then we present an SDL description of a simplified TCP and its *InTraN* extension—we call the new protocol iTCP. In chapter 5, first we show relevant implementation details of iTCP, and then we design a *transientware* solution for a TCP-friendly elastic video traffic. The solution also includes an adaptive video transcoder that can adjust its transmission rate based on feedback signals from iTCP. In chapter 6, we present our second *InTraN*-based solution—IPMN. This is a mobility scheme for IP networks that can provide loss-free, rapid handoffs and eliminates triangular routing. It chapters 5 and 6 we also present extensive experimental results and performance analysis for both projects (iTCP and IPMN). In chapter 7 we show how the *InTraN* paradigm can be used to model other solutions or protocol extensions. Here we show a modeling examples of two well-known protocols proposed in the literature to improve TCP performance over wireless networks: Snoop [Bal95] and WTCP [SiV99]. In chapter 8 we give concluding remarks.

CHAPTER 2

Related Works

We have selected two main paradigms from the literature that were proposed to address the issue of protocol reconfiguration and network service extension. These are the *Programmable and Active Networks* paradigm, and the *Protocol Composition* paradigm.

2.1 Programmable and Active Networks

Introducing new services into the existing 'best effort' networks was usually a manual, time consuming, and costly process. *Programmable Networks* were proposed to simplify the deployment of new network services, leading to extensible networks that explicitly support service creation and deployment. Programmable networks architectures provide programmable interfaces that can support a variety of service composition methodologies. In *Active Networks*, service delivery and control is achieved through code mobility. A number of research groups have been developing programmable network prototypes, with each group focusing on different set of characteristics [Cam99], namely (1) *networking technology*, (2) *level of programmability*, and (3) *communications abstractions*. Below; we briefly discuss these three categories and in each one, we preview some of its most prominent prototype implementations.

2.1.1 Networking Technology

Different programmable network projects have been designed to target certain networking technologies which ultimately decide the type of programmability that can be carried to the higher levels. By making the targeted networking technology more programmable, it becomes easier to overcome particular deficiencies in the communication services supported by that technology. For example, xbid [Chan96] by Chan, et al, was designed for ATM technology to support better QOS features like admission control and resource reservation. By separating control algorithms from the hardware, xbid was able to provide interfaces that allow open access to node resources and functions. Another example is Smart Packets [Kul98] by Kulkarni, et al, which introduced a code-based packet concept to create programmable IP environment.

2.1.2 Level of Programmability

New services can be established into the network with a range of methodologies and granularities. Programmability level can vary from highly dynamic (e.g., capsules [Ten96]) to highly conservative models (e.g., RPC interfaces [Vin97]). Among the most prominent works is ANTS [Wet98] by Wetherall, et al, which provides a set of core services (transportation of mobile code, loading of code on demand and caching techniques) that facilitates the introduction or extension of existing network protocols, these in turn can be used to introduce programmable network services such as enhanced multicast, mobile IP routing and application level filtering. In ANTS, Capsules serve as atomic units for network programmability that can support processing and forwarding interfaces. Other proposals put more focus on security requirements such as Switchware [Alex98] by Alexander, et al. In this prototype, a component in the active router allows active extensions to be safely loaded via a set of secure methods such as encryption, authentication and program verification. A thirds example is the CANEs project [CANE] which provides composition methods (programming languages with enhanced language capabilities) to construct composite network services from components.

2.1.3 Communications Abstractions

The programmability of network infrastructure can enable different levels of virtualization (i.e., virtual middleware and node support). Communications abstractions include programmable virtual routers, virtual links and mobile channels. Among these, is a node operating system called NodeOS [Pet99] by L. Peterson which represents the lowest level of DARPA's architectural framework for active networking [Calv98]. NodeOS provides node kernel interfaces at routers that enable them to host multiple execution environments (EEs). These EEs support communication abstractions such as threads, channels and flows. The architectural framework for active networking is being implemented in the ABone testbed [ABone]. Another example is the Netscript project [Yem96] which takes a functional language-based approach to capture network programmability using universal language abstractions. Netscript supports Virtual Active Networks as programmable abstractions that can be systematically composed, provisioned and managed.

2.2 Protocol Composition

This paradigm suggests designing a new networking infrastructure that supports creating complex protocols from smaller off-the-shelf components. The composition can be perceived as a whole *middleware* offering complex services for distributed applications. These services include: (1) providing communication abstractions (e.g., reliable multicast, mobility support), (2) allowing adaptation (e.g., switching protocols to overcome a security threat, changing data rates to accommodate a slower link), and (3) supporting the creation (and coordination) of multiple communication channels with different QOS requirements.

The protocol composition paradigm offers a number of advantages over traditional monolithic approaches [Bir87] [Dol96] [Mal96], such as, higher configurability, reusability, and extensibility. [Men03] has classified protocol composition frameworks that have been proposed in the literature into two families; the first family contains the x-Kernel [Hut91], and its successors Coyote [Bhat96], [Bha98] and Cactus [Hil98], and the he second family contains Horus [Van93] [Van96] and its successors Ensemble [Hay98], Appia [Mir99] [Mir01] and JavaGroups [Ban02].

The x-Kernel [Hut91] is an early and influential work on protocol composition and was the first to propose building a system in which protocol layers could be arbitrarily configured. A main feature of the x-Kernel is its support for a uniform interface to all protocols which allows two protocols providing the same semantics to substitute each other. However, the x-Kernel had a few shortcomings; for example, configuration was done before system compilation and not at run-time. Also, the x-Kernel was intended mostly for point-to-point communication, and had limited support for dynamic membership. Cactus [Hil98] is an evolution of the x-Kernel that inherits and extends its composition and concurrency model to provide a finer-grain level of composition. In Cactus, the internal structure of an x-kernel protocol consists of the composition of several protocols (called micro-protocols). These protocols are event-driven and their composition is not hierarchical, allowing them to directly interact without artificial restrictions imposed by protocol stack hierarchy. Cactus allows several event handlers to be bound to the same event so that all these handlers are executed upon occurrence of this event.

In Horus [Van93] [Van96] and Ensemble [Hay98], protocol layers can be arbitrarily stacked in a variety of ways, and thus, they were able to offer more flexible and configurable group communication support for distributed applications. Both frameworks use a single generic architecture and separate the basic group communication interfaces from their implementations. This configuration enables the designer to plug-in certain implementations that match the specific needs of the application, and also to arrange a stack of micro-protocols that provides the needed properties (or service guarantees). Appia [Mir99] is a re-engineering of Ensemble and it inherits all its features, but its composition model has been extended to offer more flexibility. In Appia, as in Ensemble, protocol modules are composed on top of each other to form a stack. The main difference is the possibility, in Appia, to have more than one protocol module at the same level in the stack.

2.3 Discussion

Despite the fact that many of these frameworks were able to achieve their goals in providing complex services and creating communications abstractions, they are still facing critical challenges in terms of

security, complexity, and scalability. For example, a weak protocol module design may incur a big overhead cost that surpasses the overall advantage of the protocol composition system, also, a really complex middleware may become difficult to maintain and scale-up as the number of group members grows substantially. Although the *InTraN* paradigm will still face the same challenges, but we believe that due to its light-weight, structural design, these challenges will be much easier to handle. Though, it will still require careful design especially with the transientware.

CHAPTER 3

Interactive Transparent Networking (InTraN)

3.1 Background

The proposed interactivity and transparency is achieved via formal meta-engineering of the network protocols so that a selected subset of their states can be engineered to be accessible by upper-layer service subscribers in a controlled manner. We use SDL (Specification and Description Language) [Ell97, SDLfrm] to formally describe (a) the protocol meta-engineering process, and (b) the network software organization needed to support interactivity and transparency. In this chapter, we first give some background information on SDL, and then we discuss the *InTraN* framework and its security model.

SDL (Specification and Description Language) is an ITU-standardized language for the formal description of communication protocols. It is also suited for any application based on the finite state machine concept, such as circuit design. The programming model used by SDL is based on extended finite state machines (EFSM) [Ell97, Byu01]. SDL augments the finite state machine model by providing variables and timers and by supporting object-oriented programming. We describe the protocol meta-engineering mechanism of *InTraN* by assuming an abstract communication protocol whose behavior is described by an EFSM. We demonstrate how *InTraN* exposes protocol's internal state to achieve controlled yet secure transparency. Informally, the EFSM is composed of states and transitions among them. For a transition to occur, the system must receive an event from the environment which triggers corresponding actions. After performing the actions, the EFSM produces output signals to the environment. An SDL system is composed of several protocol entities; each entity is designed as a single EFSM. Formally, An EFSM is a 6-tuple (S, s_0 , E, f, O, V), where S is a set of states, s_0 is an initial state, E is a set of events, f is a state transition function, O is a set of output signals, and V is a set of variables. The function f returns a next state, a set of output signals, and an action list for each combination of a current state and an input event. An EFSM also uses predicates to control the behavior of the protocol. These predicates usually allow similar states to be grouped therefore reducing the total number of states [Ell97]. Upon receiving an event, the machine checks a predicate that is composed of variables, logical operators

Table 1. Main components of the *InTraN* framework

Component	Definition
Protocol Entity (*PE*)	A communication protocol instance that provides specific communication service in the protocol stack (e.g., TCP). It is described as an EFSM and has been meta-engineered according to the *InTraN* paradigm—we use PE and EFSM interchangeably in the text.
Subscriber Program (*SP*)	A user program that uses network services (e.g., video server). It is regarded as a potential subscriber of the *InTraN* service.
Transientware Module (*TM*)	A piece of code that is specifically designed to handle one or more events in a certain *PE*. One or more *TM*s can implement a protocol modification/extension at the application layer instead of embedding the code in the network layer itself.
Subscription Manager (SM)	An interface between application layer components (i.e., *SP*s, *TM*s) and network components (i.e., *PE*s). One *SM* manages the subscription preferences of a single *SP*. It handles subscription requests, maintains updated information about active *TM*s, and handles their read/write requests.

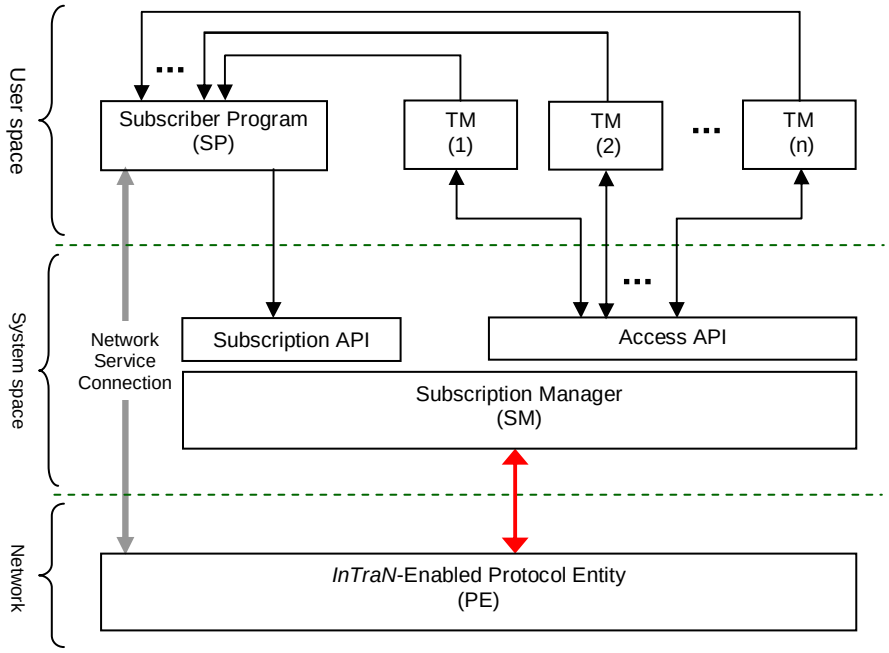

Figure 1. *InTraN* basic methodology

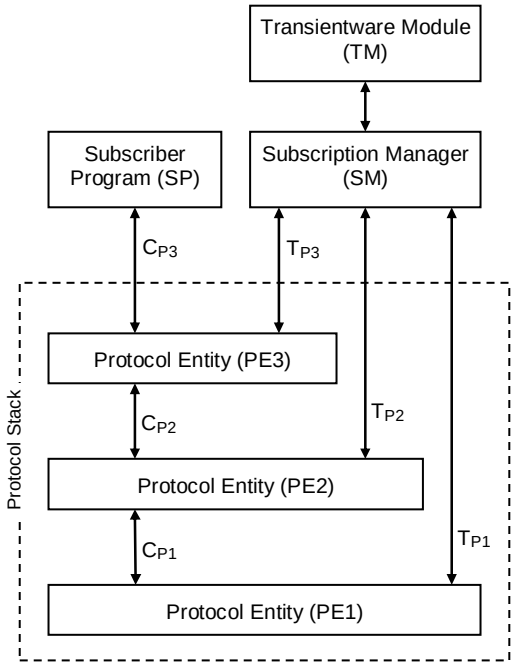

Figure 2. *T-type* channel extension

(e.g., AND, OR), and relational operators (e.g., <, =, >). If a predicate is true, the EFSM performs the actions and produces output signals (if applicable).

3.2 A Framework for the *InTraN* paradigm

3.2.1 Components and Architecture

The main components of the *InTraN* framework are shown in Table 1 and its basic architecture is shown in Figure 1. A Subscriber Program (*SP*) starts by binding an event in a specific *PE* with a *TM* via a special *Subscription API*. The *SM* maintains updated information about all active subscriptions. When a subscribed event occurs in a *PE*, it signals the *SM* which responds by activating the *TM* bound to the event. A special *Access API* allows active *TM*s to access *PE*'s internal data through the *SM*.

According to the SDL language, EFSMs can communicate only through specific channels. Protocol Entities (*PE*s) can perform input and output operations to exchange user data and control messages through these channels. In order to integrate *InTraN* in this setup, we need to create a communication channel between every *PE* and the Subscription Manager (*SM*). These channels will serve as interaction mediums between *PE*s and *TM*s through the *SM*. Figure 2 shows the basic architecture of an abstract system with a

Table 2. Types of variables and their access privileges

Variable Type	Set	TM access privilege
A	$V_P - V'_P$	No access
B	$V'_P \subseteq V_P$	Read only
C	$V''_P \subseteq V'_P$	Read and write

Table 3. Subscription API

Primitive	Meaning
Bind*(e, P, T)*	Associates a *TM* with an event *e* in protocol *P.* The *TM T* is invoked whenever the specified event occurs.
Unbind*(e, P, T)*	Remove the association between the *TM T* and the event *e*.
Update*(e, P, T)*	Remove the current association of event *e* and replace it with a new association with the *TM T*.

Table 4. Types of Transientware Modules

TM Type	Definition
Signal-Only	If *TM* T_i is bound to an event e_i in protocol *P*. When event e_i occurs, T_i is only activated. It is not allowed to access protocol's internal variables. No *TM-instance* record is created for T_i in the *SM*.
Read-Only	If *TM* T_i is bound to an event e_i in protocol *P*. When event e_i occurs, T_i is activated and a *TM-instance* record is created for T_i in the *SM*. T_i is granted read-only access to readable variables in *P* (i.e., all variables $v \in V'_P$).
Read-Write	Same as *Signal-Only* mode, but in addition to that, T_i is granted write access to modifiable variables in *P* (i.e., all variables $v \in V''_P$).

stack of three protocols. Normal information flow from/to user application goes through channels (C_{P3}, C_{P2}, and C_{P1}), to augment with *InTraN*, we added channels (T_{P3}, T_{P2}, and T_{P1}). These new channels—which we call *T-type* channels—are used by *PE*s to pass event signals and exchange data between *PE*s and the *SM*. The *T-type* channel is defined in Table 6.

*TM*s are also classified into three types based on their access privileges to protocol's internal variables. These types are described in Table 4. A *TM* is granted read-only access to a subset of *PE*'s local data. In certain circumstances the *TM* is allowed even to modify a subset of these accessible variables as long as

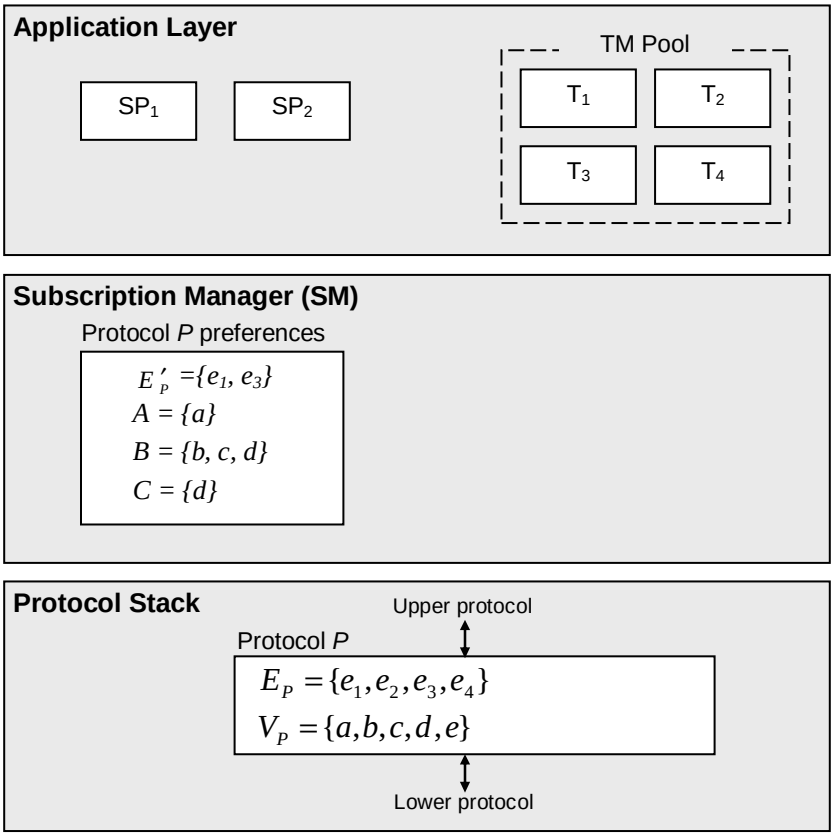

Figure 3. Subscription example

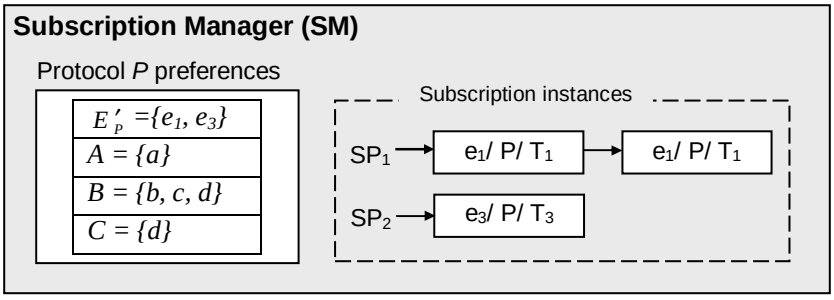

Figure 4. *SM* state after performing the four operations

this modification serves the intentions of the protocol designer. Let V_P be the set of all variables in the *PE*, the designer can designate a subset of V_P called V_P' as *read-only*, and a subset of V_P' called V_P'' as *read-write* (i.e., $V_P'' \subseteq V_P' \subseteq V_P$). In Table 2 we define three types of variables: *A*, *B*, and *C*, based on their access level. In addition, the protocol designer should designate a subset of protocol's events as subscribable. Let E_P be the set of all events in protocol entity *P*, and E_P' be the set of subscribable events in *P*, then $E_P' \subseteq E_P$.

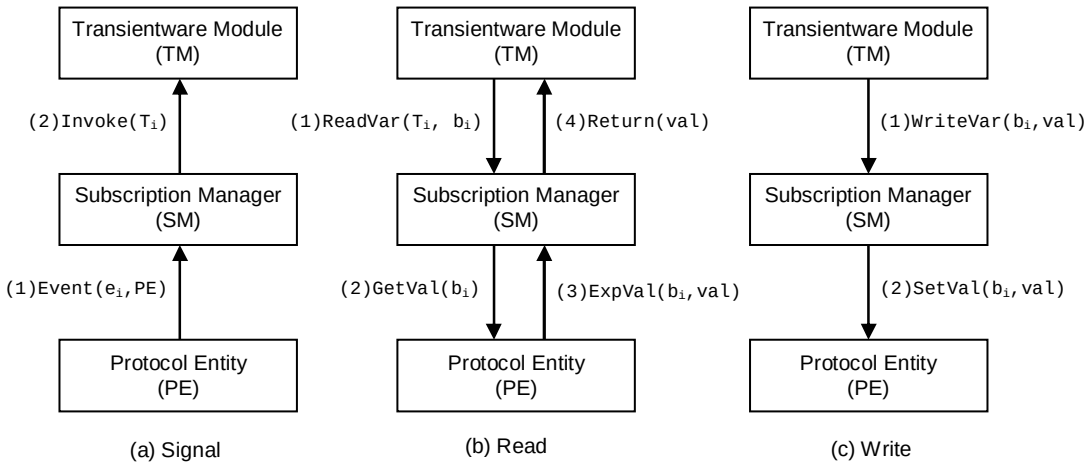

Figure 5. Interfacing between the *PE* and the*TM*

3.2.2 SP-SM Interfacing: Subscription Mechanism

The *InTraN* paradigm offers a *Subscription API* for *SP*s to manipulate their subscription preferences at the *SM*. The three primitives of the *Subscription API* are shown in Table 3. A Subscriber Program (*SP*) which opts to subscribe with protocol entity *P* must associate an event in E'_P with a *TM* via the Bind() operation. The binding between events and *TM*s is *one-to-many* relationship. i.e., a *SP* can bind one or more events to a specific *TM*, but a specific event can be bound to one *TM* only by a specific *SP*. This restriction is needed to avoid ambiguity when event signals are sent to the *SM*. The *SP* can use the Unbind() operation to cancel an existing subscription, or the Update() operation to replace the current association of an event with a new one. The three subscription primitives can be used dynamically during run-time for maximum flexibility. For example, a *SP* can start by binding e_1 to TM_1 by calling Bind(e_1,*P*,TM_1). Later (e.g., after a certain time has elapsed), it may call Update(e_1,*P*,TM_2) to change the association of e_1 from TM_1 to TM_2.

Example: We present a simple example in Figure 3 to illustrate these concepts. The figure shows a system with two subscriber programs (SP_1 and SP_2) and a pool of four *TM*s (T_1, T_2, T_3, and T_4). In this example we only highlight one protocol (*P*) from the protocol stack. Therefore, we assume that all four *TM*s can be bound to subscribable events in *P*. The Subscription Manager (*SM*) maintains the subscription preferences of *P*—among other protocols in the stack as well. *P* has four events and five local variables shown in E_P and V_P respectively. Among these, two events are subscribable (E'_P), three variables are read-only accessible (type B), and only one variable is modifiable (type C). Assuming the following operations were performed in this order by their respective *SP*s:

1. SP_1: Bind(e_1, *P*, T_1)
2. SP_2: Bind(e_3, *P*, T_2)
3. SP_1: Bind(e_3, *P*, T_1)
4. SP_2: Update(e_3, *P*, T_3)

Figure 4 shows the state of the *SM* after these four operations are performed. Here we show two subscription threads for the two *SP*s represented as linked lists for easy update. A record in the list represents a live subscription instance which creates the binding between a protocol event and a *TM*. Notice that the Update() operation has replaced the binding of e_3 on the SP_2 thread from TM_2 to TM_3.

3.2.3 TM-SM-PE Interfacing: Access Mechanism

All communication between the *TM* and the *PE* must go through the *SM*. The *SM* provides the interfacing between all *TM*s and the *PE*s through a special *Access API* and *Signals*—these are shown in

Table 5. *InTraN* Access API and Signals

Access API (TM-SM interface)	
ReadVar*(T, V)*	The *TM* (*T*) issues a read request to the *SM* to retrieve the value of variable (*V*) from its correspondent *PE*.
WriteVar*(T, V, val)*	The *TM* issues a write request to the *SM* to write the value (*val*) to the variable (*V*) in its correspondent *PE*.
Return*(val, F)*	The *SM* returns the value (*val*) of a variable (*V*) to a *TM* which is blocking on a ReadVar() request. If the Boolean flag (*F*) is (true), then (*val*) is valid, otherwise, the *TM* just ignores (*val*).
Return*(F)*	The *SM* returns a feedback to the *TM* that has issued a WriteVar() request. If the Boolean flag (*F*) is (true), this indicates a successful write operation, otherwise, it indicates a failed write operation.
Invoke(*T*)	The *SM* invokes a registered *TM* (*T*) after receiving an *Event()* signal from a *PE*.
Finish(*T*)	The *TM* (*T*) informs the *SM* that it is going to terminate. The *SM* responds by removing the *TM-instance* of the terminating *TM*.
T-type Channel Signals (SM-PE interface)	
GetVal*(V)*	The *SM* signals the *PE* to read the value of the local variable (*V*)
SetVal*(V, val)*	The *SM* signals the *PE* to write the value (*val*) to the local variable (*V*)
SetFlag*(SF, val)*	The *SM* signals the *PE* to set the subscription flag (*SF*) by sending (*val=true*) or to reset the flag (*SF*) by sending (*val=false*). This signal will enable/disable the event that is associated with (*SF*).
Event*(evt, PE)*	The *PE* Notifies the *SM* that event (*evt*) has just occurred in protocol (*PE*)
ExpVal*(V, val)*	The *PE* exports the value (*val*) of local variable (*V*) to the *SM*

Table 6. Definitions

Name	**Definition**
T-type channel	A private bidirectional channel that connects every *PE* in the system with the *SM*. Every T-type channel has a unique name (T_P) where P is the protocol connected to the *SM* through this channel.
TM-instance	A record created by the *SM* whenever a new *TM* process is activated. The TM-instance enables the *SM* to handle future read/write requests that might be made by the *TM* to access the protocol's local variables. The *SM* stores the following information in a TM-instance: a) The process ID of the *TM*. b) The name of the *T-type* channel connecting the *SM* to the target *PE*. c) Temporary copies of protocol's variables targeted by read/write requests.

Table 5. We impose this mode of communication to preserve the integrity of the system and to let the *SM* enforce access privileges as specified by the designer.

Figure 5 explains the interfacing provided by the *SM*. The figure shows the sequence of operations that gets executed when (a) a *PE* issues an Event() signal, (b) a *TM* issues a ReadVar() request, and (c) a *TM* issues a WriteVar() request. We explain the three scenarios below:

(a) TM Invocation and Termination

When a subscribed event (signal) is consumed in the EFSM of a *PE* (*P*), the signal Event(e_i, *P*) is sent to the *SM* indicating the event type and the protocol. The *SM* searches its subscription lists to find the *TM* that is currently bound to such (event, protocol) pair. Assuming a *TM* (T_i) was found, the SM activates T_i via the Invoke(T_i) operation. Whenever the *SM* activates a *TM*, it also creates a record in its data store that we call (*TM-Instance*) to be able to handle any future requests that might be made by the *TM*—the *TM-Instance* is defined in Table 6. When the *TM* finishes, and before it is terminated, it sends a Finish(*TM*) message to the *SM*. The *SM* then removes the *TM-Instance* record of the terminating

TM.

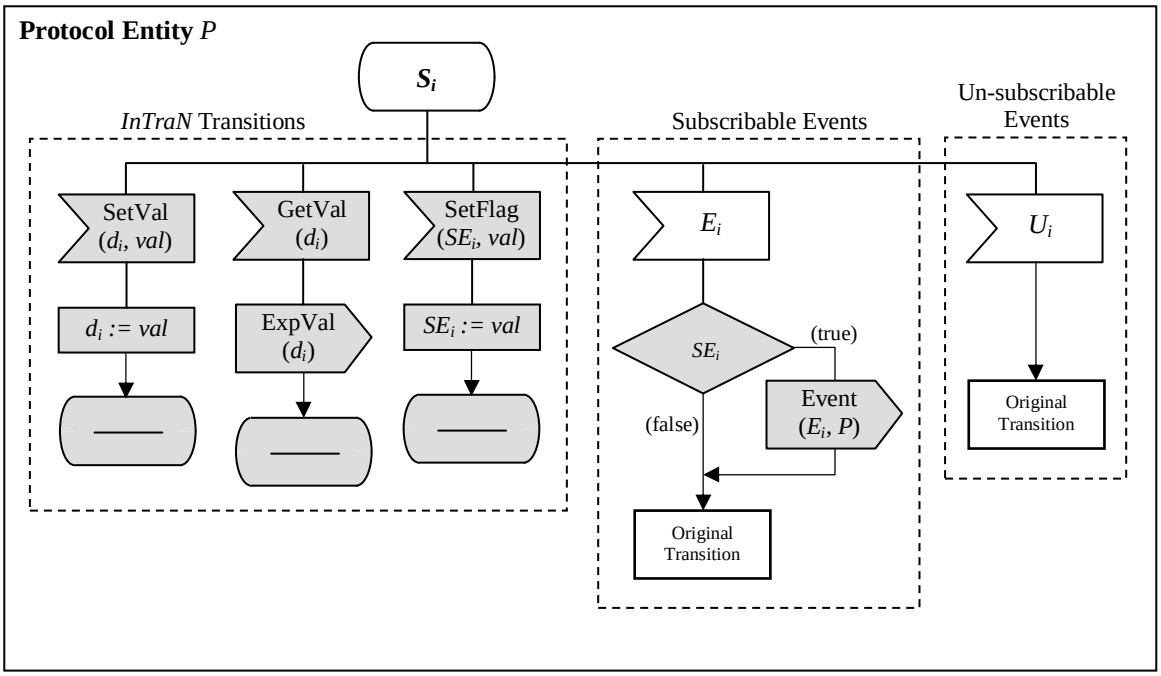

Figure 6. Protocol meta-engineering extension

(b) Read Access

When a *TM* (T_i) wants to read the value of a certain variable v_i from the underlying *PE*, it sends a ReadVar(T_i, v_i) request to the *SM*, then it blocks waiting for the value of v_i. The *SM* checks if the requested value is accessible (i.e., $v_i \in V_p'$) and if T_i is eligible to issue a read request (i.e., it is *Read-Only* or *Read-Write* type). If this is true, the *SM* issues a GetVal(v_i) signal to the *PE* specifying the name of the requested variable, otherwise it replies with a Return(-1, *false*) to T_i. When the *PE* receives a GetVal(v_i) signal it returns the value of v_i to the *SM* via a signal ExpVal(*val*). The *SM* then forwards the value *val* to T_i via a Return(*val*, *true*) operation.

(c) Write Access

As we mentioned earlier, some *TM*s can modify certain variables in the EFSM of the *PE*. If a variable v is modifiable (i.e., $v \in V_p''$), then, its value can be overwritten by a *Read-Write*-type *TM*. However, the protocol designer should be careful when choosing the members of V_p'' in each *PE*. Technically, since a *TM* in the *InTraN* paradigm represents a soft alternative for hardcode protocol modifications, this relaxation should make *TM*s even more dynamic and powerful. On the EFSM level of the *PE*, modifying a variable can trigger a state transition; this, of course, should reflect the designer's intention. Therefore, protocol modifications can be realized through a group of carefully designed *TM*s which can manipulate certain properties of the EFSM through interaction, i.e., (reading from) and (writing to) protocol's local variables. As with the reading case, writing to *PE*'s local variables must go through the *SM*. A *TM* (T_i) makes a write request by passing the variable name and its new value to the *SM* via a WriteVar(T_i, v_i, *val*) operation. If v_i is modifiable and T_i is *Read-Write* type, the *SM* generates a signal SetVal(v_i, *val*) to the *PE* and issues a Return(*true*) message to T_i. Otherwise, it issues a Return(*false*) message to T_i indicating a failed write operation. When the EFSM of the *PE* consumes the SetVal() signal, it simply runs the assignment $v_i := val$.

3.2.4 Protocol Meta-Engineering

The meta-engineering of a *PE* involves adding new events and transitions to its EFSM. Basically, the *SM* should be able to tell the *PE* which events in its E'_P set are currently subscribed by *SP*s. These events will be marked in the EFSM, so that, whenever any one of them occurs, the EFSM sends a signal to the *SM* over its *T-type* channel.

Figure 6 depicts the necessary meta-engineering of the EFSM of any classical protocol entity *P* in order to make it *InTran* enabled—new components are shown in shaded SDL symbols. Let S_i be any state in *P*, E_i be any subscribable event, and U_i be any un-subscribable event, then the following components are added to the EFSM:

- A new transition triggered by the signal SetVal(d_i, *val*).
- A new transition triggered by the signal GetVal(d_i).
- A new transition triggered by the signal SetFlag(E_i, *val*)
- For every E_i a Boolean flag (SE_i) is created in *P* to remember the current subscription status of E_i. SE_i is set to *true* if E_i is currently subscribed. We augment the transition of E_i right after the SDL *input symbol* as shown in Figure 6. After consuming E_i, the EFSM checks the associated subscription flag (SE_i) of the consumed event. If SE_i = *true* (i.e., an *SE* is currently subscribed to E_i), the EFSM outputs the signal Event(E_i, *P*) to the *SM*. Otherwise, no action is taken.

The *SM* uses the SetFlag() signal to manage subscription flags (i.e., SE_i flags) as follows: Assume an *SP* made the subscription: Bind(E_i, P_j, TM_k), the *SM* registers this subscription instance in its internal data store, and then it checks if there are other *SP*s currently subscribed to E_i. If no active subscription instance is found, the *SM* sends the signal SetFlag(E_i, *true*) to the EFSM of protocol P_j. When the EFSM consumes this signal, it enables E_i signaling by setting the subscription flag SE_i associated with E_i to *true*. However, if the *SM* does find at least one active subscription instance to E_i in its data store, this indicates that E_i signaling is already enabled in the EFSM, and therefore the *SM* takes no further action. Conversely, if an *SP* made Unbind(E_i, P_j, TM_k), the *SM* updates its internal data store, and also checks if any *SP* is still subscribed to E_i after executing the Unbind(). If at least one such instance is found, the *SM* takes no further action, but if the Unbind() has caused the last subscription instance of E_i to be deleted from the data store, the *SM* sends the signal SetFlag(E_i, *false*) to the EFSM of protocol P_j to disable the signaling service of E_i. The SetVal() and GetVal() signals correspond to the write-access and read-access operations which were described in the previous sub-section.

3.2.5 Security Model

Since the *InTraN* framework exposes the internal state of the protocol to entities running in the user space (i.e., *TM*s), it must address the *correctness* and *safety* issues of the underlying protocol appropriately. We can claim that access modes that only involve *signaling* or *reading* are safe (i.e., *Signal-Only* and *Read-Only TM*s) since they do not alter protocol's internal state. We have to be concerned only when a *TM* is allowed to write to protocol's variables (i.e., *Read-Write* mode). Here, we propose a security model which allows controlled access to protocol's internal variables and at the same time maintains system stability. We define two types of designers who can be involved in any *InTraN*-based solution: (1) protocol designer, and (2) *TM* designer. The protocol designer must be a super-user. He basically performs the meta-engineering on protocol entities. This includes, deciding the three classes of protocol's variables (*A*, *B*, and *C*), identifying subscribable events (i.e., E'_P), and extending the EFSM by adding *InTraN* components as in Figure 6. The *TM* designer can be any user; he simply implements a particular protocol solution/extension by coding one or more *TM*s. He uses the services offered by the underlying *InTraN*-enabled system through the *Access API* to implement the intended solution.

Only when a *Read-Write* type *TM* tries to update a *C* type variable, then system stability can be compromised—we define this combination as the *dangerous combination.* The danger may come from two sources: (1) a flaw in the protocol design, and (2) a malicious *TM* of type *Read-Write*. When a system is running with a dangerous combination, the operating system activates a *guarding* program that verifies any attempts made by *TM*s to update *C* type variables. If the update is safe, it is allowed to proceed. But, if the update may cause instability in the system (i.e., it is attempting to change a *timer* or *index* variable in the protocol) then the write operation is blocked immediately and the offending *TM* is shut down. The guarding program itself is simple and can be implemented as utility program that belongs to the operating system.

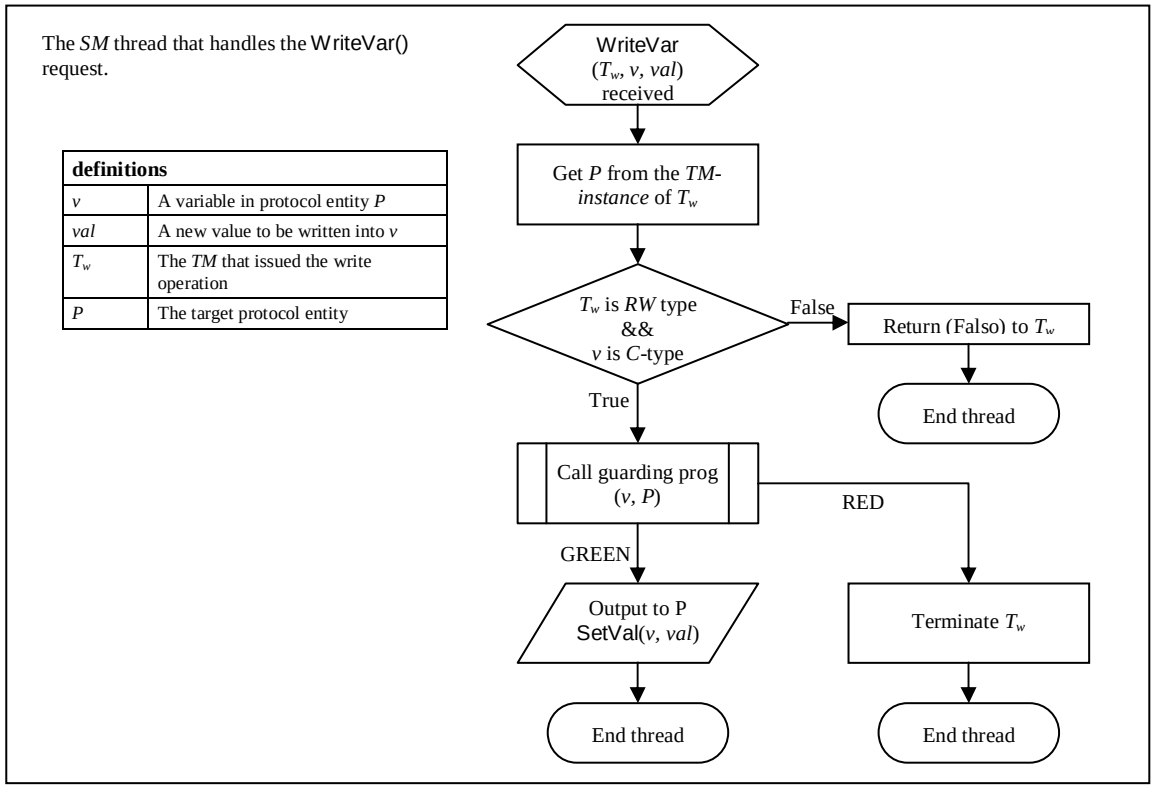

Figure 7. *SM* handling of the WriteVar() operation

Basically, it needs to know which updates on any *PE*'s internal variables are safe and which are not regardless of protocol designer classifications in Table 2. This way, the integrity of the *InTraN*-enabled system can be preserved even in the presence of design flaws.

What are the performance implications of this added security? We can show that by careful implementation the overhead should be very small. Here, we propose an implementation path using event-driven run-time screening, but other choices can be taken as well, such as static analysis of the *TM* source code (similar to that of [Hau04]). The *SM* can be programmed to initiate a special thread program to handle the WriteVar() operation and the *dangerous combination*. Figure 7 describes the basic algorithm; Assuming a *TM* called (T_w) has issued the following write operation: WriteVal(T_w, *v, val*). First, the *SM* consults the *TM-instance* of T_w to retrieve the protocol entity *P* associated with it. Next, this operation must pass the initial screening at the *SM* (i.e., the *SM* checks if T_w is a *Read-Write* type *TM* and *v* is a *C* type variable). If the write operation passes this test successfully, then the *SM* invokes the guarding program to perform a second-level independent screening and waits for its decision. The *SM* passes two parameters to the guarding program: target variable *v* and target protocol entity *P*. If the guarding program finds that this write operation is safe, it sends a GREEN signal to the *SM* to allow it, the *SM* then continues normally by issuing a SetVal(*v, value*) signal to *P*. Otherwise (i.e., the write operation is not safe), it sends a RED signal to the *SM* which responds by canceling the write operation and shutting down T_w. Let *N* be the number of *PE*s and let *K* be the maximum number of unsafe variable updates in any *PE*. Then, the guarding program will make *O(N+K)* comparisons in the worst case.

3.3 Conclusion

The proposed *InTraN* meta-engineering presented in this chapter has a number of distinguishing features; first, it is light-weight and requires only limited changes (and additions) on the original protocol. Secondly, it is generic and can be applied to any protocol as far as it has some state information and programmable interface, and finally, it leaves the *InTraN*-enabled protocol fully compatible with legacy

network components and non-subscribing applications. Furthermore, it allows maximum flexibility since it puts most of the work in programmable components that can be updated or changed at anytime.

CHAPTER 4

iTCP part I: InTraN Meta-Engineering

4.1 Introduction

With the advent of advanced applications and their advanced transport needs current transport services are increasingly becoming inadequate. This inadequacy has also prompted recent attempts towards recreating new and more complex functionalities inside the network or system middle layers. For example, Congestion Manager [And00], [Bal99] is a system layer component that provisions aggregate congestion control when multiple streams from the same endpoint attempt to send. Unfortunately, majority of these—though they offer specific functional advantages—enormously increase the network or system layer complexity. Such complex permanent addition to the network software appears questionable. When the complexities of such solutions are weighted against their general advantage over a broad range of applications, they do not seem to be gaining any acceptance. Due to the same inadequacy, in the past few years it has also been felt that for advanced applications (e.g., real-time streaming), it is better to engage the applications themselves in the solution. Particularly promising are the research in the new TCP friendly paradigm [Pra00], [Rej00], [Sis98]. Due to the lack of convenient means to obtain real-time information about network state, these systems had to rely exclusively on application layer techniques to compensate for the network impairment. Several works such as [Bri99], [Wol97] suggested sending multilevel redundant information which will eventually increase the burden on the network. Also, due to the inherent round trip delay involved, adaptation time can be unbearable for more time-critical applications. Overall, it is very difficult to build a network friendly application if the network itself is non-friendly and unwilling to interact.

A particular problem we address with iTCP is the congestion management and particularly the one for time-sensitive streaming traffic. Most of the network level schemes for congestion control are based on delaying traffic at various network points. The more classical schemes depend on numerous variants of packet dropping in network, prioritization (graceful delay in router buffer), admission control (delaying at network egress points), etc. However, a key aspect to note in all is that they introduce *time distortion* in the transport pathway of the application. Though this is harmless to time-insensitive traffic such as email or FTP, but they distort the temporal characteristics of time-sensitive traffic such as multimedia streaming or control data. Recent solutions are also based on complex network or system layer addition (such as [And00]). We demonstrate a simple *InTraN*-based congestion management scheme for time-sensitive elastic traffic. In contrast to network or system layer solutions, the general principle we follow is simple and intuitive; it seems an effective delay conformant solution for time-sensitive traffic may be designed if the original data volume can be reduced by its originator—the application.

To demonstrate the efficacy of the principle, we have also designed a corresponding advanced video rate transcoder system [Kha01] that works in symbiosis with the network. This transcoder actively participates in a custom symbiotic back-off scheme in the application layer with deep application level knowledge resulting in much more effective joint quality/delay sensitive communication. The adaptation is applicable for traffic where it is possible to dynamically adjust the data generation rate—we call it *elastic traffic*. Most perceptual data, such as audio and video streams generally belongs to this traffic class. The resulting scheme is similar in spirit to the TCP-friendly approaches. However, there is a fundamental difference in how it is done. The network or system layers remain as simple as possible. The responsibility of the network layer is simply to pass on only selected end-point events to the application. Since, the solutions are now implemented at application level; therefore these can be made much more sophisticated without and significant increase in network layer complexity.

In this chapter we first give some background information on congestion control mechanisms in TCP and then we discuss the *InTraN* meta-engineering of TCP that will yield iTCP. In the next chapter we discuss implementation details and present experimental performance results of the controlled iTCP/video symbiosis.

```
initially, cwnd = 1 (one segment);
ssthresh = 65535 bytes;
win_size = min (cwnd, snd_wnd);
When congestion occurs, do:
        ssthresh = max(win_size/2, 2);
        if congestion was due to timeout
                cwnd = 1;
        for every ACK received:
                if (cwnd <= ssthresh)
                   cwnd =  2 * cwnd;
                else
           cwnd = cwnd + segment_size;
```

Figure 8. Slow Start/Congestion Avoidance mechanism (SSCA)

```
When a 3rd duplicate ACK is received:
  ssthresh = max(2, min(cwnd, snd_wnd)/2);
  Retransmit missing segment;
  cwnd = ssthresh + 3;

Each time another duplicate ACK arrives, do:
  cwnd = cwnd + 1;
  transmit a new segment;

When a new ACK arrives, do:
  cwnd = ssthresh;
```

Figure 9. Fast Retransmit/Fast Recovery mechanism (FRFR)

4.2 Congestion Control in TCP

TCP is a connection-oriented unicast protocol that offers reliable data transfer as well as flow and congestion control. TCP maintains a congestion window that controls the number of outstanding unacknowledged data packets in the network. Sending data consumes slots in the window of the sender and the sender can send packets only as long as free slots are available. When an acknowledgment (ACK) for outstanding packets is received, the window is shifted so that the acknowledged packets leave the window and the same number of free slots becomes available.

4.2.1 Congestion Control Algorithms

On startup, TCP performs slow-start, during which the rate roughly doubles each roundtrip time to quickly gain its fair share of bandwidth. In steady state, TCP uses an additive increase, multiplicative decrease mechanism AIMD) to detect additional bandwidth and to react to congestion. When there is no indication of loss, TCP increases the congestion window by one slot per roundtrip time. In case of packet loss indicated by a timeout, the congestion window is reduced to one slot and TCP reenters the slow-start phase. Packet loss indicated by receiving three duplicate ACKs results in a window reduction to half its previous size. Therefore, the two principal mechanisms that TCP uses to detect network congestion are (a) when the retransmission timer times out and (b) when three ACKs arrive. Two algorithms then contribute to TCP's congestion control behavior; these are the classic algorithm of slow start/congestion avoidance [Jac88], and the augmentation of fast retransmit/fast recovery [Jac90]. The two algorithms are outlined in Figure 8 and Figure 9 respectively.

Table 7. TCP Congestion Control Internal Events

Event	Meaning	Description	SSCA	FRFR	Sub
1	Retransmission timer timed out	Possibly congested network or the segment was lost	X		X
2	A new ACK was received	Increment snd_cwnd either exponentially (if less than sstheresh) or linearly otherwise	X		
3	snd_cwnd has reached the slow start threshold ssthresh	Switch incrementing snd_cwnd from exponential to linear	X		
4	A third duplicate ACK was received	A segment was probably lost, perform fast retransmit		X	X
5	A fourth (or more) duplicate ACK was received	One segment has left the network; we can transmit a new segment		X	
6	A new ACK was received	Retransmitted segment has arrived at the destination and all out of order segments buffered at the receiver are ACKed		X	X

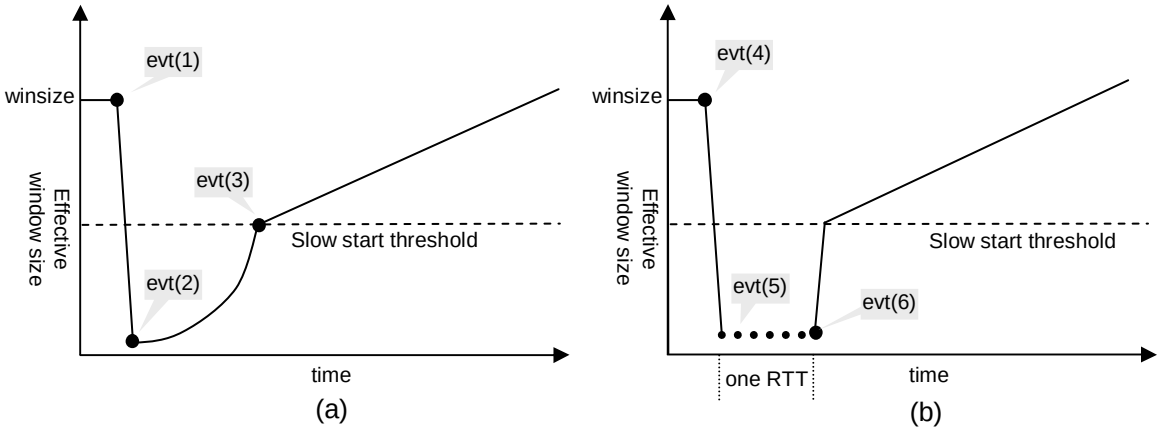

Figure 10. Changes on TCP's sending window due to congestion control events

4.2.2 Congestion Control Events

Table 7 lists six events that internally occur when the TCP invokes congestion control algorithms. Although many other TCP events might occur during a TCP session (e.g., flow control events or connection establishment and termination events), we are only interested in congestion control events.
In Table 7, the column labeled (SSCA) refers to events that take place in the Slow Start/Congestion Avoidance algorithm, and the label (FRFR) refers to events that take place in the Fast Retransmit/Fast Recovery algorithm. These events are also presented in Figure 10. Plot (a) of the figure shows the sequence of events of the SSCA algorithm and their affect on effective window size, and plot (b) shows the same sequence for the FRFR algorithm. However, in general design we expect only a subset of the internal events of the protocol to be of interest to the subscriber application. Only a subset of these is made accessible via the interface. An application instance typically subscribes even to a subset of the accessible events. The column (Sub) shows subscribable events in our design.

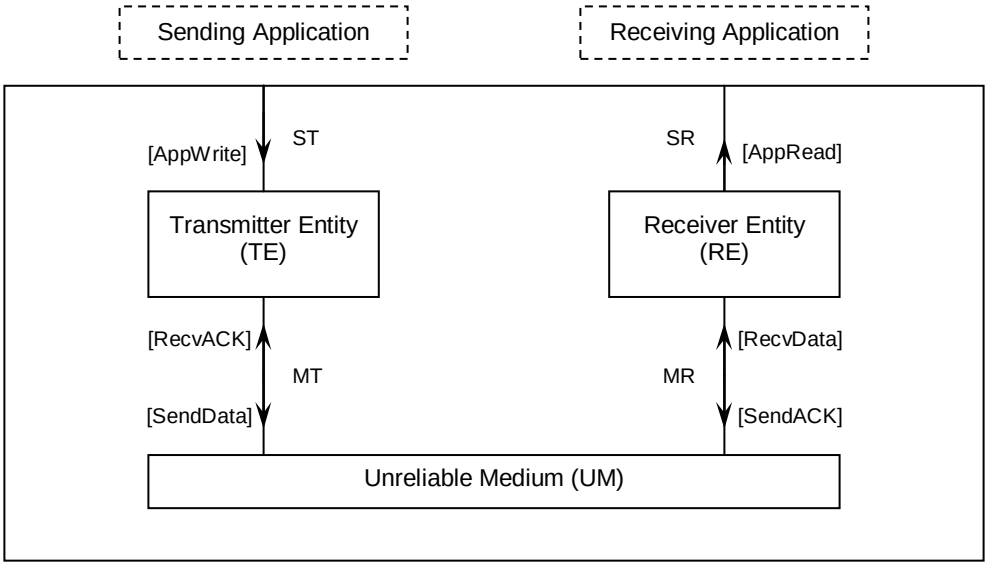

Figure 11. Simple TCP system composition

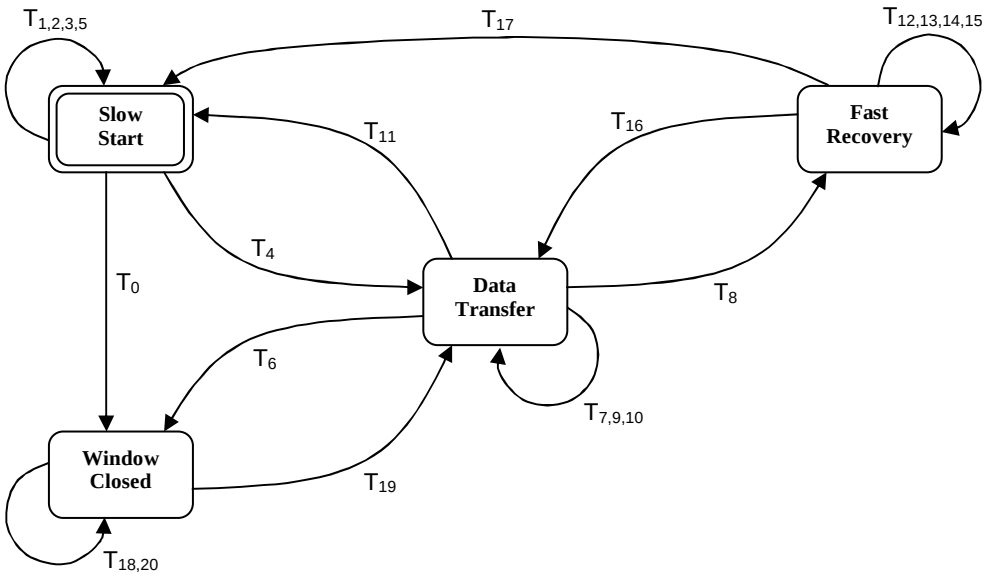

Figure 12. EFSM of TCP

4.3 TCP Meta-engineering

Now we show how to perform the meta-engineering extension on TCP and make it *InTraN*-enabled—we call the extended protocol iTCP.

4.3.1 The SDL Model

First, we formally describe the abstract protocol using SDL, and then we augment the protocol by adding *InTraN* components. [Tur93] described a simple sliding window protocol in SDL that featured positive acknowledgments and retransmission mechanisms. We transformed this protocol into simplified TCP by adding congestion control support. The simplified TCP can be modeled as a composition of three blocks, *Transmitter Entity* (TE), *Receiver Entity* (RE), and *Medium*. The *Medium* represents the underlying unreliable service (e.g., IP and lower layers) while TE and RE represent the two endpoints of a TCP connection. Figure 11 describes the composition. The sending and receiving applications are located in the environment. They interact with the system via two service access points modeled by two unidirectional channels, *ST* (from the environment to the TE) and *SR* (from RE to the environment). The channel *ST* carries the *AppWrite* signal from the sending application to the TE, and the channel *SR* carries the *AppRead* signal from the RE to the receiving application. The TE uses a bidirectional channel *MT* to send data (via a *SendData* signal) and to receive acknowledgments (via a *RecvACK* signal) over the *Medium*. One the opposite side, the RE also uses a bidirectional channel *MR* to receive data (via a *RecvData* signal) and to send acknowledgments (via a *SendACK* signal) through the *Medium*.

In Figure 14 (a 4-pages figure at the end of the chapter) we formally present in SDL notation the fundamental part of TCP's congestion control and flow control mechanisms at the sender (*Transmitter Entity*). The system describes a unidirectional data service. In this abstract description, we only focus on the sliding window and congestion control aspects of TCP, many of the details in conventional TCP are hidden, such as: buffer size issues, sequence number calculations (e.g., sequence number wrap around), and checksum tests. Furthermore, many of the irrelevant details are hidden inside procedure calls, e.g., CalcRTO().

The EFSM of this system is depicted in Figure 12 and can be described as:

- *S = {Slow Start, Data Transfer, Fast Recovery, Closed Window},*
- s_0 *= Slow Start,*
- *E = {AppWrite, RecvACK, rexmt timeout},*
- *O = {SendData},*
- *V = {seqno, ackno, RAW, dACK, pACK, FRFlag, RTO, rexmt, Cwnd, Swnd, LU, LS, ExpBoff}.*
- *f = {*T_0*,* T_1*, …,* T_{20}*},* The transitions of *f* are labeled in Figure 14.

4.3.2 EFSM of iTCP

We want iTCP to track two events: '*retransmission timer timeout*' and '*receiving third duplicate ACK*'. Both events signify packet loss and usually cause TCP to trigger congestion control procedures. Therefore, the augmented EFSM of our *Transmitter* protocol becomes: (*InTraN* additions are shown in bold)

- *S = {Slow Start, Data Transfer, Fast Recovery, Closed Window},*
- s_0 *= Slow Start,*
- *E = {AppWrite, RecvACK, RexmtTimeout,* ***GetVal***, ***SetVal***, ***SetFlag****},*
- *O = {SendData,* ***ExpVal***, ***Event****},*
- *V = {seqno, ackno, RAW, dACK, pACK, FRFlag, RTO, rexmt, Cwnd, Swnd, LU, LS, ExpBoff,* ***RA***, ***RT****}.*
- *f* is augmented as we described in Figure 6 (i.e., by adding three transitions for the *GetVal, SetVal, and SetFlag* events, and modifying existing transitions of subscribable events in every state).

RA and *RT* are the Boolean subscription flags associated with events *RecvACK* and *RexmtTimeout* respectively. We chose the sets E'_P, *B*, and *C* as follows:

- E'_P = {*RecvACK, RexmtTimeout*},
- *B* = {*dACK, Swnd, RAW*},
- *C* = {}.

The *InTraN*-added members of *E* (i.e., *GetVal*, *SetVal*, and *SetFlag*) are for internal *SM* use only. Therefore, they are not included in E'_P (i.e., they cannot be subscribed by a *SP*). The same applies to the subscription flags (*RA*, *RT*) which cannot be included in the set *B* or *C*. In Figure 15 (at the end of this

chapter) we show the *InTraN*-enabled SDL version of the (slow start) state only. The remaining states can be extended by adding exactly the same components.

4.4 A Complete TCP EFSM/SDL Model

In this section we provide an EFSM model for the original TCP standard that was proposed in RFC 793 [Pos81]. We have augmented the original standard to include the congestion control mechanism of TCP Reno described above. We have posted the complete SDL description of this EFSM in a technical report [Zag05] which is posted on our web server. We have developed this model as a supplement material for the *InTraN* paradigm. Using this model, any *InTraN*-enabled protocol extension solution can be formulated as we described earlier by selecting the sets E'_P, *B*, and *C* and by writing a set of *TM*s that implement the proposed extension. We felt that this model can be beneficial for other researchers who might be interested in the formal description of the TCP standard using the EFSM/SDL notation.

4.4.1 Remarks and Simplifying Assumptions:

1- The EFSM always remembers the current state in the variable (CurrState) and the previous state in the variable (PrevState),
2- The TCP endpoint has unlimited buffer space (e.g., buffer space to queue SENDs and RECEIVEs is always available)
3- In any state, whenever a segment is sent, the segment is added to the Retransmission Queue (RexmtQueue) and the retransmission timer (REXMT) is started.
4- The (REXMT TIMEOUT) event has been modeled in all states except (FIN-WAIT-2, TIME-WAIT, CLOSED), since in these states the endpoint have already received an ACK of its FIN segment (i.e., will not transmit any segments afterwards).
5- The (TIMEWAIT TIMEOUT) event has been modeled in (TIME-WAIT) state only. In all other states, this timer is irrelevant.
6- The following were not modeled from RFC 793:
   a) Security/Compartment and Precedence processing.
   b) The STATUS user call.
   c) The PUSH mechanism (i.e., PSH control bit)
   d) The URGENT mechanism (i.e., URG control bit)

4.4.2 The Complete TCP EFSM

The TCP EFSM=(S, s_0, E, f, O, V) can be described as follows:

1. States (S) = {CLOSED, LISTEN, SYN-SENT, SYN-RCVD, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSING, CLOSE-WAIT, LAST-ACK, TIME-WAIT}.

2. Initial State (s_0) = {CLOSED}

3. Events (E)

User Calls (subscriber events) = {Active OPEN, Passive OPEN, SEND, RECEIVE, CLOSE, ABORT}.

Arriving Segments (service events) = {SEGMENT ARRIVE (SYN, ACK, RST, FIN)}.

Timeouts (internal events) = {
REXMT TIMEOUT: The Retransmission Timer (REXMT) has timed out,
, TIME-WAIT TIMEOUT, USER-TIME TIMEOUT}.

4. Transition Function (f) = {described in [Zag05]}

5. Output Signals (O) = {Return (message), Return Error (error message), Signal User (message), and Segment (SEG)}.

6. Variables (V)

A. Segment Variables

SEG.SEQ: segment sequence number
SEG.ACK: segment acknowledgment number
SEG.LEN: segment length
SEG.WND: segment window (Receiver Advertised Window)
SEG.CTL: control bits (ACK, RST, SYN, FIN)

B. Send Sequence Variables

SND.UNA: send unacknowledged
SND.NXT: send next
SND.WND: send window
ISS: initial send sequence number

C. Receive Sequence Variables

RCV.NXT: receive next
RCV.WND: receive window
IRS: initial receive sequence number

D. Timers

REXMT: Retransmission Timer.
TIMEWAIT: Time-wait Timer
USERTIME: User Timer

E. Counters

dACK: duplicate ACK counter
ExpBoff: exponential backoff counter

F. Other

CurrState: Current State
PrevState: Previous State
RTO: Retransmission Timer Out value
RTT: Round Trip Time—used to calculate RTO
SRTT: Smoothed RTT—used to calculate RTO
CWND: Congestion window
MSS: Maximum Segment Size
SSthresh: Slow Start Threshold
MSL: Maximum Segment Lifetime

G. Buffers

Send Buffer: Send Buffer
RCV Buffer: Receive Buffer
OO RCV Buffer: Out of Order Receive Buffer
Rexmt Queue: Holds sent but unacknowledged segments
User Calls Queue: Holds outstanding user calls (e.g., SEND, RECEIVE, CLOSE)

4.5 Classification of EFSM Components

Figure 13 classifies the main components of a generic communication protocol EFSM and connects those to the EFSM components of TCP. The upper part of the figure (shown in yellow) presents the generic classification with three main components: Events, States, and Variables. The lower part of the figure (shown in green) classifies the TCP components.

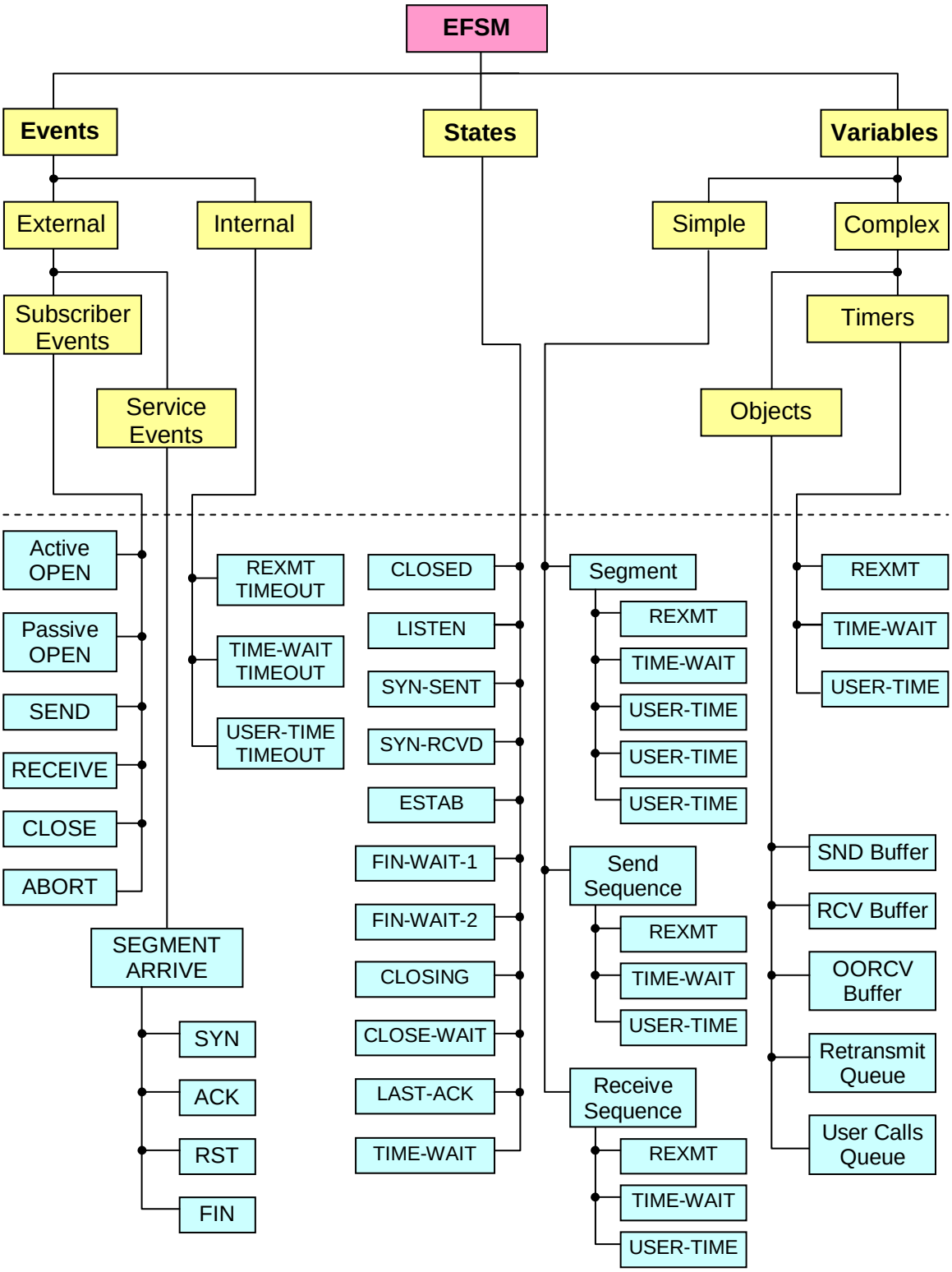

Figure 13. Classification of the EFSM components and their TCP counterparts

4.5.1 Events

Events can be External (i.e., they are triggered by receiving a signal from an external entity), or Internal (i.e., they are triggered when a timer times out). Signals that trigger external events can come from two types of entities: *Service Providers* (i.e., lower-level protocols that provide communication services to this EFSM), or *Service Subscribers* (i.e., upper-level protocols that uses the communication services offered by this EFSM). A signal received from a Service Provider triggers a Service Event, and a signal received from a Service Subscriber triggers a Subscriber Event. In the TCP part, the Service Provider is IP and the Service Subscriber is the user, therefore, user calls like (OPEN, SEND, RECEIVE, etc) are classified as Subscriber Events, and receiving a segment from IP (SEGMENT ARRIVE) is classified as a Service Event. Whenever one of the three timers in TCP expires, it generates an Internal Event. Internal Events in TCP happen whenever one of the timers expires.

4.5.2 States

States can be classified in a hierarchy, where the top level contains the states in this EFSM. Each state on the top level can itself contain a smaller EFSM whose states can be considered as second level states. For example, the four states of the congestion control EFSM presented in the previous section (i.e., Slow Start, Data Transfer, Window Closed, and Fast Recovery) are all considered to be part of the (ESTABLISHED) state of this EFSM, and therefore they can be classified as second level states. In Figure 13 we only show the 11 states of the complete TCP EFSM at the top level.

4.5.3 Variables

We have classified Variables into two categories: Simple and Complex. Simple variables have simple data types like integers or character strings. Complex variables are class objects defined with *methods* and *values*. Timers are special type of complex variables since (i) they have built in methods in SDL (e.g., SET, RESET) and (ii) they trigger internal events when they expire. In the TCP part we classify simple variables into three parts: Segment, Send Sequence, and Receive Sequence (the EFSM has additional variables but we did not include them in the figure due to space limitation). We also show the three timers (REXMT, TIME-WAIT, and USER-TIME) and all the Buffers/Queues as Complex variables.

4.6 Conclusion

In this chapter we have first reviewed some of the issues concerning congestion control in TCP and the need for application involvement in designing adaptive TCP-friendly solutions. Then, we have shown a real application of the *InTraN* meta-engineering on TCP which gave an *InTraN*-enabled version of TCP (or iTCP). In the next chapter we demonstrate a TCP-friendly, congestion management scheme based on iTCP and the *InTraN* Transientware mechanisms.

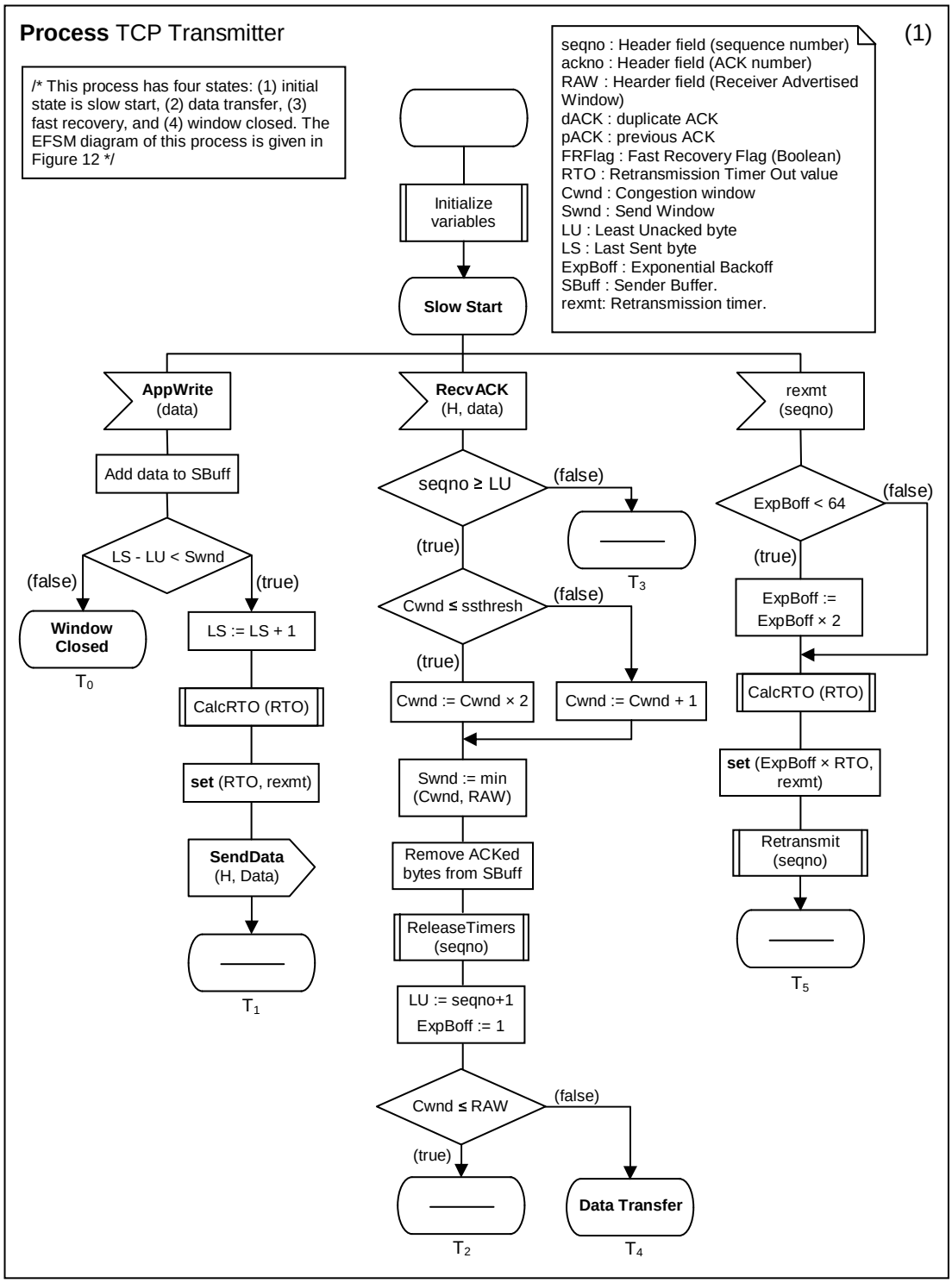

Figure 14. SDL description of a simplified TCP transmitter

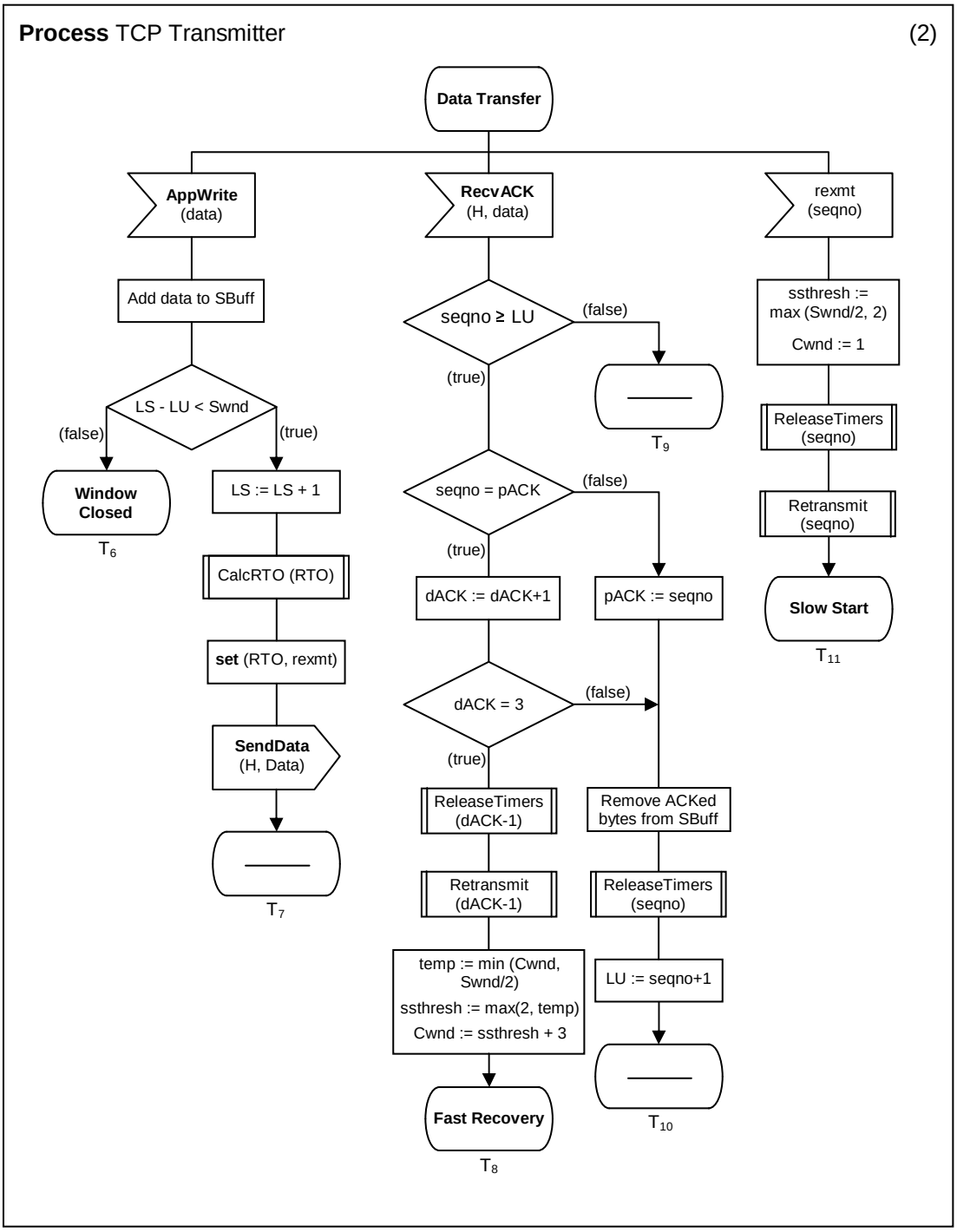

Figure 14 (continued)

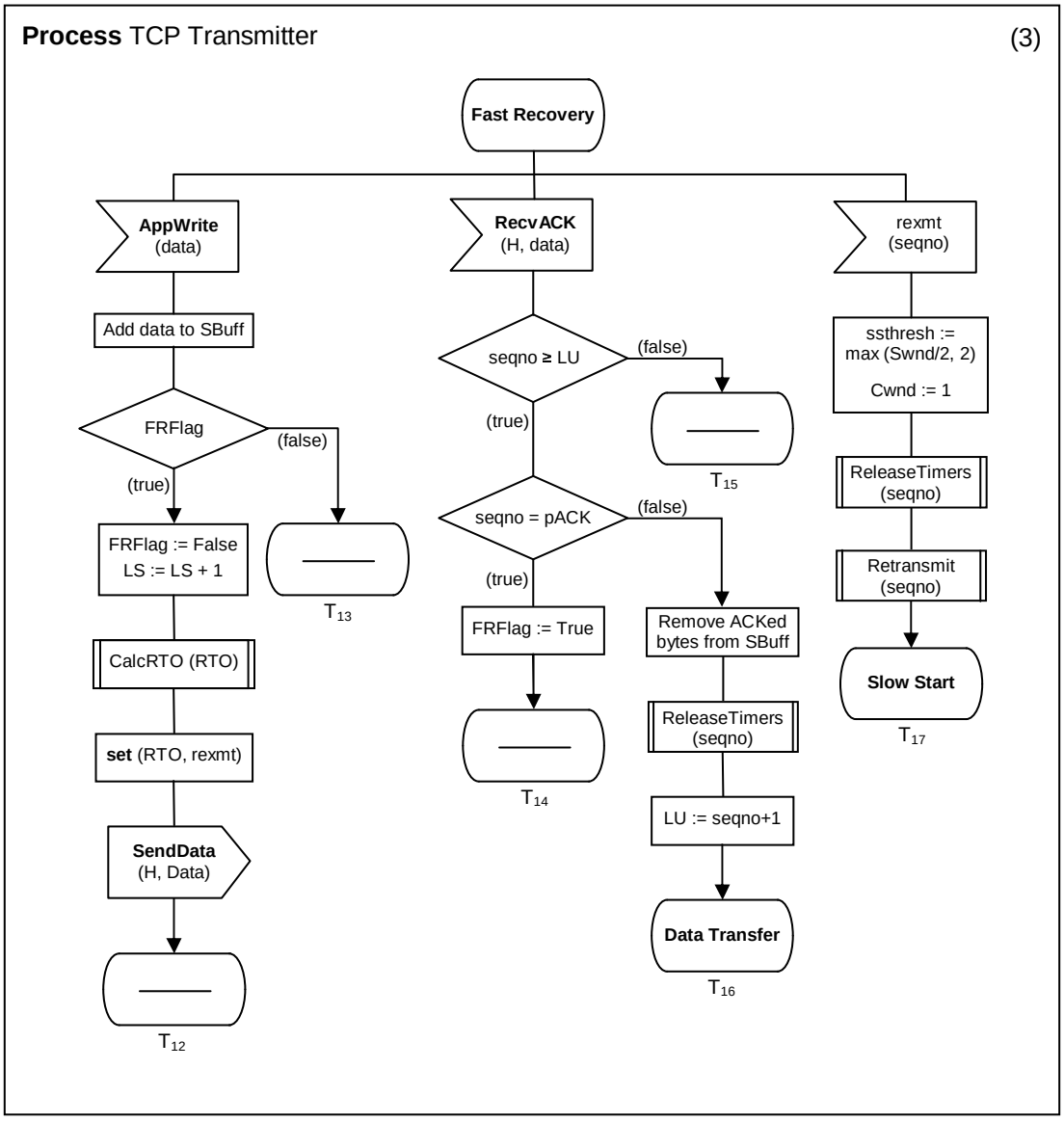

Figure 14 (continued)

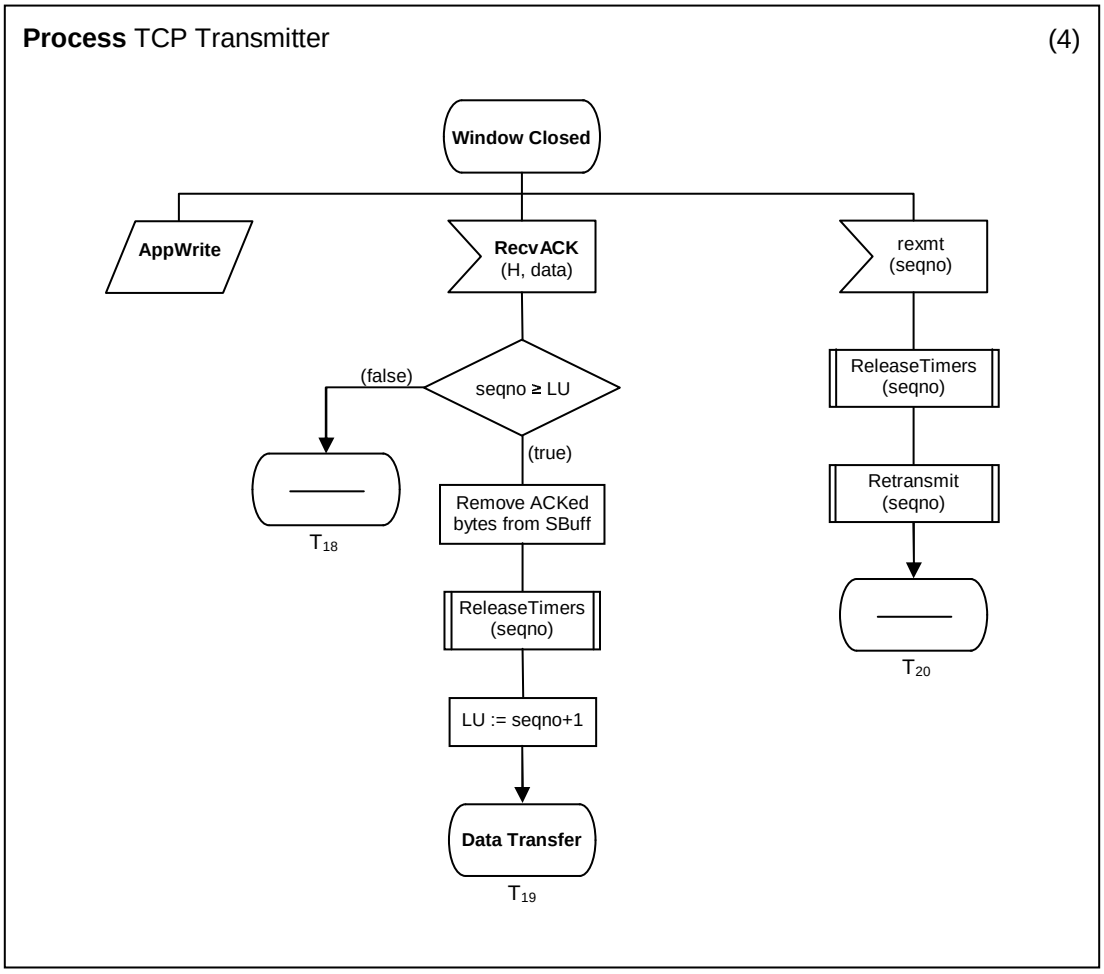

Figure 14 (continued)

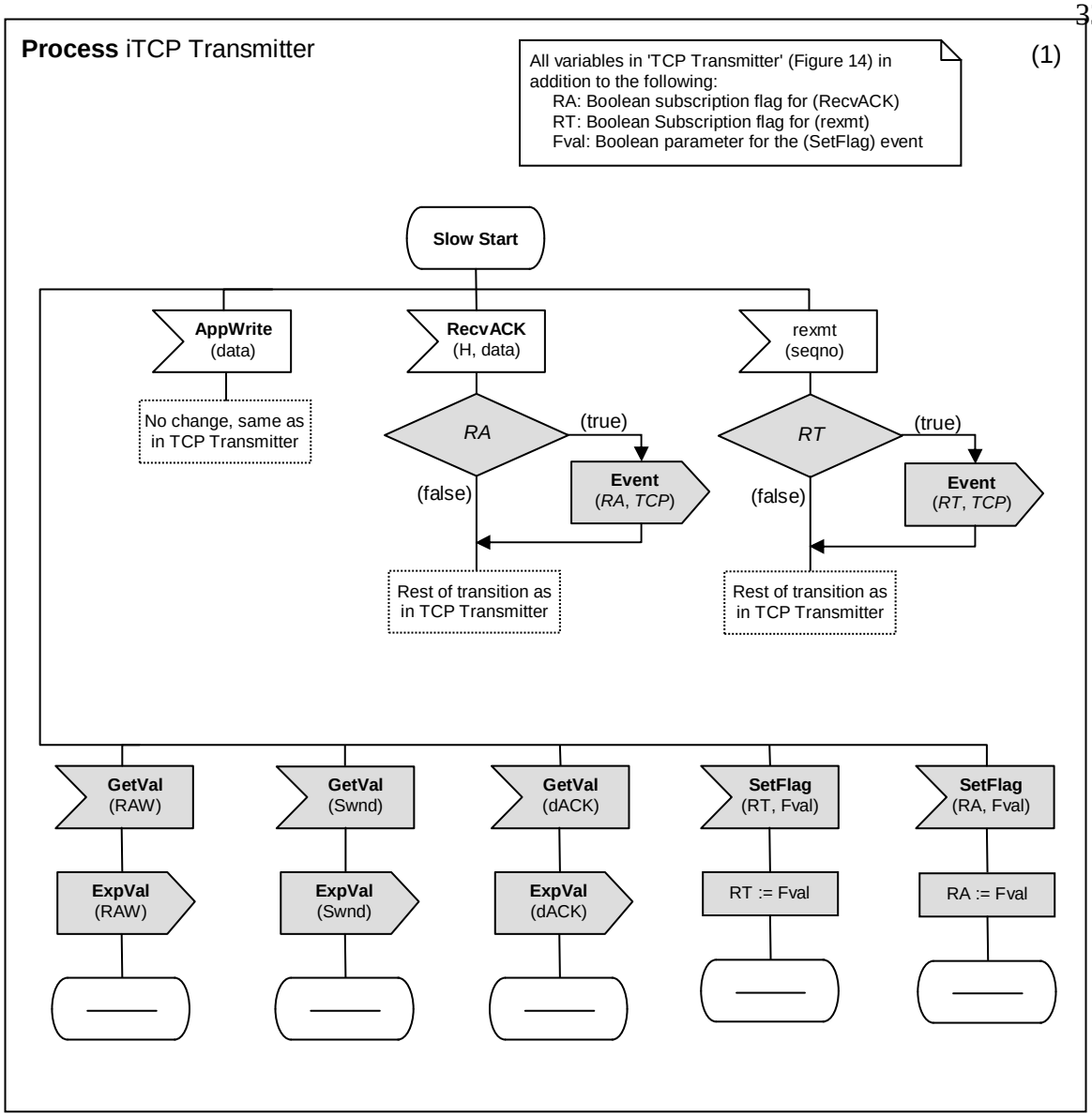

Figure 15. iTCP's (Slow Start) state extended with *InTraN* components

CHAPTER 5

iTCP part II: Implementation and Performance

5.1 Implementation Details

5.1.1 System Architecture

Figure 16 depicts the conceptual architecture of the system on FreeBSD. The scheme works in three spaces: user space, system space, and kernel space. Once it establishes a TCP connection, the user process starts by binding the TCP kernel with a set of chosen events from Table 7 using a *Subscription API* that extends the

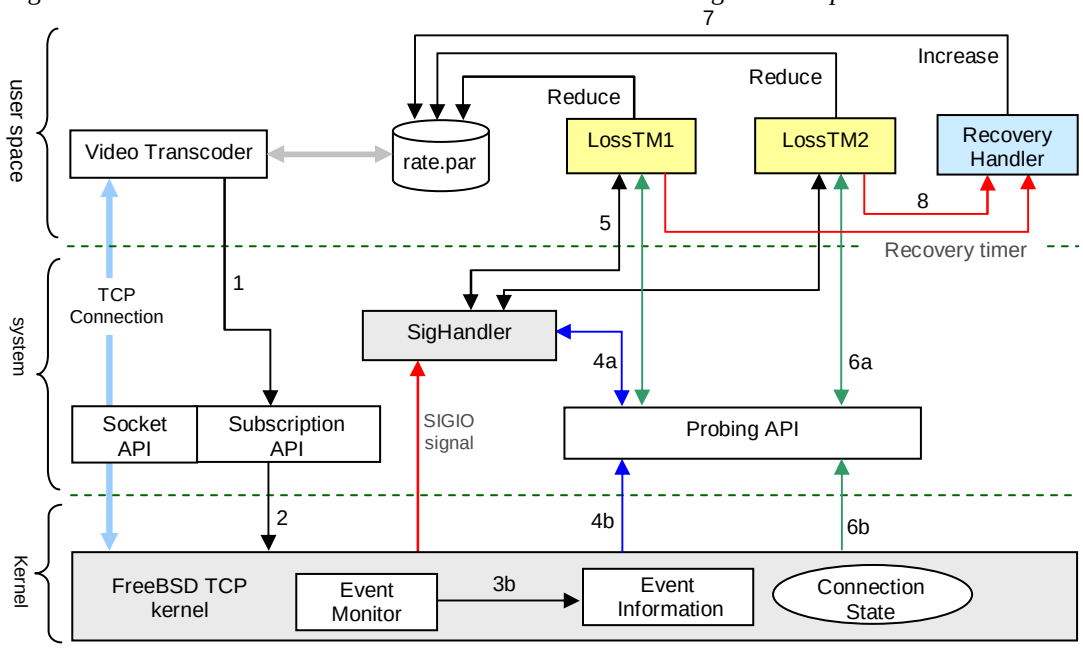

Figure 16. The TCP-interactive extension and API

standard socket API (1). An entity called *Event-Monitor* runs in the TCP kernel space and monitors all subscribed events for every socket (2). Assuming at some point event (`evt`) occurs in socket (`sock`). The Event-Monitor sends a (`SIGIO`) signal that is caught by the (`Signal Handler`) (3a), and at the same time writes the socket descriptor of the socket (`sock`) in the process structure `proc{}` of every process that is currently subscribed with this socket. Also, it marks all subscriptions of event (`evt`) in the socket (`sock`) as outstanding (3b)—i.e., waiting to be handled. The OS activates the signal handler (`SigHandler`) associated with the (`SIGIO`) signal whenever this signal is caught. The (`SigHandler`) first uses the *probing API* to retrieve the socket id (`sock`) of the socket that generated the event. Then, it uses the probing API again to access the socket (`sock`) and get relevant information about the outstanding subscription of event (`evt`). The information retrieved includes the event type and the name of the *TM* bound to it (4a,b). Immediately after that, the (`sighandler`) invokes this *TM* (5). *TM*s are usually small programs supplied by the user or by a third-party as ready to run executables custom-designed to handle certain events. One *TM* is forked by the (`sighandler`) for each valid (`SIGIO`) signal. The probing API allows the *TM* to probe additional information about the state of the TCP connection (6a, b). We show two *TM*s in Figure 16, `LossTM1` and `LossTM2` to handle the two loss events mentioned above. These *TM*s

Table 8. The API Extension of iTCP

Level	Caller	Description
`void GetEvents (int *NumOfEvents, evtInfo *EventList[]);`		
User	User process	Retrieve the complete list of available events in the TCP kernel. Retrieve **`evtInfo{}`** struct for each event in the list.
`int SubscribeEvt (int sock, int evt, int T-ware);`		
User	User process	Subscribe with the socket **`sock`** for event type **`evt`**. Register handler **`T-ware`** for this event. This call will add a **`subInstance{}`** structure to the **`evtList`** list in the subscribed socket.
`int UnsubscribeEvt (int sock, int evt);`		
User	User Process	Unsubscribe a previously subscribed event. Afterwards, no signal will be sent when this event occurs. Remove the **`subInstance{}`** from the **`evtList`** list in the subscribed socket.
`int GetSockid (void);`		
System	Signal Handler	Get the descriptor of the socket that sent the signal when the subscribed event had occurred. This is necessary since a process can subscribe to many sockets, and the *Signal Handler* needs to know which socket triggered the event.
`int ProbeEvtInfo (int sock, struct evtInfo *info);`		
System	Signal Handler	Get the *number* and the *Handler* name of the event that has just occurred in the socket **`sock`**.
`int ProbeSocket (int sock, struct connState *conn);`		
User	Event Handler	Probe the socket **`sock`** to retrieve the *current state* of the TCP connection is the **`connState{}`** structure.
`int GetSubPerm (int sock, int evt, int *perm);`		
User	User Process	Get the current access permission string **`perm`** for the event **`evt`** subscribed with socket **`sock`**. Get four flags: (*Read*, *Write*, *Subscribe* and *Trigger*) for two levels (*System* and *User*).
`int GetSubPriority (int sock, int evt, int *priority);`		
User	User Process	Get the *Priority Level* of the event **`evt`** subscribed with socket **`sock`**. Returned **`Priority`** is between 1 and 3.
`int GetHandlerPerm (int sock, int evt, int *mask);`		
System	Root Process	Get the *Connection Access Mask* **`mask`** for the event **`evt`** subscribed with socket **`sock`**. The returned value in **`mask`** specifies which fields in the **`connState{}`** struct are accessible by the *T-ware* and which fields are not.
`int SetSubPerm (int sock, int evt, int perm);`		
System	Root Process	Set a new access permission string **`perm`** for the event **`evt`** in the socket **`sock`**. The integer **`perm`** should specify four flags: (*Read*, *Write*, *Subscribe* and *Trigger*) for two levels (*System* and *User*).
`int SetSubPriority (int evt, int priority);`		
System	Root Process	Set a new *Priority Level* for the event **`evt`** in the socket **`sock`** by assigning a value to **`priority`**.
`int SetHandlerPerm (int sock, int evt, int e_hand, int mask);`		
System	Root Process	Set a new access mask **`mask`** for the event **`evt`** in the socket **`sock`**. The integer **`mask`** should specify which fields in the **`connState{}`** structure are accessible and which fields are not.
`int GetEvtState (int evt, int *state);`		
User	User Process	Get the *Subscription State* of event **`evt`**. Return *zero* in **`state`** if the event is subscribable or *one* otherwise.
`int DelEvent (int evt);`		
System	Root process	Set the *deleted* flag in the **`evtInfo{}`** structure to *true*. Afterwards, **`evt`** will be ignored by subsequent system calls.
`int AddEvent (int evt);`		
System	Root Process	Reset the *deleted* flag in the **`evtInfo{}`** structure to *false*. Afterwards, **`evt`** will be reported by subsequent calls.
`int UntriggerEvt (int sock, int evt, int status);`		
User	User process	Trigger/untrigger subscribed event **`evt`**.

employ a symbiosis throttling mechanism based on the TCP state to calculate an optimized reduced bit rate (h_{best}) to put the transcoder in a frugal state. They also calculate an optimal duration ($T_{recovery}$) for the frugal

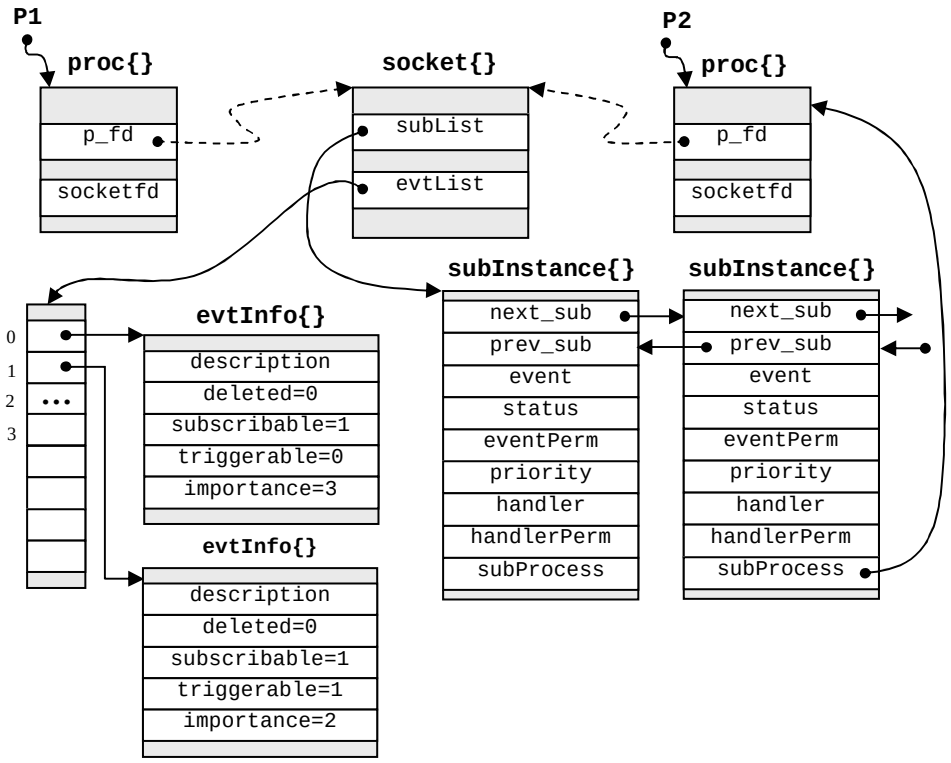

Figure 17. iTCP internal data structures

```
struct evtInfo *evtList[]
int n;
s = makeSocket();
SubscribeStub{
   GetEvents(&n, evtList);
   For(i=0; i<n; i++){
      if (evtList[i]->subscribable == 1)
        if ((i == REXMT_TOUT) ||
            (i == THIRD_DACK))
         SubscribeEvt(s, i, EvtHandler[i]);
   }
   iSockets = iSockets ∪ s;
}
```

```
struct evtSubInfo *ESinfo;
Probe{
   s = GetSockid();
   if (s ∈ iSockets){
        ProbeEvtInfo(s, ESinfo);
        if (ESinfo->evt in (REXMT_TOUT,THIRD_DACK)){
                Switch (ESinfo->Tware){
                   case 1: Tware1();
                   case 2: Tware2();
                   ...
                }
        }
   }
}
```

Figure 18. Subscription and probing scenarios

state—after which the transcoder returns to its normal rate. They convey this rate reduction to the transcoder by writing the new rate to the file named "`rate.par`" (7) and they start a timer that will expire when $T_{recovery}$ time has passed after which a recovery handler is invoked to write the normal rate into "`rate.par`" (8).

Table 9. Implementation details `evtInfo{}` and `subInstant{}`

(a) `struct evtInfo{}`

Field type	Field Name	Description
`char *`	`Description`	A brief description text about the event and its meaning.
`int`	`Deleted`	A flag to mark the event as deleted.
`int`	`Subscribable`	A flag to decide if the event is subscribable.
`int`	`triggerable`	A flag to decide if the event can be triggered by the subscribing process.
`int`	`Importance`	Importance level of the event.

(b) `struct subInstance{}`

Field type	Field Name	Description
`Struct subInstance *`	`next_sub`	A pointer to the next entry in the linked list
`Struct subInstance *`	`prev_sub`	A pointer to the previous entry in the linked list
`int`	`event`	Event number/name.
`int`	`status`	The status of the signal.
`int`	`eventPerm`	Access permission string for this subscription instance.
`int`	`Priority`	Priority of the event in this subscription instance.
`int`	`handler`	The number of event handler for this subscription instance.
`int`	`handlerPerm`	Access pattern mask for the connection state variables.
`struct proc *`	`subProcess`	A pointer to the subscribing process.

5.1.2 API

Table 8 shows the complete API system designed. In this table for each system call we list its prototype, the level of its caller (user or system), its potential caller (application, signal handler, or *TM*), and a brief description about its functionality. Some of these functions are designed for the network administrator (root process) to manage event subscription by granting priority levels and access permissions for the user process.

5.1.3 Internal Data Structures

We have implemented the scheme on FreeBSD 4.5 kernel. Here, we discuss some of the internal details of iTCP implementation. A user process can open one or more TCP sockets. At the same time a socket can be used by more than one process. Figure 17 shows the relevant data structures needed to implement the subscription and probing scenarios in iTCP. An open socket maintains a list of events called (`evtList`) as an inventory of all events supported by iTCP. The socket uses the (`evtList`) field to retrieve the static information related to any event type. The list is implemented as an array of pointers to a structure called `evtInfo{}`. The structure `evtInfo{}` shown in Table 9 (a) represents one event type and stores information about the event such as its description and relevant attributes. The socket also maintains a doubly linked list of subscribed events for every subscriber process called (`subList`).

Whenever a user process subscribes with a new event, the socket adds a new entry to this list called `subInstance{}`. The structure `subInstance{}` represents one subscription instance for a given process/socket pair. It contains information such as event number, status and name of the *TM* bound to the subscribed event. Table 9 (b) shows the complete `subInstance{}` structure. The socket removes a `subInstance{}` entry from the (`subList`) if a user process decides to unsubscribe from a previously subscribed event.

5.1.4 Subscription and Probing Scenarios

Figure 18 demonstrates subscription and probing scenarios. We explain an application stub routine `SubscribeStub()` which handles this stage. After creating a socket (`s`), the `SubscribeStub()` routine uses the `GetEvents()` system call to retrieve the set of events available from the socket in `evtList[]` and their number (`n`) from the kernel. Let's assume that the application wants to subscribe to two events: The *retransmission timer time-out* event (REXMT_TOUT) and the *third duplicate ACK* event (THIRD_DACK). Here, we let the index (`i`) represent the event number in the list `evtList[]`. `SubscribeStub()` first checks if the current event (`i`) is either (REXMT_TOUT) or (THIRD_DACK), if this is true; it makes a `SubscribeEvt()` system call to subscribe to the event. After finishing the loop this system call adds the socket (`s`) to the set (`iSockets`), which includes all sockets that the application had subscribed with. When the kernel sends a `SIGIO` signal, a system routine catches it. This routine then uses the probing function `Probe{}` (shown in Figure 18) to handle the signal. The `Probe{}` routine calls `GetSockid()` to find out which socket has sent the event, and stores its descriptor in (`s`). If the socket (`s`) was among the set of subscribed sockets (`iSockets`) of this application, it calls `ProbeEvtInfo()` to retrieve the subscription information for this subscription instance. Internally, when the `ProbeEvtInfo()` system call is made, the kernel traces the `subList[]` of socket (`s`) and looks for a subscription instance `subInstance{}` whose status field equals 1, i.e., this is an outstanding instance waiting for the signal handler attention. Normally there should be only one outstanding instance per application in the socket's `subList[]`. Once found, the kernel returns two fields from the outstanding instance to the application in the `evtSubInfo{}` structure: the event number (`evt`), and the TM name (`Tware`). When the `ProbeEvtInfo()` returns, the `Probe()` checks if the event (`evt`) is iTCP related, i.e., it is either (REXMT_TOUT) or (THIRD_DACK), and then it executes the proper *TM* as dictated by the value returned in `ESinfo->Tware`.

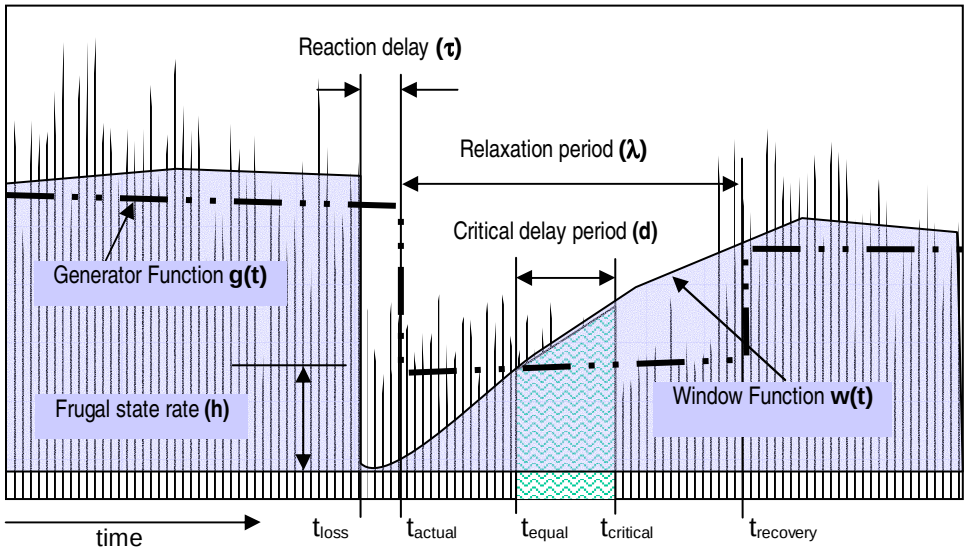

Figure 19. Symbiosis throttling model

5.2 Symbiosis Throttling Model

The key to the system is the intermediate event gluing mechanism—or as we call it *symbiosis throttling*. It performs the key task of dynamically specifying the target rate for the application based on the event notification interrupt. The idea is to accept the event feedback provided by the underlying interactive transport layer, and generate a corresponding rate feedback for rate formation capable applications. This feedback is estimated in a way that ensures transport service with applications specified delay conformation over the otherwise classic transport service.

The main idea is that when a time-out event ($\xi = 1$) occurs in the transport, we let the subscriber rate retract to a smaller rate. We call this retraction state as *frugal state*. The key issue is how to optimally design the frugal state's retraction point so that the overall system meets the delay bound of the application.

5.2.1 Analysis of Symbiotic Throttling

Let *g(t)* be the generation function denoting the data rate at which the rate formation capable application produces data as a function of time. Let *w(t)* is the bandwidth function provided by the transport channel over which, the application sends the data. Figure 19 explains the model. During normal operation $w(t) \geq g(t)$. When a loss event is detected (e.g., timeout) the transport bandwidth retracts to some smaller effective value due to window resizing. The underlying cause might be a packet loss or a congestive delay deep inside network. In either case, the sender transport buffer builds up and results in increased communication delay. In response to the loss event, we let the subscriber adjust its generation rate to a lower generation state (we call this state the *frugal bandwidth* state). The normal operation is however by a *satisfied bandwidth* state. In any practical feedback system there is also always a reaction delay in the feedback loop. Let τ be the reaction time needed by the subscriber process to react and adjust its rate. Given the above model, the particular design problem we address is the following:

> *Given the bandwidth function w(t), the generation function g(t), the satisfied state bandwidth (B_{sat}), and the upper bound on the acceptable data delivery delay (d_Q), determine the best possible frugal state (generation rate and its duration) for which the bound d_Q can be ensured.*

Here the delay bound d_Q is the maximum delivery delay an application can sustain between generation endpoint and delivery endpoint of the application layer. We now further define two additional concepts important for the derivation to be presented.

5.2.2 Critical-delay-point inequality

Assuming the loss is detected at time t_{loss}. After the loss assume it takes t_{equal} time for the transport system to again equalize the transport bandwidth with the frugal state generation rate of the subscriber. This is the point where $w(t) = g(t)$. We call this point the *even-point*. Since the generation rate is larger than the transport rate before the *even-point* is reached, therefore the transport buffer will build up until the even-point is reached. The buildup will gradually decrease after the even-point. Thus the bytes entering the buffer exactly at the even-point will face maximum delay. Let this time be called critical-delay-point $t_{critical}$. Thus, if the transport buffer already has Q bytes in it (before moving to the frugal state), the buffer size at even-point is given by the LHS of equation—(1a). Let d be the maximum acceptable delay, then the following inequality must hold. We name it *critical-delay-point* constraint:

$$\max(0, Q) + \int_{t_{loss}}^{t_{equal}} g(t) - w(t) \cdot dt \leq \int_{t_{equal}}^{t_{critical}} w(t) \cdot dt$$

Or, $$\max(0, Q) + \int_{t_{loss}}^{t_{equal}} g(t) \cdot dt \leq \int_{t_{loss}}^{t_{equal}+d} w(t) \cdot dt \qquad \text{--(1a)}$$

5.2.3 Recovery-point inequality

The bytes entering the transport buffer after the even-point will face less but non-zero delay. This data too will be entering into the buffer quite full. Additional bytes, those generated between the even-point and the critical-delay-point, will still populate the buffer. Therefore our ultimate goal is to take the buffer into pre-event state before returning to normal generation. Thus, the subscriber system should still continue to operate at somewhat less than satisfied state. This extended frugality will allow remaining buffer buildup to dissipate—completely erasing the effect of the timeout event. We define this time as the *full-recovery-point*. Let's call it the recovery time $t_{recovery}$, then the following second inequality in equation—(1b) must hold. We call it *full recovery-point* constraint.

$$\max(0,Q)+\int_{t_{loss}}^{t_{recovery}} g(t)\cdot dt \le \int_{t_{loss}}^{t_{recovery}} w(t)\cdot dt \qquad \text{--(1b)}$$

5.2.4 Frugal State Determination

The two inequalities respectively can provide a general solution for the level and duration of the frugal state for any general transport bandwidth and generation function. It can also predict the corresponding recovery time.

Below, we solve specifically for the case where the iTCP transport control is similar to TCP (binary-back-off and additive-increase) and a piecewise step $g(t)$. For simplicity, we assume that when a loss event is detected the window function decelerates to zero (i.e., $w(t_{loss})=0$). We first solve for a fast reacting system, where the reaction time is very small and let the buildup before subscriber reaction is Q. Let $g(t)$ is a piece-wise step function. We further assume that the post-fault $w(t)$ is a linear function with bandwidth acceleration m.

Let d_Q is the maximum buffer delay tolerable by the application data. Given a maximum propagation delay limit T_P, and bandwidth $w(t)$, we can say that $d_Q = d+T_P+(1/w(t))$ where d is the total delay faced by the byte entering at critical-delay-point. Since, typically $w(t)>>1$, then d can be approximated by $d = d_Q - T_P$. Let T be the time it takes the system to reach the even-point (i.e., $T = t_{equal} - t_{loss}$). Then critical buffer equality (1a) can be expanded into:

$$Q + mT \le \frac{1}{2}\cdot m \cdot [T + d]^2 \qquad \text{--(2)}$$

$$T^2 - 2Td - d^2 + \frac{2\cdot Q}{m} \le 0$$

It solves to:

$$T = d \pm \sqrt{2d^2 - \frac{2Q}{m}} \qquad \text{--(3)}$$

Only positive real solutions are practical. For any given system arbitrary delay bound cannot be met. In that case both the solutions are imaginary. The model can now be used to determine the limit on the maximum acceptable delay. For the real solution the minimum delay requirement cannot be smaller than:

$$d_{\min} \ge \sqrt{\frac{Q}{m}} \qquad \text{--(4)}$$

T can have two solutions. Both solutions are positive if:

$$d \le \sqrt{\frac{2Q}{m}} \qquad \text{--(5)}$$

Otherwise, only one solution is positive. From T, we can determine the frugal state bandwidth of the generator function. It should be stepped down to:

$$h = mT = md\left(1 \pm \sqrt{2 - \frac{2Q}{md}}\right) \qquad \text{--(6a)}$$

Out of the two solutions, the best possible frugal state (the one which allows higher transmission rate in the frugal state) is:

$$h_{best} = mT = md\left(1+\sqrt{2-\frac{2Q}{md}}\right) \quad \text{--(6b)}$$

And the other solution is:

$$h_{other} = mT = md\left(1-\sqrt{2-\frac{2Q}{md}}\right) \quad \text{--(6c)}$$

The second solution, when exists, provides a second possible frugal state with lower generation rate. If this solution is taken, the data-generation allowance at frugal state will be lower. However, it will result in faster recovery.

The next question we ask is how long the system should stay in frugal state. We first derive a lower bound. This is given by the critical recovery time:

$$T_{critical} = 2T = 2d\left(1\pm\sqrt{2-\frac{2Q}{md}}\right) \quad \text{--(7)}$$

For the special case, when, the initial buildup (or reaction time) is zero, the corresponding height and duration of the frugal state is:

$$h_{best} = md\,(1+\sqrt{2}) \quad \text{--(8)}$$

$$T_{critical} = 2d\,(1+\sqrt{2})$$

For step *g(t)*, between the critical-point and recovery-point the system continues to be in frugal state accelerating the recovery. Corresponding recovery time is the complete duration of the frugal state. It can be determined by solving equality—(2), and is given by:

$$T_{recovery} = \frac{h}{m}\left(1+\sqrt{1+\frac{2Qm}{h^2}}\right) \quad \text{--(9)}$$

For the general case, when there is a buffer buildup due to the reaction delay=τ, the buildup can be estimated from the satisfied state generation rate and the reaction delay. Let *H* be the bandwidth satisfied state generation rate, when τ is small, *B* can be approximated by:

$$Q = \tau(H-\frac{m\tau}{2}) \quad \text{--(10)}$$

The slop *m* can be approximated from the effective RTT and the segment size (up to the current *threshold* TCP window grows exponentially).

$$m = \frac{B_{channel}}{RTT\left(\log_2\frac{B_{channel}}{2I}+\frac{B_{channel}}{2I}\right)} \approx \frac{2I}{RTT} \quad \text{--(11)}$$

Here $B_{channel}$ is the target channel bandwidth, *I* is the increment step or segment size and *RTT* is the round trip delay estimate used by TCP to resize its window. For symbiosis with the underlying transport protocol, each time a retransmission timeout event (at *t=0*), reported the *frugal state bandwidth* is determined as following.

$$g(t) = w(t) \quad when\ \xi = 0, t = 0 \quad \text{--(12)}$$

$$= md\left(1+\sqrt{2-\frac{2Q}{m.d}}\right) when \xi = 1$$

$$= w(t) \ \ at \quad t > T_{recovery}$$

5.3 Symbiosis Mechanism: The Transientware

The important task of gluing between the transport layer and the application unit (MPEG-2 rate transcoder) is finally performed by the symbiosis unit (Transientware Module or *TM*). The *TM* essentially executes the throttling model. It estimates the parameters required to execute the model by probing iTCP as

needed and finally it provides the rate parameter to the application as it requires operating in symbiosis. Below we describe its parameter estimation process and invocation operations.

5.3.1 Estimation of the Model Parameters from iTCP States

To be able to use the symbiosis throttling model described above, we now show how the model parameters can be estimated from the TCP state and event times made accessible by the iTCP. Namely, we want to find τ, *H*, *RTT*, and *I* from the TCP internal state variables now made available by iTCP.

A) Reaction Delay (τ)

The reaction time τ was approximated as following:

$$\tau = (t_{ResponseTime} - t_{EvenTtime}) + u_{rate} + u_{TCP} \quad \text{--(13a)}$$

`EventTime` is when the signal handler was invoked. The quantity u_{TCP} is a constant approximating the time taken by iTCP's kernel signaling. We assume $u_{TCP} = 0$. Thus `EventTime` is used here as an approximation of the real time when the event has occurred deep in the TCP layer. `ResponseTime` approximates the time of the real rate reduction (i.e. when the calculated h_{best} is saved to "`rate.par`" file). Quanltity u_{rate} is the estimate of the rate control systems reaction time after receiving the new rate, we also assume $u_{rate}=0$.

B) Round Trip Time (RTT)

RTT is directly returned by TCP from its state variable `TCPstate->t_rtttime`. TCP implementation uses the following process to measures round trip time (*RTT*) and retransmission timer out (*RTO*). First, TCP measures the *RTT* between sending a byte with a given sequence number and receiving an acknowledgment that covers that sequence number (*M* denotes the measured *RTT)*. Afterwards, TCP updates a smoothed *RTT* estimator *R* using the low-pass filter:

$$R \leftarrow \alpha.R + (1-\alpha).M \quad \text{--(13b)}$$

Where α is a smoothing factor with a recommended value of 0.9. The smoothed *RTT* is updated whenever a new measurement *M* is made. This means that 90% of each new estimate *R* is from the previous estimate and 10% is from the new measurement *M*. TCP then calculates a new retransmission timer out value (*RTO*) based on the mean and variance of the *RTT* measurement. The technique was proposed by Jacobson [Jac88]. He used the mean deviation as a good approximation of the standard deviation since it is easier to compute. In each *RTT* measurement *M*, the following calculations are made:

$$Err = M - A \quad \text{--(13c)}$$

$$A \leftarrow A + gErr$$

$$D \leftarrow D + h_{gain}(|Err| - D)$$

$$RTO = A + 4D$$

Where *A* is the smoothed *RTT* (an estimator of the average) and *D* is the smoothed mean deviation. *Err* is the difference between the measured value just obtained and the current *RTT* estimator. Both *A* and *D*

```
1:  SignalHandler(signum){
2:    struct evtSubInfo *handInfo;
3:    if (signum == SIGIO){
4:      gettimeofday(eventTime);
5:      s = GetSockid();
6:      ProbeEvtInfo(s, handInfo);
7:      if (!(child = fork())){
8:        execl(handInfo->handler, s, eventTime);
9:        exit(0);
10:     }//end if
11:   }//end if
12: }//end SignalHandler
```

(a)

```
1: Loss-TM(socket s, eventTime){
2: struct connState *TCPstate;
3:      probeSocket (s, TCPState);
4:      fscanf(timeFile, "%ld", videoStartTime);
5:      H = (TCPState->t_rtseq – TCPState->t_iss)*8
               / (videoStartTime - eventTime);
6:      gettimeofday(respTime);
7:      responceDelay = respTime - eventTime;
8:      m = 2*(TCPState->t_maxseg)*
                     8 / TCPState->t_rtttime;
9:      B=responceDelay * (H –(m*responceDelay)/2);
10:     h_best = m*d* (1 + sqrt(2-(2*M/m*d)));
11:     T_recovery = (h_best/m) *
                    (1 + sqrt(1+(2*B*M)/(h*h)));
12:     ratefile = fopen("rate.par", "w");
13:     fwrite(h_best, ratefile);
14:     StartRecoveryTimer(Recovery-TM);
15: }//end LossTware
16: }
```

(b)

```
1:  RecoveryHandler(signum){
2:     if (signum == SIGALRM){
3:       waitTimecount++;
4:       if ( waitTimecount && !rateOK &&
                       (waitTime > T_recovery)){
5:            ratefile = fopen("rate.par", "w");
6:            fwrite(originalRate, ratefile);
7:            rateoK = 1;
8:       }//end if
9:     }//end if
10: }//end RecoveryTware
```

(c)

Figure 20. (a) Signal Handler, (b) Loss TM and (c) Recovery handler

are used to calculate the next *RTO*. The gain *g* is for the average and is set to 1/8. The gain for the deviation is h_{gain} and is set to 1/4.

C) Maximum Segment Size (*I*)

RTT is directly returned by TCP from its state variable `TCPstate->t_maxseg`. Maximum segment size MSS (we called it *I* in our model), is the largest 'chunk' of data that TCP can send to the other end. When a connection is established, each end has the option to announce the MSS it is willing to receive. When TCP sends a SYN segment, it can send an MSS value up to the outgoing interface's MTU, minus the size of the fixed TCP and IP headers. In our experiment, TCP chose an MSS of 1460 bytes.

D) Satisfied State Bandwidth (*H*)

H can be calculated by finding the ratio: number of bytes transmitted so far over elapsed time since the video has started. This is estimated from two TCP state variables (*t_rtseq* and *t_iss*) and two local measurements:

$$H = \frac{(TCPstate \rightarrow t\_rtseq - TCPstate \rightarrow t\_iss) \times 8}{u_{videoStartTime} - u_{eventTime}}$$

The difference:

$$TCPstate \rightarrow t\_rtseq - TCPstate \rightarrow t\_iss$$

Between the state variables gives how many bytes have been transmitted so far. We multiply it by eight to convert it to bits since all our calculations will be in bit/second units. The time $u_{videoStartTime}$ is the time when the video started; it was saved in a file by the encoder prior to sending the first frame.

5.3.2 Transientware Implementation

The Symbiosis Throttling of equation 12 is actually implemented in the loss event handler or the *TM*. Basically, we need to calculate h_{best} and $T_{recovery}$ every time the *TM* is invoked. The role of the signal handler was merely to catch the signal from the kernel and invoke the appropriate *TM*. To simplify things we let the encoder subscribe with the *retransmit timer out* event only. Figure 20 (a) outlines a sketch of the signal handler code. After catching the `SIGIO` signal, it needs to know which socket generated the event (line 5) then it probes the socket to get the event number and the *TM* id (line 6). Once retrieved, it forks a new child and executes the appropriate *TM* for the event type (lines 8-10). If a loss event is detected, e.g., timer out event, the handler activates the *TM* shown in Figure 20 (b) which we call `Loss-TM`. The signal handler passes the socket id (`s`) and time when the event occurred (`eventTime`) to the *TM*. Once activated, the `Loss-TM` first probes the socket to retrieve the following parameters from TCP: `t_rtttime` (round trip time), `iss` (initial send sequence number), `t_rtseq` (sequence number being timed), and `t_maxseg` (maximum segment size). Then it calculates the satisfied state bandwidth generation rate *H*, the reaction delay τ as explained before. Afterwards, the `Loss-TM` calculates *m*, *B*, h_{best}, and $T_{recovery}$ in a straightforward manner (lines 8-11). In line 13, it stores the reduced rate h_{best} in the "`rate.par`" file which will be noticed immediately by the symbiotic encoder. Finally, it starts a timer for recovery and associates a handler (`RecoveryHandler`) with this timer—this handler is outlined in Figure 20 (c). When the timer reaches $T_{recovery}$, the recovery handler writes the normal rate (i.e., original rate before reduction) into the file "`rate.par`".

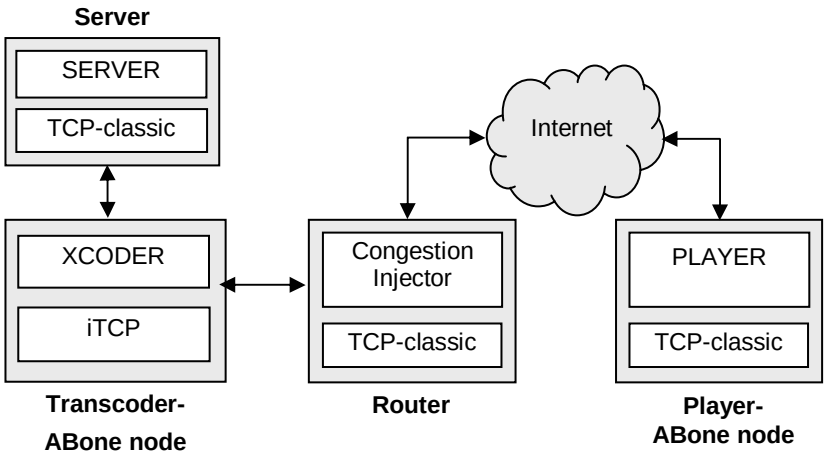

Figure 21. Video transcoder experiment setup

5.4 Experiment and Performance Analysis

We ran the experiment using the real implementation of iTCP kernel and the MPEG-2 Symbiotic Transcoder. The performance results were obtained from a live experiment of video delivery sessions over the Internet. Before presenting our results first we will describe the testbed and the setup.

```
int bursts = 3;
int burstTime[]={3, 3, 3};
int interBurstTime[]={10, 10, 0};
sleep(10);
for (i=0; i<bursts; i++){
   remove entry from routing table;
   sleep(burstTime[i]);
   return entry to routing table;
   sleep(interBurstTime);
}
```

Figure 22. Congestion Injector mechanism

5.4.1 The ABone Testbed

We wanted to run the experiment on the real Internet environment. This required running the symbiotic transcoder, a sender equipped with iTCP transport protocol, and a set of players on remote hosts around the world. We could have done this manually by conventional methods to reach a number of remote nodes worldwide. But this would have required extensive overhead to setup the testbed and maintain. Therefore, we decided to run the experiment on ABone testbed [Ber02b]. The ABone, developed under the DARPA Active Network program forms a virtual network infrastructure on which a growing set of active network components can be tested and experimentally deployed. ABone is an operational network and provides an Internet wide network of routing as well as processing capable nodes. Providers can contribute confederation of computing capable nodes. Independent application involving multiple trust domains can be securely launched and executed. It also specifically allows new transport protocol components to be remotely deployed. ABone nodes are available from Europe, Asia and North America. Individual nodes are contributed and managed locally and independently by the contributing site administrators. However, the administrators do not have to manage the remote users. Researchers can remotely install and execute programmed components on any collection of these nodes via the ABone backbone management and control backplane being a part of a centralized user pool. The codes are distributed via an enlisted set of Trusted Code Servers (TCS), which help authenticating them prior to distribution. The security domains are handled by the backplane control system. The backplane is being maintained by the ABone Coordination Center (ABOCC) at ISI at the University of Southern California. ABone status can be monitored live from the ABOCC web site [Ber02b]. In addition to the iTCP machine we have a cluster of 10 registered ABone

Table 10. Player locations on the ABone

Target ABone node	Country	RTT measurment				Number of hubs
		min	average	max	mean deviation	
ave.willab.fi	Finland	0.16355	0.16606	0.16647	0.798	24
zzz.abone.supermedia.pl	Poland	0.14705	0.14844	0.15701	3.023	23
abone-01.cs.princeton.edu	USA	0.03945	0.04002	0.04524	1.319	17
dad.isi.edu	USA	0.06548	0.06572	0.06610	0.186	19

Table 11. Experiment control flags and running modes

Control flag	Effect
iTCP	Turns on/off the interactivity service.
EVENT	Turns on/off the event notification service.
SYMB	Turns on/off the symbiosis feature of the transcoder. When this flag is set, the signal handler invokes the event handler to reduce the bit rate of the decoder. Otherwise, the signal handler just records the event type and time.
OPT	Means (OPTimal mode). Used to choose between two modes of Symbiotic rate reduction (i) optimal backoff mode which uses the symbiosis throttling model described in section 4. or (ii) exponential backoff mode which uses a preset retraction rate and duration.

Running mode	Control Flags				Comments
	iTCP	EVENT	SYMB	OPT	
iOPT	ON	ON	ON	ON	Full interactivity. Use the optimal backoff symbiosis throttling.
iEXP	ON	ON	ON	OFF	Full interactivity. Use the exponential backoff symbiosis throttling.
iOFF	ON	ON	OFF	X	Subscribe, report event, but do not change bit rate. Used to measure overhead.
Classic	OFF	X	X	X	Turn off all interactivity features.

Table 12. Average frame delay and acceptance ratio

mode		princeton.edu		isi.edu		willab.fi		supermedia.pl	
		Average Delay	Accept Ratio	Average Delay	Accept Ratio	Average Delay	Accept Ratio	Average Delay	Accept Ratio
d=2	iOPT	0.518	0.797	2.018	0.415	2.504	0.319	1.38	0.692
	iEXP	2.613	0.529	-0.015	1	-1.239	1	2.411	0.284
	iOFF	6.279	0.455	10.82	0.197	8.752	0.155	8.485	0.133
	Classic	3.047	0.461	10.957	0.217	6.615	0.273	8.485	0.147
d=4	iOPT	0.897	0.976	2.029	0.737	-0.641	1	0.727	1
	iEXP	2.613	0.529	-0.015	1	-1.239	1	2.411	0.777
	iOFF	6.279	0.455	10.82	0.197	8.752	0.293	8.485	0.277
	Classic	3.047	0.805	10.957	0.395	6.615	0.299	8.485	0.291
d=6	iOPT	0.883	1	3.974	0.679	1.623	1	1.387	1
	iEXP	2.613	0.997	-0.015	1	-1.239	1	2.411	1
	iOFF	6.279	0.455	10.82	0.329	8.752	0.295	8.485	0.277
	Classic	3.047	0.805	10.957	0.395	6.615	0.535	8.485	0.52

nodes in our lab at Kent State University (mk00-mk09.maunakea.medianet.kent.edu). Four of these nodes run on FreeBSD and the rest run on Linux. At the time of our experiment (Nov. 2003), there were 24 Linux nodes, 5 Solaris nodes, and 12 FreeBSD nodes registered at the ABone. For our experiment we simply sent our video player to one of the ABone's trusted code server at (http://bro.isi.edu/KENT). Then we configured and registered our iTCP-kernel machine (kawai.medianet.kent.edu) as a primary node on the ABone to run iTCP and the symbiotic transcoder. The server remained in a traditional (non active) node. The ABone allowed the automatic loading of the sessions on designated machines worldwide.

5.4.2 Experiment Setup

This experiment describes the performance of an MPEG-2 ISO/IEC13818-2 (176×120) resolution video encoded with base frame rate of 2 Mbps at main profile. Figure 21 illustrates the deployment setup.

The video server runs on a classic TCP machine (*manoa*) and feeds the video stream into the transcoder, which runs on the iTCP active node (*kawai*). To create some forced congestion in the experiment we also run a congestion injector program on a first-mile active gateway router (*lahaina*). The injector creates congestion bursts. Figure 22 shows the congestion injector. It allows the *duration*, and the *interval* between bursts to be programmed for three consecutive bursts. During a congestion burst the router will simply disrupt its routing table by removing the entry that leads to the player machine. When the burst time is over, the router restores the routing table back to normal. In our experiment, we used 3 three-second bursts at 10 second intervals. A three-second burst usually triggers 1 to 2 retransmit timer out events depending on the player's location. We ran the players on selected remote ABone node. We repeated the experiment on four ABone nodes, two in the US and two in Europe. Some general network conditions observed of the four target nodes are shown in Table 10. The player and transcoder units were enhanced to collect detail frame arrival, and delivery measurements.

5.4.3 Impact on Video Frame Delay

For detail comparison we have performed several sets of experiment. These are: `iOPT`, `iEXP`, `iOFF`, and `Classic` modes. In the `iOPT` (optimal backoff) mode, we activated the throttling model described above to calculate h_{best} and $T_{recovery}$ with every loss event. We challenged the system to provide the frames at guaranteed *d=6, 4,* and *2* seconds. As a base case we also repeated the experiment with the same congestion schedule in classical mode where the interactive and symbiotic rate adaptation features were turned-off and the entire system run in classical TCP mode. We call it `Classic` mode. For comparison we also included the case of `iEXP` used in our previous work to demonstrate the effectiveness of iTCP. It is a non-optimized simple heuristics-based symbiosis which performs a lazy binary back-off scheme for the generation rate. The method adapts but it can not provide QoS guarantee as of the throttling model. Detail of this simple scheme is in [Kha03c]. With the `iEXP` (exponential backoff) mode, we used a predetermined

Table 13. h_{best} and $T_{recovery}$ statistics for three ABone nodes

d(sec)	event	princeton.edu		willab.fi		supermedia.pl		isi.edu	
		h_{best}	$T_{recovery}$	h_{best}	$T_{recovery}$	h_{best}	$T_{recovery}$	h_{best}	$T_{recovery}$
2	e1	512240	1.520135	1443311	15.68562	1333511	14.49148	824810	4.952
	e2	491954	1.459965	1292875	14.04808	1223663	13.29792	969318	5.81185
	e3	496552	1.476599	1309004	14.22661	1257601	13.66691	891772	4.94007
4	e1	279963	0.851075	602184	6.551785	665086	7.228908	453645	2.7236
	e2	261526	0.792414	564819	6.145352	584324	6.355469	499010	2.58743
	e3	259565	0.788820	604674	6.579283	602115	6.550372	399208	2.47848
6	e1	186117	0.573669	486808	5.301540	467211	5.085250	340233	1.90652
	e2	173629	0.525550	419186	4.557723	401019	4.368733	323222	1.73493
	e3	172046	0.539606	307244	3.343913	411801	4.480751	299405	1.86838

reduction ratio (α = 0.35) and multiplied that with current bit rate to calculate the frugal state bit rate, we also used a fixed recovery time of 4.0 seconds. We also repeated the experiments in another mode called `iOFF` for overhead estimation. The mode is similar to classic TCP. No symbiosis is performed. But the event subscription mechanism remains active. This will be explained later. In all the iTCP enabled runs (`iOPT`, `iEXP` and `iOFF`), the transcoder subscribes with iTCP for the retransmission timer out event. In the experiment, we took frame-wise detail event trace of the first 750 frames of the video at both sending and receiving ends. For a given discard threshold time in the receiving end we also traced which frame was successfully received or not at the MPEG-2 player. As explained earlier, we traced four transport aware cases (`iOPT` with three values of delay tolerance *d=2, 4,* and *6* seconds and `iEXP`) and two transport unaware cases (`iOFF` and `Classic`). Please, return to Table 11 for running modes details.

Now we show the dramatic impact of iTCP's interactivity based symbiosis. In Figure 23 we plot the delay experienced by the video frames in terms of frame arrival time at the player for the six modes mentioned above. In addition to that, we also show the ideal expected frame delivery time—`Expected` in the figure—based on linear generation rate. As can be seen iTCP outperformed classical TCP; after every

Table 14. Percentage of total bits delivered for each mode

	princeton.edu		isi.edu		supermedia.pl		willab.fi	
	Target	**Actual**	**Target**	**Actual**	**Target**	**Actual**	**Target**	**Actual**
iOPT, d=2	0.912	0.913	0.898	0.901	0.886	0.886	0.929	0.929
iOPT, d=4	0.966	0.966	0.892	0.897	0.814	0.826	0.789	0.791
iOPT, d=6	0.975	0.98	0.94	0.945	0.87	0.88	0.867	0.874
iEXP	0.843	0.843	0.835	0.835	0.862	0.862	0.86	0.86
iOFF, Classic	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

congestion burst, the unaware cases (`Classic` and `iOFF`) continuously fell behind. The delay built up and it could hardly recover. This is evident by the step jumps in the delay line. The TCP aware cases also suffered some step buildup, but it was much smaller and it could recover after few seconds due to the rate retraction. In Table 12 we present the frame delay and acceptance ratio comparison for the whole stream. The table shows the performance for three choices of delay tolerance *d=2, 4,* and *6* seconds. For each value of *d* we traced the four running modes (`iOPT`, `iEXP`, `iOFF`, and `Classic`) and recorded the average delay in seconds that each frame has experienced and the frame acceptance ratio at the four receiving player ABone nodes. It can be clearly noticed that iTCP/aware modes achieved low delays and high acceptance percentages while the unaware/classic modes suffered from higher delays and lower acceptance percentages. We present this information visually in Figure 24. In this figure we show the number of frames accepted at by the video player for the three choices of delay tolerance. Clearly iTCP's *TM* mechanism allowed the application to use sophisticated optimization techniques to optimally control the temporal qualities of its traffic.

5.4.4 Symbiotic Rate Control

In the next set of experiments we present the internals of the symbiosis mechanism in more detail. Figure 25 depicts the symbiotic frame rate transcoding that occurred due to the joint rate specification at the rate control logic at the symbiosis unit and in the transcoder for each frame. In the figure we show four plots for the four target ABone nodes. Each plot represents the iOPT mode run for the delay tolerance case *d=4*. Table 13 presents the actual values of h_{best} and $T_{recovery}$ that controlled the frugal mode operation as calculated by the symbiosis controller *TM* after being activated by each one of the three loss events created in the experiments. In each plot of Figure 25 we see the target bit rate and the retraction ratio as specified by the symbiosis controller, and the resulting outgoing actual frame rate generated by the transcoder. The timer out events (in this case there are 3 timeout events) reported by iTCP resulted in the symbiosis unit to modify the rate according to the optimal backoff symbiotic rule (equation 12). Though, the precise MPEG-2 generation rate varied widely from frame to frame to accommodate the frame type, but the general trend followed the specified target. Table 14 provides the overall stream compression due to symbiotic adaptation for the entire stream (`iOPT` and `iEXP` cases), as compared to the normal non-symbiotic cases (`iOFF` and `Classic` cases). In the `Classic` and `iOFF` cases, there were no adaptation (thus retraction =1). Compared to this both `iOPT` and iEXP reduced the overall delivered bits about 83-95%. However, it is interesting to note that `iEXP` without its optimization logic, operated more aggressively and compressed

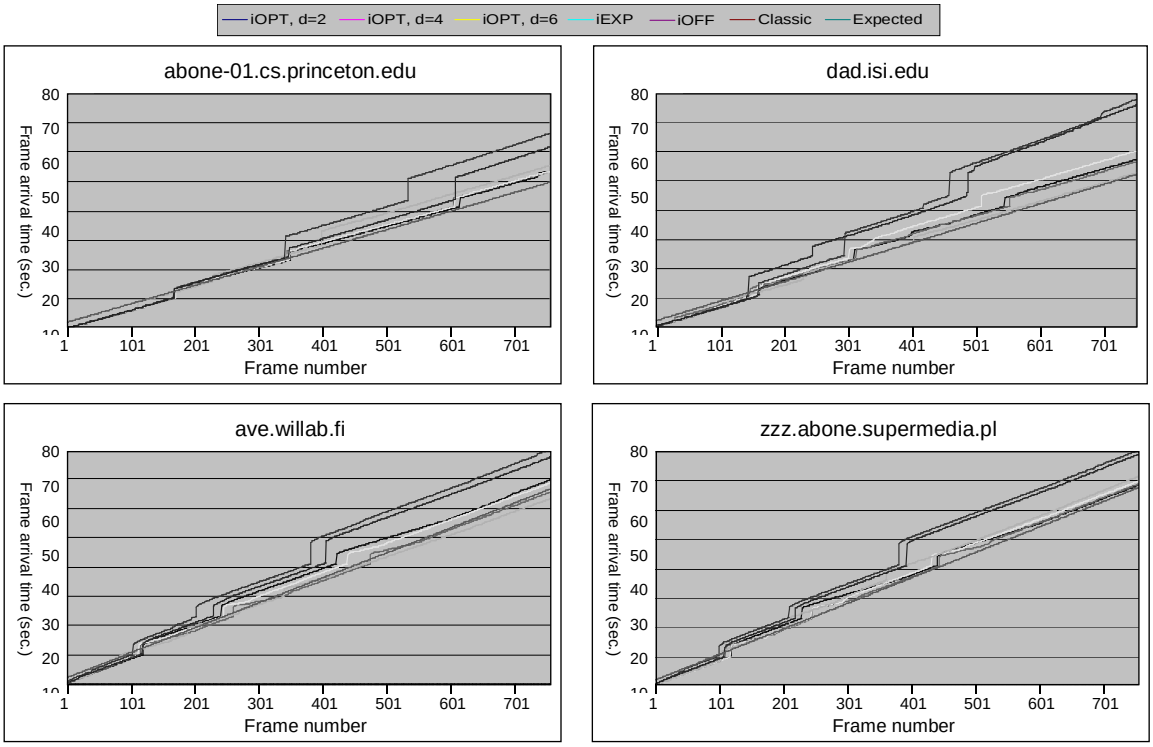

Figure 23. Frame Arrival Trace

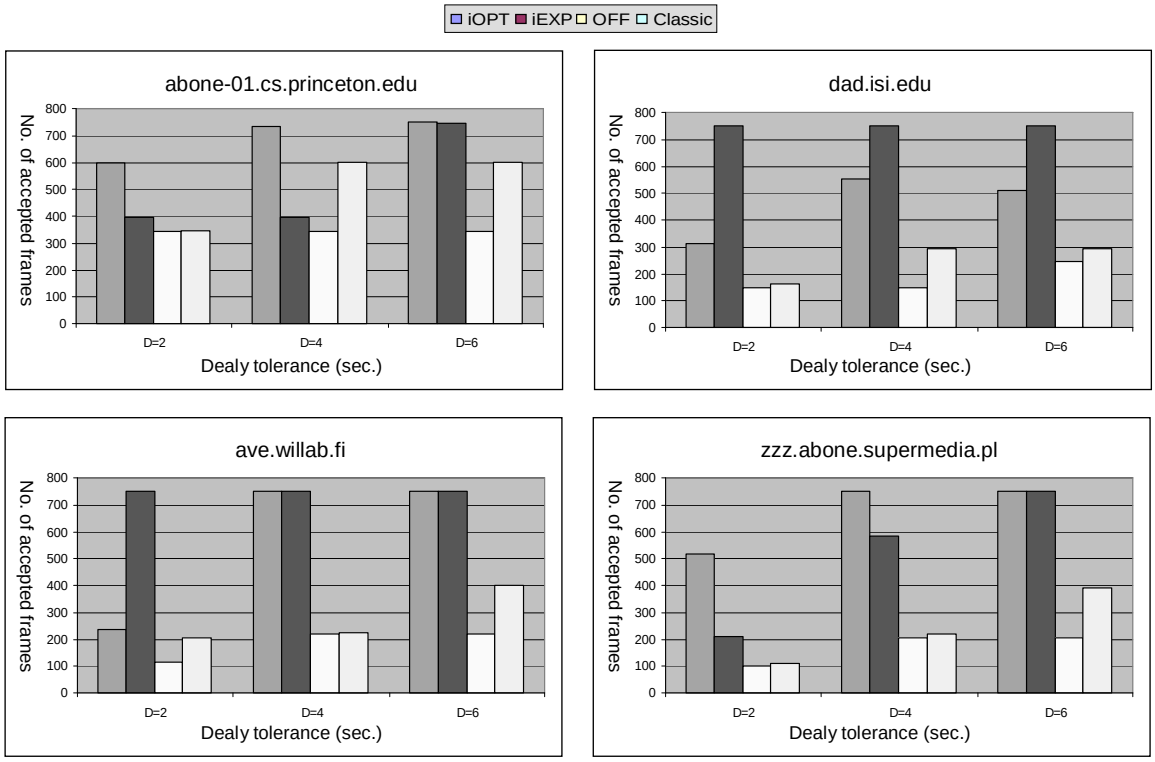

Figure 24. Number of frames accepted for three values of delay tolerance

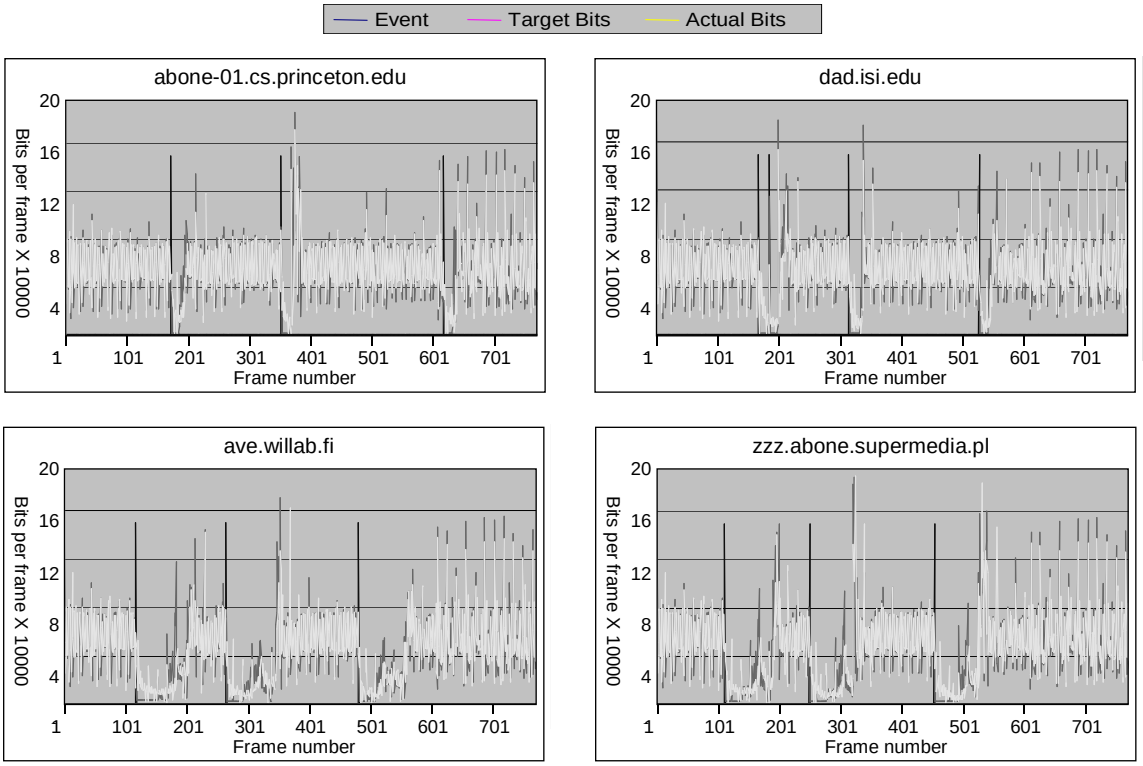

Figure 25. Symbiotic Rate Reduction

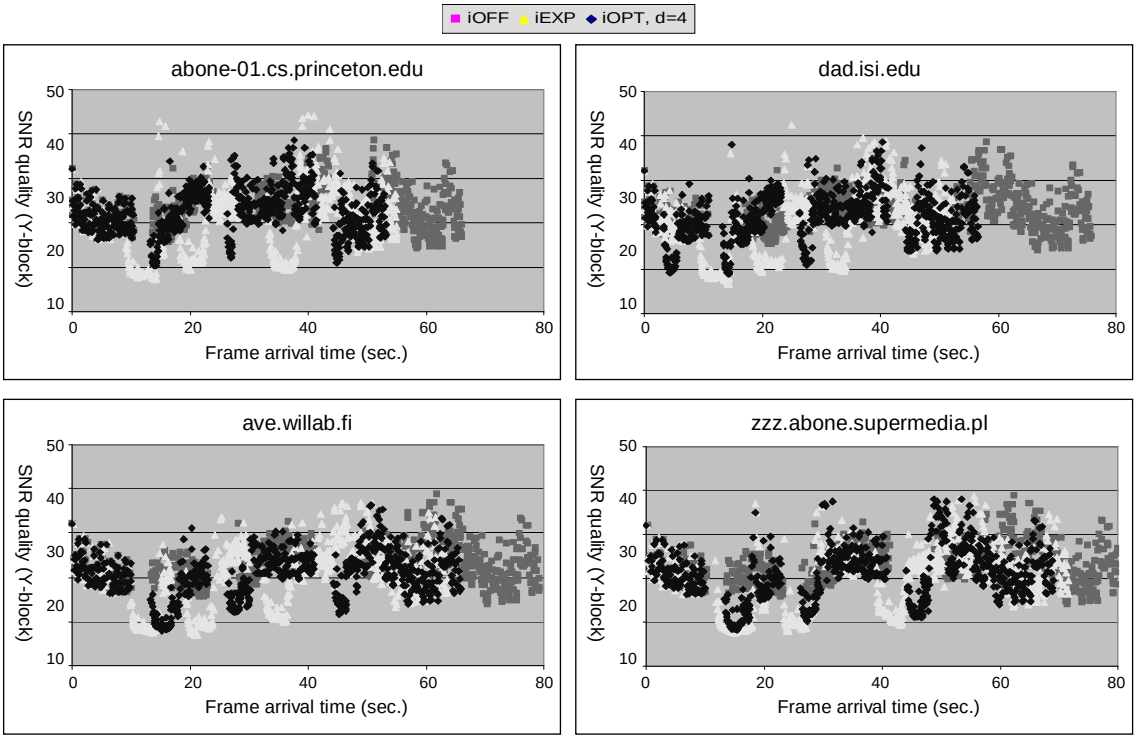

Figure 26. Frame Arrival time and frame SNR quality tradeoff

more (for example `iEXPs'` 85% vs. `iOPTs'` 91% in *d=2*). In comparison `iOPT` operated more confidently (i.e., reduced less bits), yet achieved higher temporal quality (average delay is 2.6 sec vs. 0.5 sec for same cases).

5.4.5 Observation at Application Level:

In the above experiments we illustrated how the symbiosis mechanism worked from the video transport protocol (MPEG-2) and the network transport protocol TCP layers beneath it. In this plot we will illustrate how this mechanism appears from the very top—at the application layer itself. An application receives and delivers uncompressed frames. The performance metric this end-system uses is the temporal and spatial quality difference between the transmitted and the reproduced uncompressed video frames. The underlying MPEG-2 system and the network layer TCP together provide the transport service. The specific compression, windowing etc. and other detail mechanisms are external techniques to the end systems.

In Figure 26 each frame is plotted as a point in the video quality/frame delay plane. The figure shows four plots for the four ABone nodes, and each plot represents three running modes (`iOPT` with *d=4*, `iEXP`, and `Classic`). As can be seen from the region of the three QoS distributions, in TCP-classic, although frames have been generated with SNR quality ranging between 18-40 dB, but many of these frames suffered long delay and were lost in transport. In contrast, the interactive `iOPT` mode managed to deliver all frames with guaranteed delay with the bulk of the frames had 10-32 dB quality. It is interesting to note that the `iEXP` mode achieved the same tradeoff, but since it took a non-optimized and thus more aggressive approach in symbiotic rate reduction the quality suffered more loss and recorded values as low as 7 dB. Fundamentally, what iTCP has offered is a qualitatively (as opposed to the quantitative improvements offered by any unaware solution) new empowering mechanism, where the catastrophic frame delay can be traded off for acceptable reduction in SNR quality.

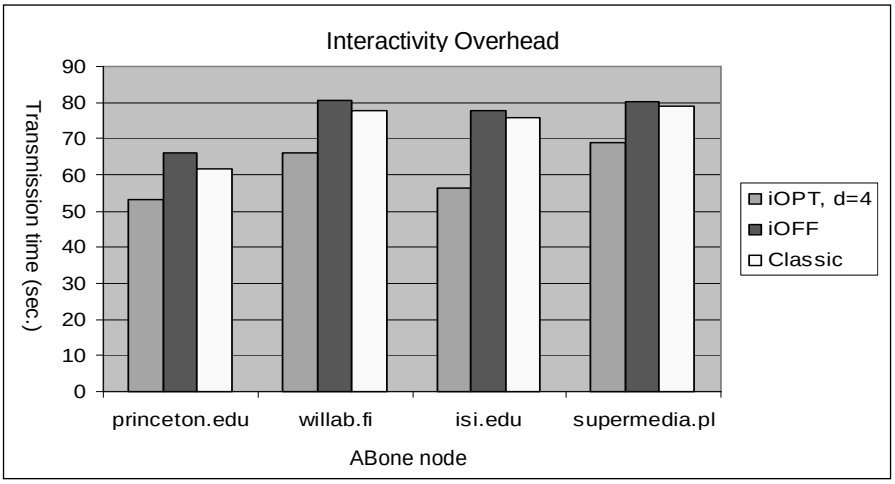

Figure 27. Interactivity service overhead

5.4.6 Interactivity Overhead

The dramatic advantage in application level performance came at a cost since the event tracking mechanism added some overhead. We were also curious to find out the overhead of the event mechanism. To track the overhead, we recorded the total data transmission time under the three conditions (`iOPT`, `iOFF`, and `Classic`). The left most bar of Figure 27 plots the transport time for the optimal interactive mode where we activate both event delivery and symbiosis. To observe the overhead of the event service, in the `iOFF` mode we used the iTCP implementation, however, we stopped the symbiotic reduction so the transport layer handled the same amount of data. As expected the overall transmission time increased in all three cases. However, in the third column (`Classic` mode) run we turned off the interactive service altogether and thus we saved its overhead and lost its benefit. As can be seen, the slight increase in the

event delivery overhead was vastly offset by the application level technique. The advantage the application gained from the event delivery was much bigger than the overhead.

5.5 Conclusion

In this chapter, we have presented a case of rate symbiosis mechanism in line with current advances in TCP-friendly systems. We collected the results of our experiment by running the video session on the global Active Network (ABone) testbed. In the previous discussion we have demonstrated the case of quality conformant congestion control for time-sensitive video traffic. The approach exposed the overall advantage of network 'friendly' applications. However, it also departs significantly from the mainstream TCP friendly systems that have been suggested recently in two senses; First, it does not add any new major component in network software structure. One of the principal strength of the proposed scheme is its relative simplicity at network layers—yet its effectiveness. It only expects some form of interactivity directly from the concerned network protocols as a general interface feature. Thus, there is no expectation of (or conflict with) additional services (such as combined congestion control from multiple applications). Secondly, the applications do not have to be designed dependent on other auxiliary indirect probing tools or network utilities, nor it excludes their use when available. Some of the information measured by the auxiliary tools suggested by other approaches might be already available (or are being estimated/tracked) at lower layers anyway. At least this is the case with TCP congestion. The direct protocol interactivity we propose thus seems to be the logical path that can avoid potential duplication of efforts.

CHAPTER 6

IPMN: Interactive Protocol for Mobile Networking

There are two well known challenges in the literature for Internet host mobility; (i) improving TCP level performance and (ii) reducing handoff latency. Until now, most solutions that were proposed were kept in lower network layers (i.e., link layer, IP, or TCP layer). In this chapter we present an end-to-end host mobility solution based on the *InTraN* paradigm. The solution allows continuous operation of TCP between the two endpoints even with the presence of handoffs and long disconnections, and it also enables the mobile node to perform fast handoffs with minimum or no loss. We have implemented the scheme on FreeBSD and tested the real system over the Internet. We show with experimentation on three types of traffic (Voice, WWW, and FTP) that our scheme can substantially reduce handoff latency and improve TCP performance.

6.1 Introduction

Classic IP protocol was designed long time ago for wireline Internet with no support for node mobility. Its routing mechanism relies on IP address semantics to deliver packets to a destination node whose location is assumed to be fixed. The same argument also applies to TCP, whose congestion control mechanism assumes that the path between the two endpoints is 'wired'. Since the advent of wireless technology and the tremendous growth of mobile networking applications, a persistence need has emerged to remedy the TCP/IP stack and make it *mobile networking compatible*. On the IP level, we need to be able to deliver packets to a mobile node regardless of its current point of attachment, and on the TCP level we need to be able to identify the reasons behind packet loss and react to them differently; if loss was due to congestion on the wired link, we let TCP run its native course–invoke the appropriate congestion control procedure. However, if the loss was due to radio disturbance on the wireless link or due to handoff disconnection, TCP should retransmit as soon as possible without any rate throttling.

Fortunately, classic IP allows a mobile node to roam from one access point to another as long as it remains in the same IP subnet. In this case, the mobile node has to perform link-layer (L2) handoffs in order to maintain its wireless connectivity and these L2 handoffs remain transparent to the IP layer (L3). However, if the mobile node migrates to a different IP subnet, its current IP address becomes topologically invalid and it must acquire a new IP address from the newly visited network, i.e., it must also perform L3 handoff. Otherwise, all its existing TCP/IP connections become useless.

Extensive research has been done recently to address these problems. One of the most well known mobility solutions on the IP level is Mobile IP (MIP) [Per96] which has been endorsed by the IETF. MIP provided a global logical solution by introducing indirection through a set of Mobility Agents. Each mobile node is identified by an IP address assigned to it by its home network—called *home address*—regardless of its current point of attachment. MIP introduced three new entities, namely the *home agent*, the *foreign agent* and the *mobile node*. Whenever a mobile node performs L3 handoff, it must register its current point of attachment with the home agent. For every registered mobile node, the home agent intercepts all incoming traffic from a given sender—usually referred to as the *correspondent node*—and redirects it through tunneling (packet encapsulation) to the mobile node's most recently registered location. Traffic from the mobile node to the correspondent node is routed normally (possibly bypassing the home agent). This kind of traffic flow is referred to in the literature as *triangular routing*. In MIP, foreign agents periodically broadcast *agent advertisements* to detect mobile node's movement. When the mobile node decides to migrate to a new subnet, it configures a new care-of-address, and then it registers this address with the home agent. The home agent updates its address binding cache and sends an ACK to the mobile node. Communication between the two endpoints cannot resume until registration is completed at the home agent.

Although MIP has provided a global solution for IP level mobility, but it has also introduced its own performance problems. Some of them stem from the complicated handoff procedures which resulted in longer handoff latencies—which have also affected TCP level performance—and others from the longer routes due to triangular routing. Since its release, MIP has gone through several modifications like route

optimization extensions [Per00] [Per01]. Actually, a great deal of the research on mobile networking is focusing on MIP and its performance. For example, Researchers aimed at reducing registration signaling delay by introducing a hierarchical structure and therefore allowing regional registration and reduced round trip delay [Cam01] [Gus01] [Hri02] [Ram99]. Other proposals took a different approach by suggesting a deployment scheme of MIP based on existing infrastructure like The RAT (Reverse Address Translation) scheme [SiT99] which is based on the network address translation (NAT) protocol [Sri99].

At the TCP level, many solutions have been proposed to fix its performance problem over mobile networks. These solutions have been classified in the literature in three categories; link layer protocols [Aya95] [Bal95], split-connection protocols [Yav95] [Bakr97], and end-to-end solutions [Mat96] [Baks97]. A good survey on TCP extensions for mobile networks can be found in [Anj03] [Ela02]. Most of these proposals however, mainly targeted TCP level performance assuming the IP mobility solution already existed. Therefore, to solve the mobility problem on both levels (TCP and IP), we can either (i) combine a TCP level solution with an IP mobility solution (e.g., use SACK [Mat96] over MIP), or (ii) just propose one complete solution for both problems. The first option can be extremely costly in terms of extra overhead—and probably redundancy—due to lack of synchronization between the two solutions. The second option may be effective only if both protocols can share relevant events and state transitions (possibly through a third party) to be able to synchronize their actions.

In this chapter we present IPMN (*Interactive Protocol for Mobile Networking*). IPMN provides a solution for IP mobility problem as well as TCP performance problem over mobile networks. The scheme—which diverts from the MIP approach—is based on the *InTraN* paradigm. With IPMN the correspondent node can send packets directly to the mobile node and eliminates triangular routing. More importantly, it can also perform rapid handoff with very little or no loss of TCP segments.

6.2 Related Work

We found several proposals in the literature with some kind of protocol interactivity. For example, Wu, et al. [Wu01] proposed an intelligent handoff scheme for mobile wireless Internet over MIP. One aspect of this scheme is to let L2 send a notification to L3 whenever L2 has successfully finished performing handoff. Also, Fikouras, et al, [Fik01] aimed at reducing movement detection delay in MIP by introducing a *hinted* based movement detection algorithm called *Fast Hinted Cell Switching* (FHCS). The scheme allows L2 to send 'triggers' to L3 whenever a handoff event is initiated. These proposals have shown that such simple form of interactivity has an obvious advantage. However, they are fundamentally different from our scheme in two aspects: (i) they are based on MIP while our scheme offers a complete mobility solution that can replace MIP, and (ii) their event notification remains within lower network layers. But, since our interactive paradigm allows interactivity between lower layers and the application layer, we can deploy the solution at the application layer itself which has several advantages over low-layer solutions only.

6.3 Interactive Protocol for Mobile Networks (IPMN)

6.3.1 The Scheme

We employed the *InTraN* paradigm to design a global IP level mobility solution which also incorporates a TCP level performance solution during handoff. The basic idea of our scheme is to enable the mobile node to obtain a new IP from the future access router before handoff is performed, replace the 'source IP' field in the TCP/IP stack of the mobile node with the new IP, and relay the new IP to the correspondent node. Once it receives the new IP, the correspondent node immediately switches to the new IP by replacing the 'destination IP' field in the TCP/IP stack with the new IP. A best case scenario for this scheme would happen if the mobile node can locate the new access router and obtain a new IP address (e.g., through a DHCP server) before loosing connection with its current access router. Once it obtains its new IP address, the mobile node proceeds with L3 handoff as follows:

1. Freeze the TCP connection by advertising a zero window to the correspondent node.
2. Perform actual L3 handoff by replacing the IP fields in the TCP/IP stack at both the mobile node and the correspondent node with the new IP address.
3. Wakeup TCP by advertising a nonzero window to the correspondent node.

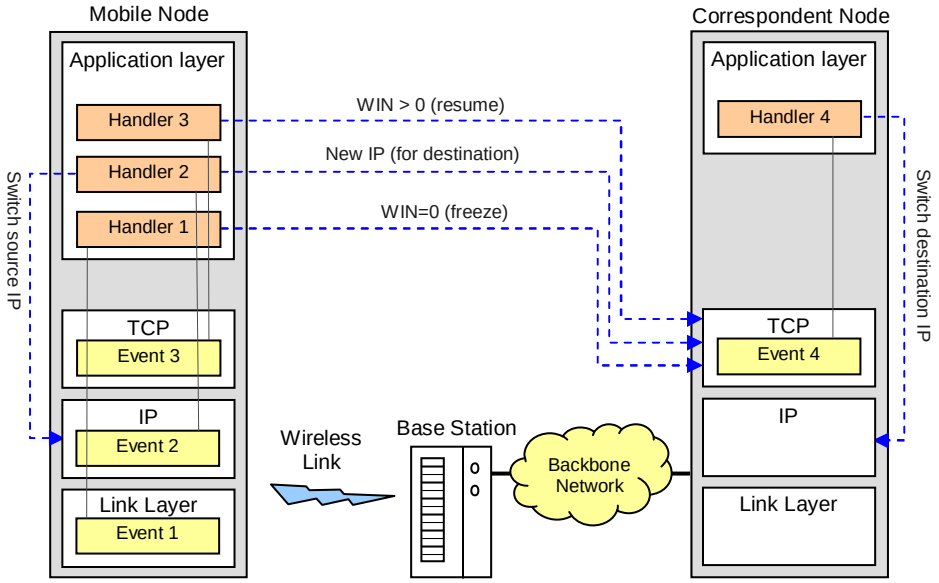

Figure 28. *IPMN-Full* architecture

Handoff pre-processing, i.e., locating a future access router and obtaining a new IP address, can also be done at the application level prior to L2 handoff. Fortunately, since we allow protocol interactivity, we can configure L2 to send an early signal the application layer about an impeding handoff. This gives the application layer a grace period to do all this 'bookkeeping' while it is still connected through the current access router. Naturally, a simple application level IP-lookup module should perform the task. We can benefit from interactivity again by allowing this IP-lookup module to probe L2 for the identity of the next access router (e.g., its IP address). Then, this module can contact the router and obtain the next IP address via a DHCP attached scheme. A number of previous works like [Fik01] [SiT99] and [Yok02] have shown excellent schemes that can support this methodology. We can re-model these schemes—or some aspects of them—via the *InTraN* paradigm to implement the handoff pre-processing illustrated above. Furthermore, we believe that since the mobile node can obtain a new IP before handoff, this pre-processing should not impact handoff latency. The purpose of this current implementation of IPMN is to experiment the basic idea of physically changing the IP number at both end-points whenever the mobile node configures a new IP address. Therefore, this version of IPMN, only implements the three-step L3 handoff procedure shown above.

6.3.2 The Architecture

We propose two modes of our scheme: A light-weight implementation that we call *IPMN-Half* and a more robust, heavy-weight implementation we call *IPMN-Full*. The primary difference between the two modes is the amount of internal inter-protocol interactivity involved. While in *IPMN-Half* a mobile node uses an explicit application-level message to relay the new IP number to the correspondent node, in *IPMN-Full* the mobile node uses interactivity and TCP level communication to perform the same task.

A) IPMN-Full Mode:

Figure 28 describes the conceptual architecture of *IPMN-Full*, and Table 15 describes the corresponding events and their handling sequences at each endpoint. At the mobile node, when the link layer detects signal fading and initiates L2 handoff (event 1), it signals the subscribing application. When the event is received at the application layer, a Transientware module (handler 1) is activated immediately; this module simply makes a simple system call which lets TCP advertise a zero window to the

Table 15. *IPMN-Full* events

Node	Event No.	Layer	Event tracked	Action taken by event handler
Mobile Node	1	LL	Wireless signal fading. Prepare to perform handoff to next BS.	Advertises a zero window to the FH. The freeze mechanism of TCP will force the FH to stop transmission.
	2	IP	A new IP has been assigned to the MN from the new BS.	Call the switch_ip() system call. This will replace the source IP filed in the IP header of the MN with the new IP and will send a segment to the FH with TCP option = SWITCH_IP to replace the destination IP field on the FH.
	3	TCP	The 'SWITCH_IP' segment has been ACK-ed.	Advertises a non-zero window to the FH. This will unfreeze the connection and enable the FH to resume transmission.
Fixed Host	4	TCP	A special TCP segment received with TCP option=SWITCH_IP.	Strip the new IP number from the options part of the segment, then call the Switch_IP() system call which stores the new IP in the destination IP field of the IP header overwriting the old IP number.

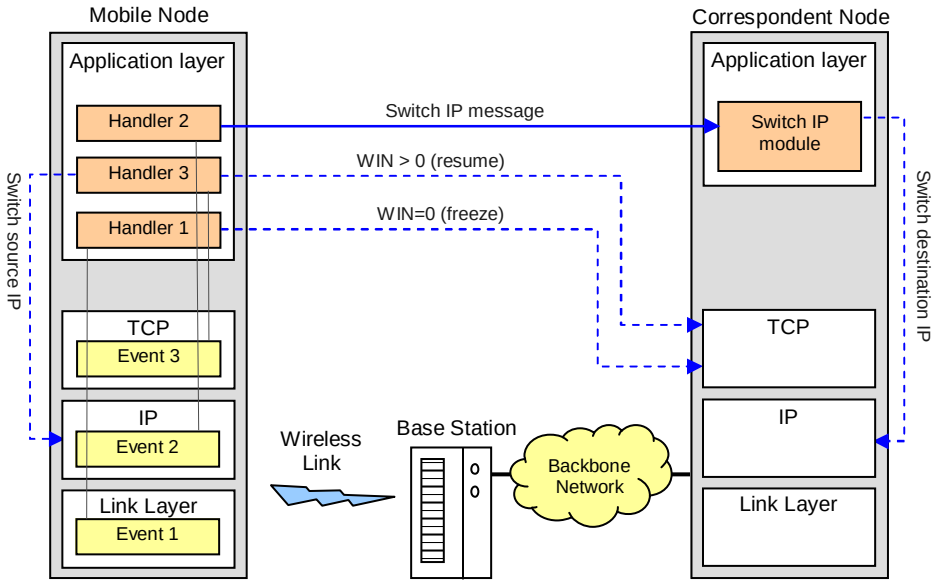

Figure 29. *IPMN-Half* architecture

correspondent node. This would normally cause the correspondent node to stop transmission. When the mobile node gets a new IP from the future network (event 2), it activates (handler 2) which transmits the future IP to the correspondent node at TCP level through a system call. The new IP is sent in a special TCP segment with '`option=SWITCH_IP`'. At the correspondent node, When TCP recognizes this option (event 4) it activates (handler 4) which then triggers a `switch_ip()` system call to replace the 'destination IP' field in the TCP/IP stack with the newly received IP number. Meanwhile, at the mobile node (handler 2) also makes a similar system call which changes the 'source IP' filed in its own TCP/IP stack. When the previous '`SWITCH_IP`' segment is ACKed at the mobile node (event 3), the mobile node advertises a non-zero window to the correspondent node which enables it to resume transmission.

Table 16. *IPMN-Half* events

Node	Event No.	Layer	Event tracked	Action taken by event handler
Mobile Node	1	LL	Wireless signal fading. Prepare to perform handoff to next BS.	Advertises a zero window to the FH. The freeze mechanism of TCP will force the FH to stop transmission.
	2	IP	A new IP has been assigned to the MN from the new BS.	Send a special message to the peer application on the FH. The message carries the new IP just assigned to the MN. When the FH receives the message, it runs a module that makes a special system call Switch_IP(). This system function will replace the destination IP field in the IP header with the new IP.
	3	LL	Handoff has just finished	Advertises a non-zero window to the FH. This will unfreeze the connection and enable the FH to resume transmission.

B) IPMN-Half Mode:

In this mode, we deploy the interactive protocol only at the mobile node. Figure 29 depicts the conceptual design and Table 16 describes the corresponding events and their handling sequences. Events 1 and 3 have the same meaning and handling as in the previous 'Full' mode. Event 2, however, is handled differently; when (handler 2) is activated, it makes a system call to probe the TCP layer for the new IP number. It then sends this IP number to the correspondent node using a normal `write()` socket operation. Naturally, since the correspondent node can '`read()`' the new IP directly from the socket, it does not have to catch any events or activate handlers. When the correspondent node receives the message it strips off the IP number and makes a `switch_ip()` system call—as in the previous mode—to change the 'destination IP' number in the TCP/IP stack. The remaining procedure is identical to the 'Full' mode.

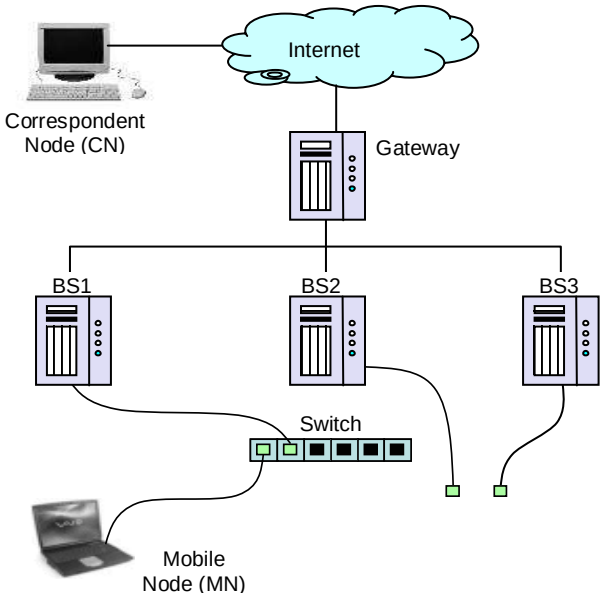

Figure 30. Experiment setup

C) Freeze TCP:

Advertising a zero window to the correspondent node to temporarily freeze the TCP connection was proposed in [Gof00] by Goff, et al, to improve TCP level performance over wireless networks. We adapted this part of the solution in our interactive scheme as a way to avoid packet loss during handoff. Although this will slightly disrupt the service while handoff is being performed, but since we avoid packet loss, the

Table 17. IPMN API

System Call	Usage
Relay_IP(IP_addr)	Let TCP transmit a special segment carrying the new IP to the other end. Used in 'full interactive' mode only.
Switch_Source_IP(IP_addr)	Changes the source IP address in local TCP/IP stack.
Switch_Dest_IP(IP_addr)	Changes the destination IP address in local TCP/IP stack.
Freeze_TCP()	Let TCP freeze its transmission by advertising a zero window to the other end.
Resume_TCP()	Let TCP resume its transmission by advertising a non-zero window.

Table 18. Correspondent node locations

Name	Location	IP number	Average RTT	Hops from MN
Local	Kent, Ohio	131.123.36.11	1 ms	3
Virginia	Chantilly, Virginia	66.94.95.235	90 ms	19
Al-Quds	Palestine	62.90.25.58	356 ms	25

correspondent node will not resort to congestion control procedures avoiding unnecessary retransmissions and sender rate throttling. As we show later, this will definitely improve TCP performance and save network resources.

6.4 Experiment Setup and Traffic Generation

We have implemented the scheme on FreeBSD-4.5 by extending the kernel source code with InTraN components. In addition to that, we have created a number of system calls that implement the system's API shown in Table 17. These functions were used by the *TM*s as we described earlier in the IPMN architecture.

6.4.1 Experiment Setup

Figure 30 explains the experiment setup. We used three machines with AMD 1.6 GHz processor (BS1, BS2, and BS2) as our *Base Stations* and a laptop with Intel P-II processor as our mobile node. The (GW) machine was our gateway to the Internet and was also used to configure each one of the Base Stations as a separate subnet with four IP numbers per subnet. We installed FreeBSD-4.5 on all BS machines, the mobile node, and the correspondent node. For IPMN experiments we installed the BSD-interactive on the mobile node and the correspondent node only. For the MIP experiments we installed the MIP implementation of the Portland State University [Bin99]–also known as PSUMIP—on the mobile node and the three BS machines. One of the three Base Stations machine (BS1) was configured as the Home Agent (HA), and the other two (BS2 and BS3) were configured as Foreign Agents.

For MIP signaling to work correctly, the time must be synchronized on all machines which run the MIP daemons. To achieve this we used the (`ntpd`) utility in FreeBSD to synchronize with three STATUM-2 external time servers. We used the simplest possible MIP configuration to reduce unnecessary overhead. We placed the correspondent node in three locations, one locally (in our lab) and two remotely: at Al-Quds University in

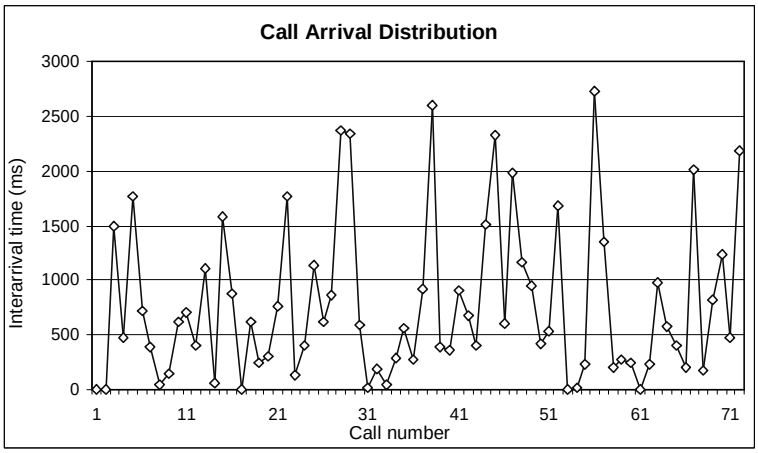

Figure 31. Sampling of call interarrival

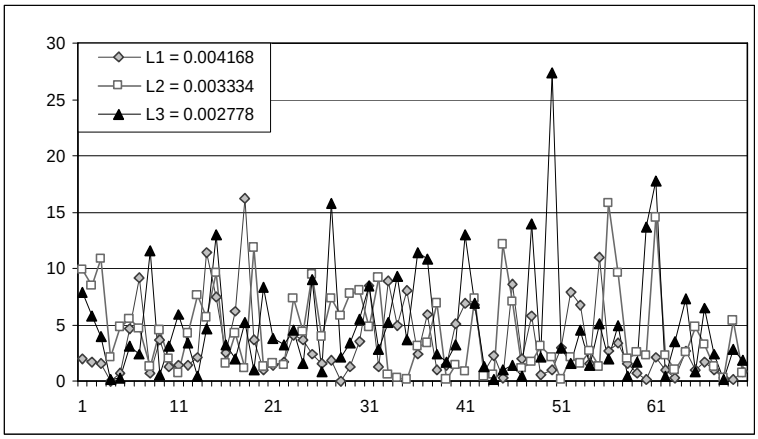

Figure 32. Sampling of call duration over 5 hours

Palestine, and in Virginia, U.S.A. Correspondent node locations and their characteristics are shown in Table 18. In each run, we let the correspondent node generate traffic and transmit to the mobile node. We also let the mobile node move along the cyclic path (BS1→BS2→BS3→BS2→BS1→…). We configured the mobile node to perform handoff every 3 minutes. We used a switch to simulate L2 wireless handoff; for example, in Figure 30 the mobile node is connected to BS1 through the switch. To perform L2 handoff from BS1 to BS2, we manually unplugged BS1 from the switch and instantly plugged BS2 to an empty port in the switch. We kept the mobile node connected to the switch all the time.

6.4.2 Traffic Characteristics

In order to model real-world traffic, we used a tool called *NetSpec* [Jon98] which was developed at The University of Kansas—to generate traffic at the correspondent node. Netspec offers several source models which can generate simulated traffic for Telnet, FTP, Video, Voice, and WWW [Lee95]. We ran the experiment with three types of traffic: Voice, WWW, and FTP. Below, we explain the statistical properties of these three traffic types.

A) Voice Traffic:

In NetSpec, voice has been characterized by a constant bit rate (CBR) source. Sampling rate is 8 kHz and each sample is 8 bits. This gives the standard bit rate of 64 Kb/sec for acceptable voice quality. Call arrivals are modeled by a Poisson process with fix hourly rates within one-hour periods. This means that the interarrival time between two calls is exponentially distributed. The probability density function of exponential distribution is given by:

$$f_X(x) = \lambda e^{-\lambda x} \qquad , \lambda = 1/mean$$

Session duration (holding time) for voice calls was also modeled by a Poisson process and followed the exponential distribution. Figure 31 shows an example of call arrivals with λ=1 over 5 hours sampling, and Figure 32 shows an example of call duration over 5 hour sampling with three values of λ: λ1=0.004167, λ2=0.003333, λ3=0.002777. If we take the inverse of these λs, we get mean call durations 3, 4, and 5 minutes respectively. At the call level, the source is presented to the network as a constant-bit stream. To generate a 64 Kb/sec voice stream, talk bursts were generated by a 144-byte blocks separated by 18 ms silence periods.

B) WWW Traffic:

WWW traffic is modeled at two levels: call level and session level. The call level models the interarrival times of multiple sessions. The session level on the other hand models the document size. Request arrivals are modeled by a homogenous Poisson process within one-hour intervals. The interarrival time between two requests is exponentially distributed. The distribution of document size is a Pareto distribution. The probability mass function of a Pareto distribution is:

$$f_X(x) = \alpha k^{\alpha} x^{-\alpha-1}$$

And its cumulative distribution function is given by:

$$F_X(x) = 1 - \left(\frac{k}{x}\right)^{\alpha} \qquad \begin{array}{l} 0.4 \le \alpha \le 0.63 \\ k \le 21 \end{array}$$

C) FTP Traffic:

Like www traffic, FTP traffic is modeled at two levels: call level and session level. The call level models the interarrival times of multiple sessions. The interarrival time between two FTP sessions is exponentially distributed. During a single FTP session multiple data items of varying sizes are transferred. NetSpec used a fixed distribution to model the number of items per session called (ftpNOfItems), and a fixed distribution to model items' sizes called (ftpItemSize). Both models were based on the log-normal distribution. For a detailed discussion on these distributions and other traffic types in NetSpec please refer to [Lee95].

For WWW traffic, we repeated the experiment with two choices of the interarrival parameter λ and shape parameter α; we combined λ_1=0.000001 and α_1=0.4 in one set of runs, and we combined λ_2=0.000005 with α_2=0.55 in another set. For Voice and FTP traffic we ran all experiments with the same two choices of the interarrival parameter λ. Since λ is the inverse of mean interarrival, λ_2 will yield longer interarrival intervals.

6.5 Performance Results and Analysis

6.5.1 Handoff Latency

One of the most important features of our interactive scheme is its short handoff latency. In Table 19 we show the handoff latency (in milliseconds) of the three traffic types (FTP, WWW, and Voice) on two cases of the correspondent node location (Local, and Virginia). The columns show the three running modes of the experiment (IPMN-Full, IPMN-Half, and Mobile IP). For each running mode we show two sub-columns (Protocol Latency, and Total Latency). The Protocol Latency column shows the time needed by the protocol (i.e. IMPN or MIP) to finish L3 handoff and become ready to resume its communication

Table 19. Handoff Latency

		IPMN-Full		IPMN-Half		Mobile IP		
		Protocol Latency	Total Latency	Protocol Latency	Total Latency	Protocol Latency	Total Latency	Difference
FTP	Local	91	98	517	520	40615	60017	19402
		102	109	543	552	52655	58841	6186
		94	103	541	545	34767	57067	22300
		89	94	540	550	37690	57966	20276
	Virginia	109	110	131	134	70934	185907	114973
		110	120	126	132	78928	130120	51192
		111	116	130	139	203575	204395	820
		109	113	129	135	64684	122791	58107
	Average	**101.88**	**107.88**	**332.13**	**338.38**	**72981**	**109638**	**36657**
WWW	Local	108	118	527	532	28687	29600	913
		90	91	544	551	53636	58148	4512
		88	93	521	524	74728	122712	47984
		92	96	539	547	54417	64867	10450
	Virginia	91	92	166	171	40533	54872	14339
		87	96	170	178	49610	57733	8123
		87	91	178	182	20564	59801	39237
		92	101	167	173	39057	61007	21950
	Average	**91.88**	**97.25**	**351.5**	**357.25**	**45154**	**63592**	**18438**
VOICE	Local	106	107	146	153	12654	90615	77961
		107	110	148	155	7124	87099	79975
		111	117	140	146	1524	70140	68616
		115	124	148	152	48945	154591	105646
	Virginia	114	121	124	134	58669	121124	62455
		106	114	139	144	24975	25763	788
		106	106	129	139	22672	25570	2898
		102	126	133	136	77414	125582	48168
	Average	**110.63**	**115.63**	**138.38**	**144.88**	**31747**	**87560**	**55813**

service. The Total Latency column shows the time needed to resume communication at the application level. We show the first four handoffs of each run. For example, for the (IPMN-Full/FTP/Local) run, in the first handoff the Protocol Latency time is 91 ms, and the Total Latency time is 98 ms. Also, for each traffic type, we add an extra row (the shaded one) to show the average of all eight runs of that traffic type.

We can make three observations on Table 19. Firstly, we observe a big difference in handoff latency between IPMN and MIP that can reach up to three orders of magnitude. For example, in (FTP/IPMN-Full) the average protocol latency is 101.88 milliseconds, while in (FTP/MIP) the average protocol latency is 72,981 milliseconds. This substantial reduction in handoff latency highlights the advantage of event-based protocols like IPMN over timer-based protocols like MIP. The former allows protocols in different layers to interact and pass events and new state information–like the new IP number in our case—to upper layers instantly. This enables peer protocols to respond immediately cutting down overhead time. Timer-based protocols on the other hand usually use a periodic probing mechanism to discover state changes. For example, in this particular implementation of MIP that we have tested, the foreign agent sends beacon signals (agent advertisements) to discover mobile node movement every 60 seconds! A best case scenario will happen if L2 handoff was performed right before the periodic beacon signal arrives. Therefore, this process will take half of that time on average–i.e. 30 seconds. Adding to this communication and registration overhead we can easily reach one minute latency or more. Secondly, actual application-level latency on MIP was even longer; by the time MIP has recovered and is ready to resume service, TCP has already timed out and will probably need even more time to discover the change and then resume communication on its own level. We show this quantity in the Total Latency column as we explained earlier. In IPMN-based protocols (Full and Half), the difference between these two quantities was very small (3 to 10 milliseconds) which can be regarded as negligible. Therefore, we only show this difference for the MIP case in the column labeled (Difference). A closer look at the Difference column, we see a great variation; it can be as low as 788 milliseconds, or as big as 114,974 milliseconds. This variation is due mainly to the dynamics of TCP congestion control and how it responds to long disconnections. But in

general, this Difference has really added yet another long delay (between 18.4 and 55.8 seconds on average) to the already high MIP handoff latency. Actually, the column Difference highlights another advantage of IPMN which takes into consideration TCP performance in addition to its main purpose as a mobility solution.

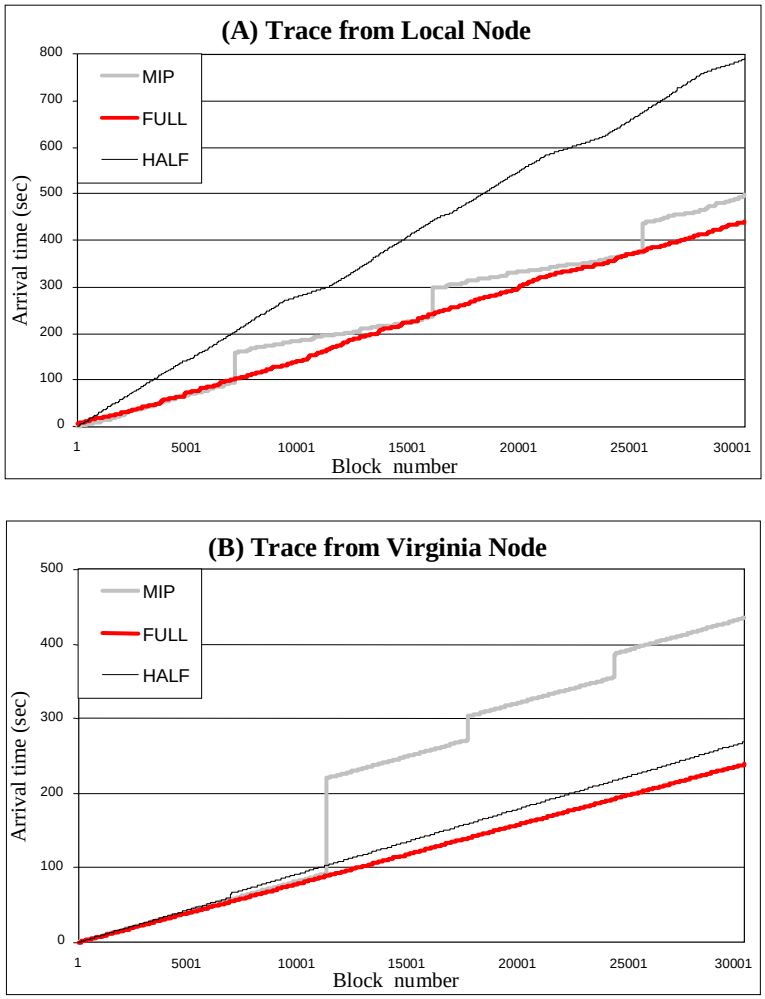

Figure 33. Voice stream arrival trace

The third observation is IPMN-Full superiority over IPMN-Half; IPMN-Full handoff took 91 to 110 milliseconds on average, while IPMN-Half handoff took 138 to 351 milliseconds on average. Again, this observation also emphasizes the benefit of interactivity. Recall that IPMN-Full employs interactivity on both endpoints while IPMN-Half uses interactivity on the mobile node only. As we can see this feature was to the advantage of IPMN-Full which managed to perform handoff in almost half the time needed by IPMN-Half.

6.5.2 Traffic Arrival Trace

A) Voice Stream Trace:

Here we show application level performance by observing voice stream arrival trace. At the MN, we kept a log file to register the arrival time of each 144-bytes block (talk burst) in the voice stream. Figure 33 plots the arrival times of the first 30,000 blocks at the MN from two of the correspondent nodes: Local and Virginia. The figure shows the case of interarrival parameter λ_1=0.000001. The λ_2=0.000005 case showed a similar trace but we did not include it for space limitation.

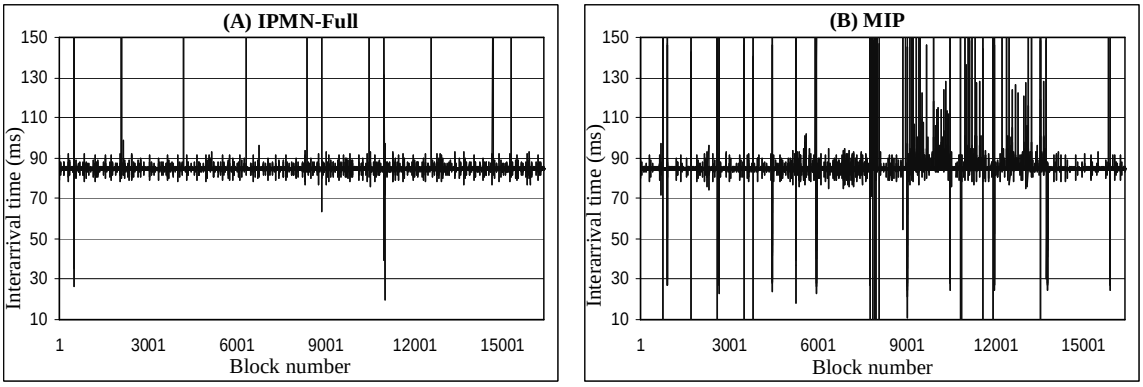

Figure 34. Block interarrival times at the MN (Jitter)

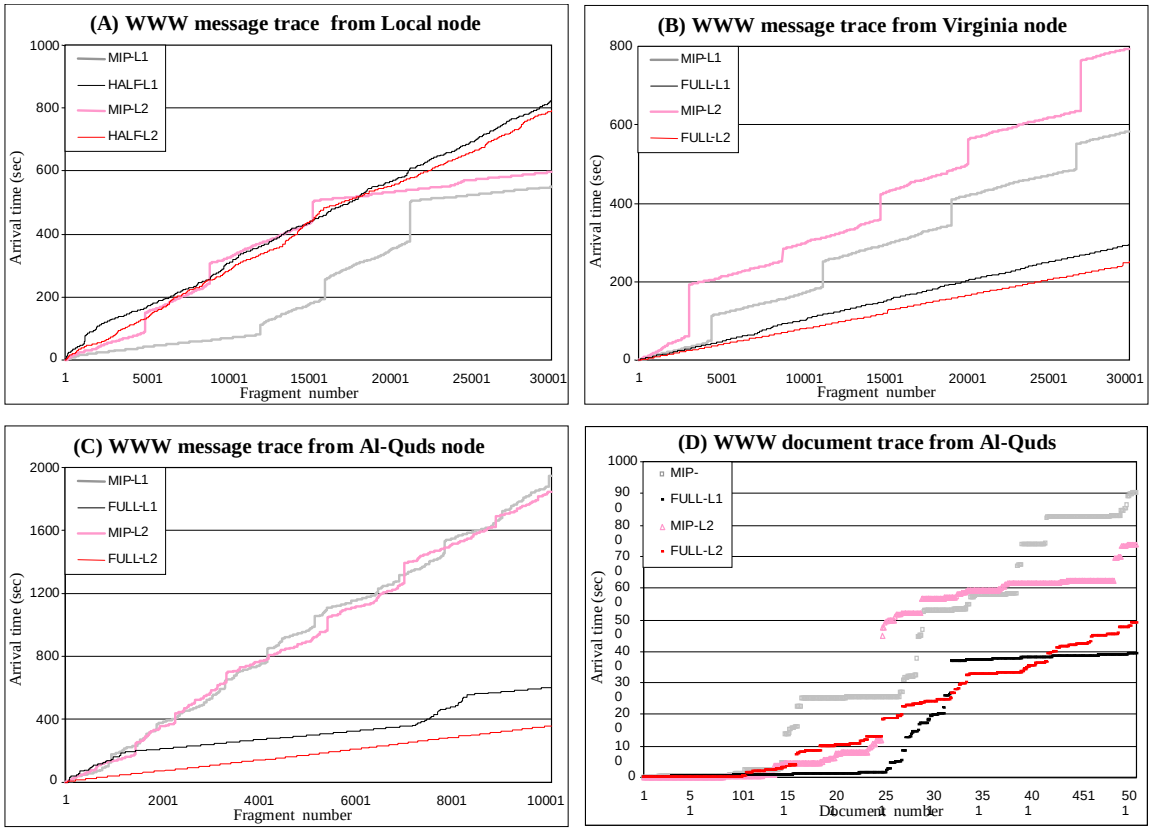

Figure 35. WWW traffic trace

We can make two observations on these plots; firstly, in the Local node plot, MIP actually outperformed IPMN-Half and almost tied with IPMN-Full. Maybe the only advantage of both IPMN schemes was the smoothness of the arrival trace—which is still important for voice traffic. This behavior can be explained by the fact that all nodes were in the same room. In such situation, only handoff latency can be seen as a performance metric, other issues will be dictated by TCP dynamics and LAN load. However, in real Internet scenario the two endpoints are usually far a part—as in case (B)—and there we can see the relevance of the interactive scheme. In Figure 33 (B), we see that both IPMN schemes outperformed MIP mainly due to the huge step jumps on the MIP trace. These step jumps and the impact of TCP dynamics

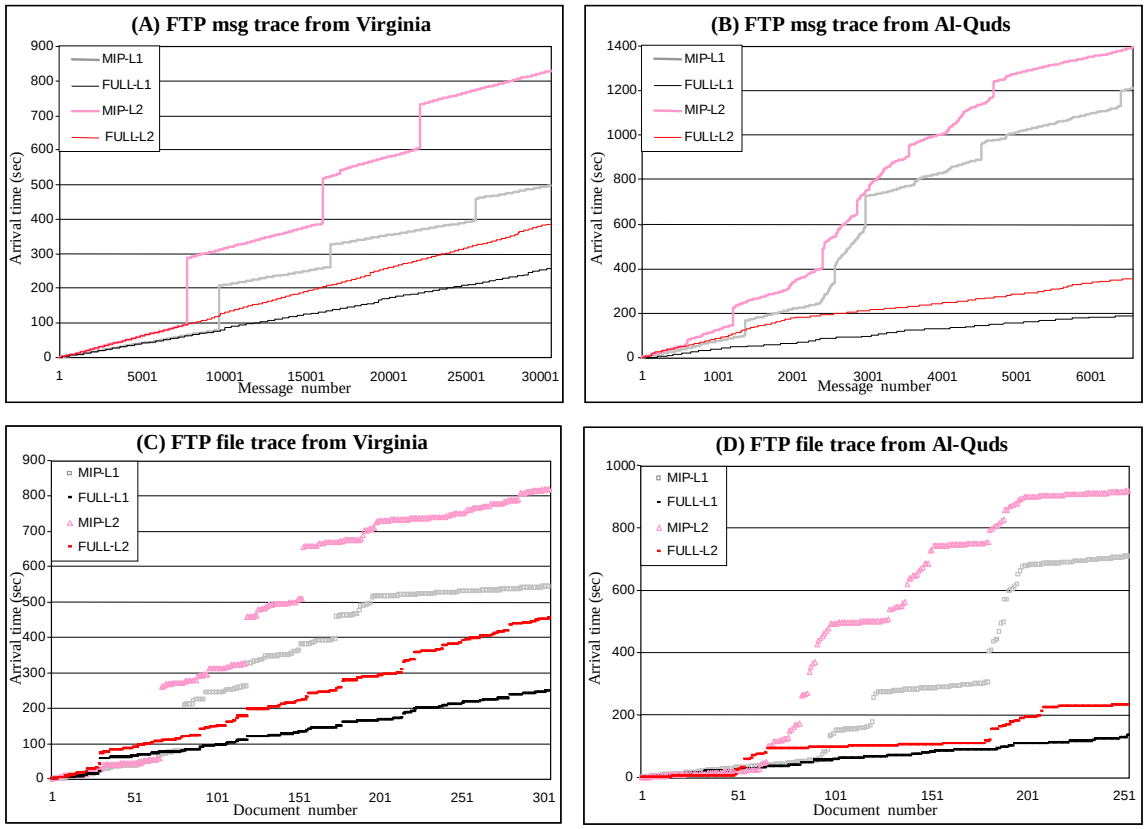

Figure 36. FTP traffic trace

created jitter on the voice stream as we will show in the next section. Also, it is worth noting that IPMN-Full was slightly more efficient than IPMN-Half.

B) Jitter on the Voice Stream:

Figure 34 plots the interarrival times of the first 16000 blocks arriving at the MN from Virginia node, (A) on IPMN-Full, and (B) on MIP. On IPMN-Full almost all blocks were delivered at (75 to 90) milliseconds apart, except (mainly) those that faced a handoff –only 22 blocks were delayed for more than 100 milliseconds. In Figure 34 we show a maximum of 150 ms on the y-axis to be able to see the mainstream case. Average interarrival time for all blocks on IPMN was 85.57 milliseconds. On MIP the situation is different; about 177 blocks in the stream faced more than 100 milliseconds interarrival –10 of these blocks faced more than 8000 milliseconds delay—and average interarrival time for all blocks was 129 milliseconds.

C) WWW Traffic Trace:

To trace WWW and FTP traffic, we kept two log files at the mobile node: one that registered the arrival time of each document and one that regarded the incoming stream as a whole sequence and registered the arrival time whenever 1 KB-fragment was received. Figure 35 (A) plots the arrival times in seconds of the first 30,000 fragments that arrived at the mobile node from the Local node, (B) traces the first 30,000 fragments from Virginia node, (C) traces the first 10,000 fragments from Al-Quds node, and (C) plots the arrival times of the first 500 documents that arrived from Al-Quds node—we show only this sample of documents' trace for space limitations. In each plot there are four runs: two for MIP and two for IPMN (all plots shows IPMN-Full except (A) which shows IPMN-Half). For each mode we generated WWW traffic using the two values of interarrival parameter λ that were mentioned earlier—shown in the figure as L1 and L2. Again, on the Local plot (A), MIP seems to outperform IPMN especially with L1 traffic. But on the Virginia plot (B), IPMN managed to deliver all fragments 2 – 3 times faster than MIP.

On the Al-Quds plots (C) and (D), the difference was even bigger. A closer look at the (C) plot reveals the impact of TCP's congestion control dynamics on the trace due to long disconnections on the MIP runs. This coarse behavior disappeared on the smoother IPMN traces. From this pattern of behavior, we can say that IPMN performs better when the two endpoints are furthest apart.

D) FTP Traffic Trace:

We show FTP traffic trace in Figure 36. Plot (A) of the figure shows the arrival times in seconds of the first 30,000 1 KB fragments that arrived at the mobile node from the Virginia node, plot (B) traces the first 6,500 fragments from Al-Quds node, plot (C) traces the first 300 files from Virginia node, and (D) plots the arrival times of the first 250 files that arrived from Al-Quds node. As in the WWW case, there are four runs in each plot: two for MIP and two for IPMN with the same interarrival parameters L1 and L2 that were used before. These plots further confirm the obvious advantage of IPMN over MIP as we saw with Voice and WWW traffic. It is worth noting the great impact of the long handoff disconnections on the trace of MIP cases in Al-Quds plots (B) and (D).

6.6 Conclusion

IPMN uses true end-to-end signaling to update the current state of the mobile node's location at both end-points. Using interactivity, the mobile node was able to freeze the TCP connection and to perform loss-free, rapid handoff by simply changing the 'source IP' field in TCP/IP stack of the mobile node and the 'destination IP' field in the TCP/IP stack of the correspondent node. As a mobility solution on the IP level, IPMN offered two key advantages over conventional timer-based MIP; (a) it allowed direct end-to-end communication between the correspondent node and the mobile node at a very little overhead cost, and (b) it dramatically reduced handoff latency by canceling movement detection and address registration. On the TCP level, IPMN managed to significantly improve TCP performance by the successful employment of the Freeze TCP technique in the *InTraN* paradigm. We have demonstrated these features of IPMN by real experimentation with Voice, WWW, and FTP traffic on remote nodes over the Internet. The results demonstrate the benefit of the principle of interactivity in networking. It enables event based action and response. It distinguishes from the traditional timer-based MIP which depends on periodic actions. The periodic agent advertisements used in MIP is one of the prime reasons for its sluggishness. MIP has to maintain a delicate balance between advertisements' frequency/size and their impact on network throughput[1]. Event-based scheme such as the one demonstrated by IPMN does not require this compromise. Indeed the benefit of instant interactivity was so dramatic that it could easily wipeout the seeming advantage of MIP's low layer implementation.

[1] The original MIP proposal [Per96] recommended shortest agent advertisement rate of 1 per second. The implementation that we have tested in this paper (PSUMIP) uses a much slower rate of 1 per minute. We tried to lower this rate, but it did not work. Since PSUMIP was the only available implementation compatible with FreeBSD-4.5 kernel at the time, we could not test with faster agent advertisement rate. Many other MIP implementations allow the user to set a preferred rate of one or more seconds. The best rate that would yield optimal network throughput is still controversial and is highly dependent on mobile node's movement frequency and traffic load.

CHAPTER 7

Protocol Modeling

In this chapter we briefly describe two well-knows protocols; Snoop [Bal95] by Balakrishnan et al., and WTCP [SiV99] by Sinha et al. They are among many other schemes proposed in the literature to improve TCP performance over wireless links. Then, we show how they can be modeled with the meta-engineering of the *InTraN* paradigm.

Wireless networks have certain characteristics that are not handled properly by regular TCP such as high bit error rate (BER) and long disconnections due to handoffs or bad reception. When a packet is lost, regular TCP assumes that it is due to congestion and will always trigger congestion control procedures at the fixed host. However, in a wireless environment, radio transmission errors or handoffs can also cause packet loss. This will result in significant reductions in throughput that can severely degrade overall performance. A good survey on proposed protocols for improving TCP performance over wireless networks can be found in [Anj03] [Bal96] [Ela02].

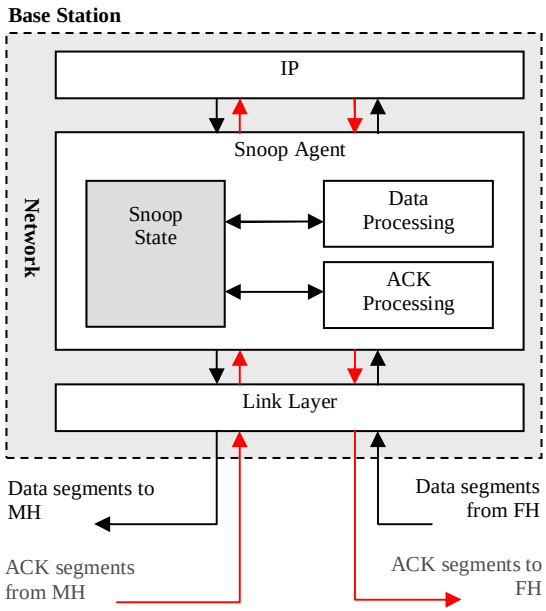

Figure 37. Conventional Snoop mechanism

7.1 Snoop

The Snoop protocol introduced a module, called Snoop, at the base station that monitors every packet that passes through in both directions. The Snoop module maintains a cache of TCP packets sent from the fixed host that have not yet been acknowledged by the mobile host. A packet loss is detected either by the arrival of duplicate acknowledgment or by a local timeout. To implement the local timeout, the module employs its own retransmission timer. The Snoop module retransmits the lost packet if it has it in the cache. Thus, the base station hides the packet loss from the fixed host, therefore avoiding its invocation of an unnecessary congestion control mechanism. Figure 37 describers the basic architecture of the classic Snoop protocol and Figure 38 shows the *InTraN*-enabled model of Snoop. The scheme represents part of the snoop protocol that handles one direction of the traffic only (Data segments from FH to MH and ACK segments from MH to FH). The Snoop protocol uses a different technique to handle traffic on the other

direction, but it can be easily modeled with the *InTraN* framework in a similar fashion. The model shown in Figure 38, assumes that data segments are cached in the network as in conventional Snoop for performance reasons.

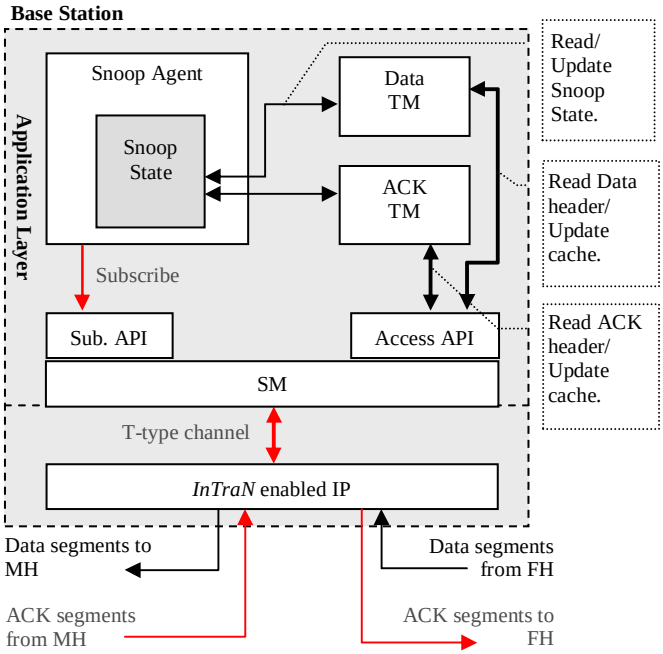

Figure 38. The interactive version of Snoop (iSnoop)

Assuming that the 'Snoop Agent' shown in Figure 38 has subscribed with the *InTraN*-enabled IP protocol (or iIP) for two events: an ACK received from MH event (evt_ACK_MH), and data segment received from FH event (evt_DAT_FH). Whenever any one of these two events occurs, iIP sends a signal to the SM which invokes the appropriate *TM*: *TM-Data* or *TM-ACK*. The *Snoop Agent* is a process that runs in the application layer. Its main role is to initialize and maintain the *Snoop State*, subscribe with the InTraN service. Afterwards, most of the work is done by the *TM*s. The Snoop State is similar to the one used in the conventional snoop protocol. The *TM-Data* handles the (*Data segment received*) event. It implements the *Data processing* algorithm of the snoop protocol. The *TM-ACK* handles the (*ACK segment received*) event. It implements the *ACK processing* algorithm of the snoop protocol. Both algorithm are describes in detail in [Bal95]. Both *TM-Data* and *TM-ACK* need to interact with iIP; they use the Access API of the *InTraN* service to (i) probe the IP layer and *Read* relevant header parameters from the TCP segment that has just arrived and (ii) to update the cache of TCP segments. The *TM-Data* adds segments to the cache and the *TM-ACK* clears the cache or part of it as decided by their respective algorithms. We assume that both *TM*s have full access to the Snoop State; they can read and update state variables as necessary.

7.2 WTCP

Wireless Transmission Control Protocol (WTCP) is specifically designed for wireless wide area networks. WTCP is based on the following two key principles: (i) it uses rate-based rather than window-based transmission control, i.e., it does not use ACKs for self clocking, and (ii) it uses the ratio of the inter-packet separation at the receiver and the inter-packet separation at the sender as the primary metric for rate control rather than using packet loss and retransmit timeouts. WTCP uses a heuristic based on the average per-packet separation to distinguish congestion losses from random losses. In this heuristic, the receiver initially predicts that all losses are non-congestion losses. The following example from the WTCP original paper [SiV99] explains the main concept of this heuristic: consider that packets i and j were received ($i < j$),

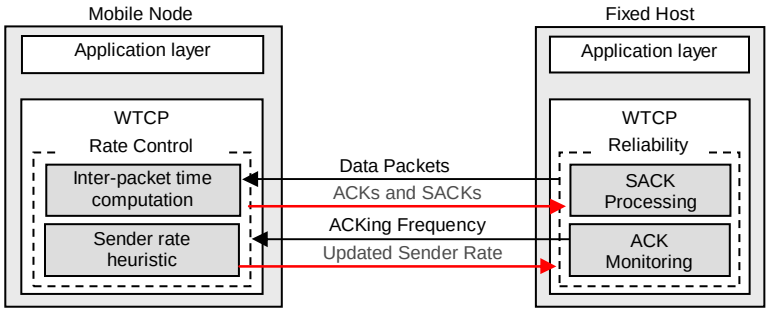

Figure 39. Conventional WTCP mechanism

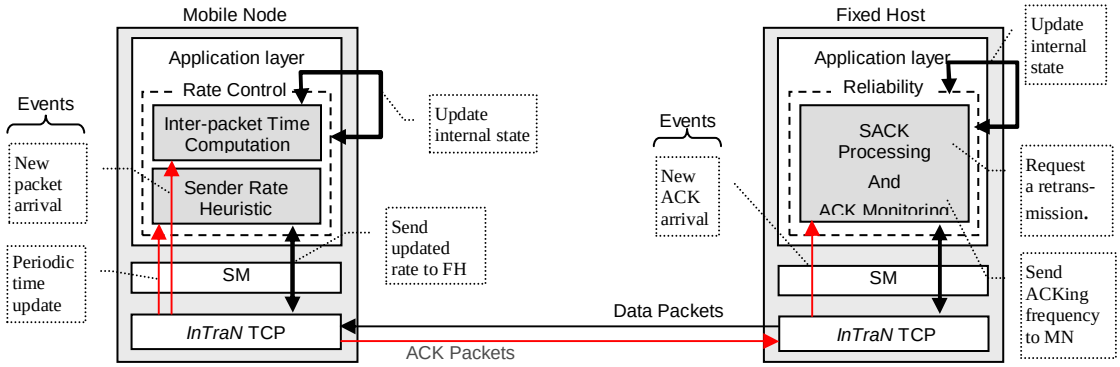

Figure 40. The interactive version of WTCP (iWTCP)

but packets $i+1 \ldots j-1$ were all lost. In this case the receiver computes the average inter-packet separation for each of the lost packets as:

$$perPktSep \leftarrow \frac{recvTime_j - recvTime_i}{j - i}$$

Where $recvTime_i$ is the time at which the last bit of packet i arrive. If the value of *perPktSep* is close to the measured inter-packet separation at the receiver (i.e., within the band [average − K × mean deviation, average + K × mean deviation], where K is a constant), then the receiver predicts that the losses were all random losses. Otherwise, the receiver predicts that there was at least one congestion loss, and the sending rate is reduced. The basic mechanism of the WTCP's rate-based scheme is shown in Figure 39. The receiver computes the desired sending rate via its rate control mechanisms, and notifies this rate to the sender in the ACK packets. ACKs, thus, carry both reliability information (SACK) and rate control information. The sender monitors the reception of ACKs, and adjusts its rate accordingly. It also monitors the ACKs to tune the ACKing frequency, which it then notifies to the receiver in future data packets. We show the *InTraN*-enabled model of WTCP in Figure 40. Basically, we have moved most of the processing to the application layer as *TM*s; i.e., the rate control algorithm at the receiver (MN) and reliability algorithm at the sender (FH). The *InTraN* extension provided the necessary API that allows TCP to trap events on both ends. On the MN, when a new packet is received, this event triggers the (inter-packet time computation) *TM*, which calculates new timers and updates the internal state of WTCP. When it is time to perform the periodic update, this event triggers the (sender rate heuristic) *TM* to calculate a new rate for the sender. The updated rate is transmitted to the sender through the API. On the sender side, when an ACK packet is received, one *TM* handles (ACK monitoring) and (SACK processing) since the ACK packet carries both ACK and SACK information. The (SACK processing) part of the *TM* discovers holes in the transmitted packet sequence, i.e., discovers lost packets, and issues retransmission request through the *InTraN* API. The (ACK monitoring) part of the *TM* calculates a new ACKing frequency rate based on the current transmission rate and the internal state and sends the updated rate to the receiver periodically.

Table 20. Cost parameters for Snoop and iSnoop

Name	Meaning
C_{ACK}	Overhead cost per ACK segment
C_{DAT}	Overhead cost per Data segment
N_{ACK}	Number of ACK segments
N_{DAT}	Number of Data segments
U_n	Update State/Cache cost in normal mode
U_i	Update State/Cache cost in interactive mode. We assume that $U_i > U_n$ since U_n might involve making a system call.
Sub	Subscription cost
S	Software Interrupt '*Signal*' cost
H	Signal Handler cost
R	Retransmit cost
T	Total transfer size (Mbytes)
C_{hoff}	Handoff cost
R	Wireless link bit rate (Mbps)

7.3 Performance Issues

7.3.1 Overhead Cost

The transparency model implementation of both protocols adds some extra cost to the original scheme as a result of the added signaling and system calls overhead. Here, we show an abstract comparison of both interactive and conventional schemes of the Snoop protocol. In Table 20 we show several quantities that

Table 21. Algebraic overhead cost of Snoop and iSnoop

Scenario	Classic Snoop	Interactive Snoop (iSnoop)
Error-free, handoff-free wireless link	$SNOOP_{free}$ = N_{DAT} $(C_{DAT} + U_n)$ + N_{ACK} $(C_{ACK} + U_n)$	$iSNOOP_{free}$ = $Sub + N_{DAT}$ $(S + H + C_{DAT} + U_i)$ + N_{ACK} $(S + H + C_{ACK} + U_i)$
Error-prone link with BER_x = 1 error / *x* MB	$SNOOP_{free} + (T / BER_x)$	$iSNOOP_{free} + (T / BER_x)$
Handoff every *n* seconds	$SNOOP_{free} + C_{hoff}$ $(8 \cdot T / n \cdot R)$	$iSNOOP_{free} + C_{hoff}$ $(8 \cdot T / n \cdot R)$

Table 22. Running modes for the `getrusage()` experiment

Mode name		Description
Classic TCP		No interactivity overhead. This is the reference case.
iTCP modes	Invoke only	Subscribe with a *Signal-only* type *TM*. The *TM* does not perform any Read/Write operations.
	File access	Subscribe with a *Signal-only* type *TM*. We let the *TM* open a disk file and perform one read operation and one write operation.
	Protocol access	Subscribe with a *Read-only* type *TM*. We let the *TM* perform one ReadVar() operation from TCP.
	Protocol & File	Subscribe with a *Read-only* type *TM*. We let the *TM* perform both a disk read/write and a ReadVar() operation from TCP.

define cost variables and wireless link characteristics. The first column in Table 21 shows the estimated cost incurred by deploying the Snoop protocol for three wireless link scenarios: (1) error-free, handoff-free wireless link, (2) error-prone link with BER = 1 error for each *x* Mbytes, and (3) a moving mobile node that triggers a handoff every *n* seconds. The second column represents the *InTraN* version of Snoop. In the first scenario (a reference case) iSnoop added overhead came from *Sub*, *S*, *H*, and U_i - U_n. Actually, in real practice these added costs should be very small (almost negligible). For example *Sub*, *H*, and U_i all involve

Table 23. CPU time

	User CPU Time		System CPU Time		Total CPU Time	
	iTCP%	SD	iTCP%	SD	iTCP%	SD
Invoke only	1.10%	55.68	3.80%	67.62	2.53%	200.08
File access	2.70%	116.47	3.70%	106.94	3.23%	272.28
Protocol access	0.90%	44.65	3.10%	59.37	2.07%	167.89
Protocol & File	1.30%	81.09	4.10%	77.95	2.82%	224.89

Table 24. iTCP context switching overhead

	Voluntarily CSW	Forced CSW	Total CSW
Invoke only	24.10%	0.19%	4.16%
File access	25.10%	0.22%	4.39%
Protocol access	24.30%	0.20%	4.22%
Protocol & File	24.20%	0.20%	4.19%

running a small system call and OS context switch cost. Besides the reference case, the other two scenarios were identical in both protocols. The same kind of analysis holds for the WTCP case. To get a real measurement of interactivity service overhead we performed a simple experiment on iTCP. We ran the video session (server, transcoder, and player) on classical TCP (the reference case) and on iTCP with four different modes by varying the access complexity of the *TM*. These five modes are explained in Table 22. We used the FreeBSD utility getrusage() to collect statistics about system resources used by the video transcoder (our subscriber program) in the five running modes. In the four iTCP modes, we measured the overhead cost of invoking the *InTraN* service which can be summarized by (1) subscription cost, (2) *SM* cost, and (3) *TM* cost. The most significant part of these is the *TM* cost since it implements the real protocol extension and its complexity can vary significantly. Therefore, we used *TM* complexity as a criterion to classify iTCP runs into four modes. Also, in each mode, we ran the video session ten times by varying the number of *TM*s that were invoked during the session from 1 to 10—we will call this number *N*. We collected the following resource usage information from the getrusage() function:

1) *utime*: The total amount of time spent executing in user mode.
2) *stime*: The total amount of time spent in the system executing on behalf of the process.
3) *vcsw*: The number of times a context switch resulted due to a process voluntarily giving up the CPU before its time slice was completed (usually to await availability of a resource).
4) *fcsw*: The number of times a context switch was forced by the OS due to a higher priority process gaining the CPU or because the current process exceeded its time slice.

The performance results of the first two parameters are plotted in Figure 41 and the latter two are plotted in Figure 42.

A) CPU time Analysis

In Figure 41 iTCP overhead time is shown on the left Y-axis at the lower part of the figure, and the total application running time is plotted on the right Y-axis. Here we see that (*utime*) overhead varied between 0 and 220 msec, while (*stime*) overhead varied between 0 and 360 msec. This is a small percentage of the total running time in both cases as we show in Table 23. In the table we also show the standard deviation of the iTCP overhead over the 10 runs. Also, we could not determine a consistent pattern of CPU time overhead as *N* increases. This means that once the *InTraN* service is deployed in the system, *N* will not have a significant impact on CPU time. But it can be seen that iTCP modes which involve a file access took more CPU time—which is reasonable.

B) Context Switching Analysis

In Figure 42 we show context switching overhead and in Table 24 we show the related statistics. It can be seen that approximately 20% of the total number context switching was voluntarily (*vcsw*) and the rest was forced (*fcsw*). But, iTCP added more to (*vcsw*)—between 1000 to 4400 context switches—than

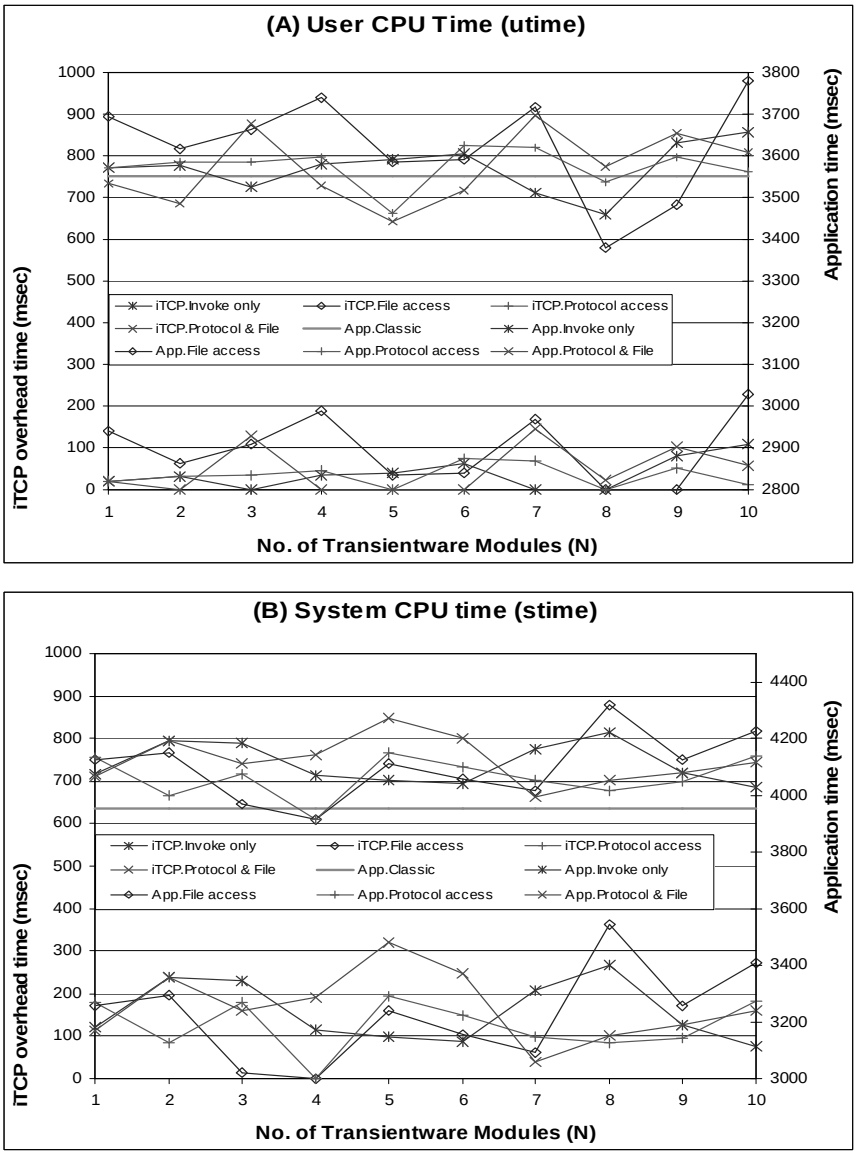

Figure 41. iTCP CPU time overhead

that it added to (*fcsw*)—between 70 to 170 context switches. Percentage wise, as shown in table Table 24, iTCP overhead is 25% of (*vcsw*) versus 0.22% of (*fcsw*). Overall, iTCP added less than 4.5% to the total context switching. Another observation is the increase pattern of (*vcsw*) as a linear function of *N* which can be described by $f = 252\ N + 1500$. This means that iTCP service deployment will add at least 1500 to (*vcsw*), and then (*vcsw*) grows linearly with a slope = 252 as *N* increases.

7.3.2 Security and Practice

The added small overhead cost can be justified for many practical gains allowed by the *InTraN* paradigm. As we mentioned earlier, since *TM*s run in the application space, they will enjoy a well developed provision tuned to run custom codes, share resources, and manage security issues. Actually, the security issue is of great importance in such engagement. Running the Active modules inside the network raises many security concerns that usually require complex techniques to maintain acceptable security level and stability within the network domain. Moving these modules up to the application layer makes security management a much easier task. Actually, Subscriber Programs and *TM*s can only access internal network

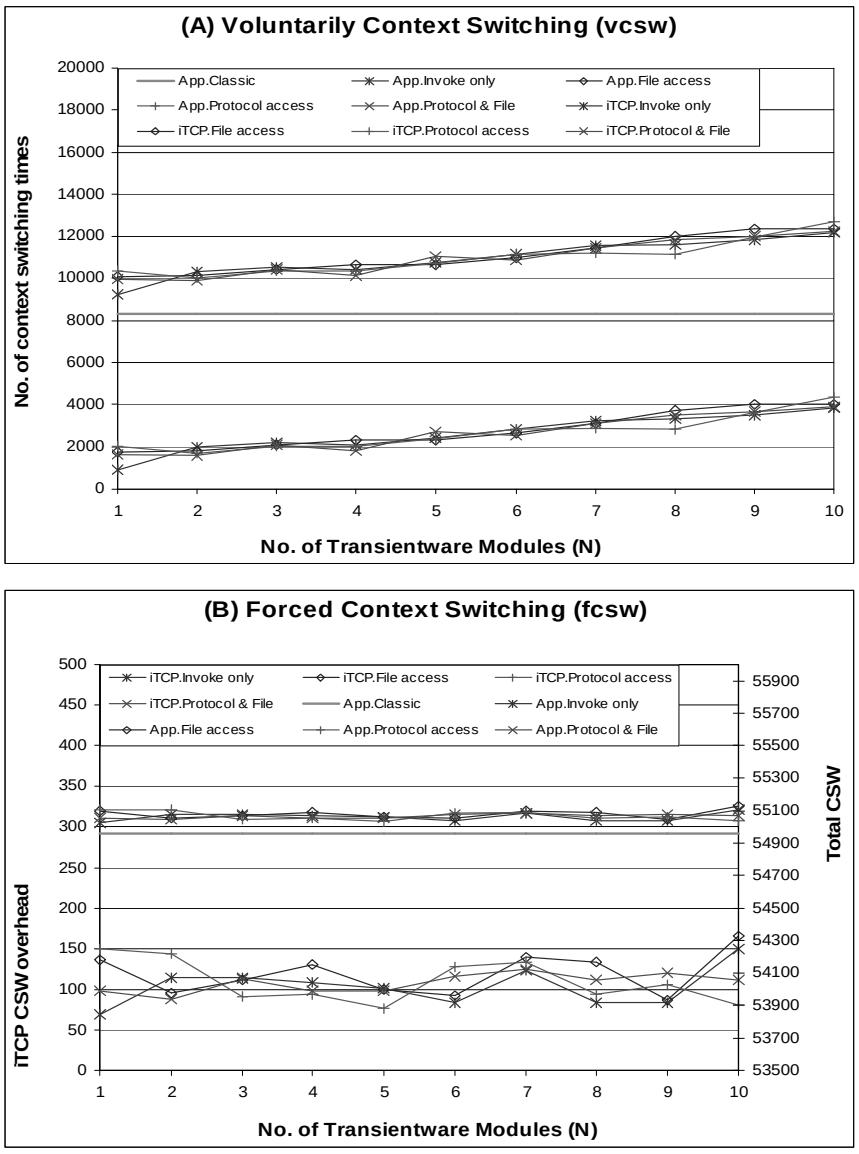

Figure 42. iTCP context switching overhead

services through the API extension. Therefore, by imposing the appropriate access restrictions on each party, we can guarantee certain security level. Furthermore, since the API extensions can be implemented as system calls, we can simply extend the OS security model and reuse available OS facilities like memory management and resource sharing to achieve even better performance. These characteristics make the *InTraN* model an attractive and a practical choice to implement and deploy many useful protocols which thus far had been only simulated or tested on a small-scale controlled testbed.

7.4 Conclusion

In this chapter we have shown that the Interactive Transparent Networking (*InTraN*) paradigm can be used to re-model existing protocol modifications by protocol meta-engineering and application level Transientware Modules (*TM*s). Actually, the *InTraN* version of the re-modeled solution can be further enhanced without changing the lower level implementation. For example, a protocol like WTCP which was intended to improve TCP performance over wireless links can also be augmented with extra *TM*s to add TCP-friendly features. We have particularly chosen two 'original source' examples for demonstrating an

implementation path via transparent networking—but this is not to endorse them. Please note because of their basic usefulness researchers have subsequently proposed several improved variants [Anj03] [Ela02]. The proposed transparency via interaction and triggered *TM* deployment will provide them implementation paths as well. In fact, since *TM*s operate at the application layer it will be much easier even to upgrade a certain *TM* from its current version to another improved one.

CHAPTER 8

Conclusion

In this dissertation we have presented and investigated *InTraN*, a new paradigm for extensible and adaptive networking that can dynamically meet the emerging requirements of modern networking, and the increasing demands of distributed applications (e.g., QOS guarantees, security, mobility, fault-tolerance, etc). The *InTraN* paradigm retains a good balance between the classical Internet design principles and the more contemporary aggressive approaches like programmable networks and protocol composition tools. On one hand, *InTraN* maintains the benefits of the classical design principles by (i) keeping the core of the network as simple and generic as possible, and (ii) maintaining the original semantics of the network layers. On the other hand, it can still achieve the goals of contemporary approaches for extensible and adaptive networking.

The *InTraN* paradigm has a number of unique features that distinguish it from other approaches; first, it offers an event/response mechanism (vs. timer-based or probing-based mechanisms) which makes it faster and more responsive especially for time-critical applications. Secondly, it requires only a light-weight re-organization of the existing kernel infrastructure (vs. heavily customized modifications or complex middleware) which adds minimal overhead to the network and tolerates high scalability. Thirdly, it allows kernel level enhancements to be performed at the application level, and thus, they become much practical/easier to deploy and implement (as opposed to direct kernel modifications). Furthermore, this feature can greatly simplify other critical issues like security management, maintenance, and resource sharing since the application layer has been optimized to handle these issues effectively. We have illustrated these principles by presenting three types of solutions based on the proposed *InTraN* paradigm; (1) adaptive applications via a video transcoder, (2) cross-layer optimization via a mobility solution for IP networks, and (3) protocol extensions. The experimental results reported in this work—from real prototype implementation and world-wide Internet experiments—have shown substantial improvements in network level performance as well as application level performance.

A number of issues are yet to be investigated though; for example, we have proposed a security model for *InTraN*, but we have not implemented that in the prototype, and even though we know—theoretically at least—that the proposed security model should be effective in maintaining the safety and correctness of the overall system, a real prototype may be needed to expose its real capabilities and complexity. Another unresolved issue is scalability; the *SM* (Subscription Manager) has been designed as a central handler, and thus, it might become a bottleneck during heavy traffic or with an increasing number of transientware modules. In this current prototype we have used one *SM* to handle all modules/applications. Another alternative approach is to use one *SM* per application instance. Therefore, a future research can reveal which approach is more effective, and to which extent it can scale-up to support bigger, more complicated scenarios. Finally, this current design of *InTraN* does not offer any guarantees that event signals will be handled in the same order by which they have occurred. However, in a more sophisticated scenario (e.g., multicasting) where potentially many signals might arrive at the *SM* simultaneously, the issue of event synchronization and timing should be addressed. This can also be picked up as a future research topic.

References

[ABone] ABone, Active Network Backbone, http://www.isi.edu/abone/

[Alex98] Alexander, D.S., Arbaugh, W.A., Hicks, M.A., Kakkar P., Keromytis A., Moore J.T., Nettles S.M., and Smith J.M., "The SwitchWare Active Network Architecture", IEEE Network Special Issue on Active and Controllable Networks, vol. 12 no. 3, 1998.

[Alm99] Almes G., Kalidindi S., and Zekauskas M., "A one-way packet loss metric for IPPM," RFC2680, 1999.

[And00] Andersen D., Bansal D., Curtis D., Seshan S., and Balakrishnan H., "System Support for Bandwidth Management and Content Adaptation in Internet Applications," Proc. of OSDI'00, Oct. 2000, San Diego, CA.

[Anj03] Anjum F., and Tassiulas L., "Comparative Study of Various TCP Versions Over a Wireless Link With Correlated Losses," IEEE/ACM Transactions On Networking, Vol. 11, No. 3, June 2003.

[Aya95] Ayanoglu E., Paul S., LaPorta T. F., Sabnani K., and Gitlin R., "AIRMAIL: A Link-Layer Protocol For Wireless Networks," Wireless Networks Vol. 1, pp. 47-60, 1995.

[Bakr97] Bakre A., and Bardinath B. R., "Implementation and Performance Evaluation of Indirect TCP," IEEE Transactions on Computers, Vol. 46, No. 3, pp. 260-278, 1997.

[Baks97] Bakshi B., Krishna P., Vaidya N. H., and Pradhan D. K., Improving Performance of TCP Over Wireless Networks," 17th International Conference on Distributed Computing Systems, pp. 365-373, 1997.

[Bal95] Balakrishnan H., Seshan S., and Katz R., "Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks," ACM Wireless Networks, Vol. 1, No. 4, pp. 469-481, 1995.

[Bal96] Balakrishnan H., Padmanabhan V., Seshan S., and Katz R., "A comparison of mechanisms for improving TCP performance in wireless networks," ACM SIGCOMM Symposium on Communication, Architectures and Protocols, Aug. 1996.

[Bal99] Balakrishnan H., Rahul H., and Seshan S., "An Integrated Congestion Management Architecture for Internet Hosts," Proc. of ACM SIGCOMM, Cambridge, MA, Sep 1999. pp.175-187.

[Ban02] Ban B., JavaGroups 2.0 User's Guide, Nov 2002.

[Ber02a] Berson S., Branden B., and Dawson S., "Evolution of an Active Networks Testbed," Proceedings of the DARPA ActiveNetworks Conference and Exposition 2002, pp. 446-465, San Francisco, CA, 29-30 May 2002.

[Ber02b] Berson S., Branden B., and Ricciulli L., "Introduction to the ABone," Feb. 2002, available at http://www.isi.edu/abone/DOCUMENTS/ABarch/

[Bha98] Bhatti N., Hiltunen M., Schlichting R., and Chiu W., "Coyote: A system for constructing fine-grain configurable communication services," ACM Trans. On Computer Systems, 16 (4), pp 321–366, November 1998.

[Bhat96] Bhatti N. T., "A system for constructing configurable high level protocols," PhD thesis, University of Arizona, 1996.

[Bin99] Binkley J., and Singh S., The Portland State University Secure Mobile Networking Project (PSUMIP), http://www.cs.pdx.edu/research/SMN/, 1999.

[Bir87] Birman K., and Joseph T., "Reliable Communication in the Presence of Failures," ACM Transactions on Computer Systems, Vol 5, No 1, pp. 47–76, Feb. 1987.

[Blu01] Blumenthal M., and Clark D.D., "Rethinking the design of the Internet: the end-to-end arguments vs. the brave new world," ACM Transactions on Internet Technology (TOIT), Vol. 1 ,no. 1, pp. 70 – 109, August 2001.

[Boc79] Bochmann G. V., and Vogt F.H., "Message link protocol (MLP): functional specification," ACM SIGCOMM Computer Communication Review, Vol. 9, Issue 2, pp. 7-39, 1979.

[Bri99] Briceño H., S. Gortler and L. McMillan, "NAIVE--network aware Internet video encoding," Proc. of the 7th ACM International Conference on Multimedia, Oct. 1999, Orlando, FL, pp. 251-260.

[Byu01] Byun Y., Sanders B., and Keum C-S., "Design Patterns of Communicating Extended Finite State Machines in SDL," 8th Conference on Pattern Languages of Programs (PLoP'01), 2001.

[Cac99] Caceres R., Duffield N.G., Horowitz J., Towsley D.F., and Bu. T., "Multicast-based inference of network-internal characteristics: Accuracy of packet loss estimation," Proc. of IEEE INFOCOM'99, pp. 371–379, 1999.

[Calv98] Calvert, K. et al, "Architectural Framework for Active Networks", Active Networks Working Group Draft, July 1998.

[Cam01] Campbell A., et al., "IP Micro-Mobility Protocols," ACM SIGMOBILE Mobile Computer and Communication Review (MC2R), Vol. 4, No . 4, pp. 45–54, October 2001.

[Cam99] Campbell A., Meer H., Kounavis M., Miki K., Vicente J., and Villela D., "A Survey of Programmable Networks," ACM Computer Communications Review, Vol. 29, No. 2, pp. 7-23, April 1999.

[CANE] "CANEs: Composable Active Network Elements", http://www.cc.gatech.edu/projects/canes/

[Chan96] Chan, M.-C., Huard, J.-F., Lazar, A.A., and Lim, K.-S., "On Realizing a Broadband Kernel for Multimedia Networks", 3rd COST 237 Workshop on Multimedia Telecommunications and Applications, Barcelona, Spain, November 25-27, 1996.

[Che96] Cheng K-T., Krishnakumar A., "Automatic generation of functional vectors using the extended finite state machine model," ACM Transactions on Design Automation of Electronic Systems (TODAES) Volume 1, Issue 1, pp. 57-79, Jan 1996.

[Dol96] Dolev D., and Malki D., "The Transis approach to high availability cluster communication," Communications of the ACM, Vol 39, No 4, pp 64–70, 1996.

[Ela02] Elaarag H., "Improving TCP Performance over Mobile Networks," ACM Computing Surveys, Vol. 34, No. 3, Sep. 2002, pp. 357–374.

[Ell97] Ellsberger J., Hogrefe D., and Sarma A., "SDL: Formal Object-Oriented Language For Communicating Systems," Prentrice Hall, Harlow, England, 1997.

[Fik01] Fikouras N., Könsgen A., and Görg C., "Accelerating Mobile IP Hand-offs through Link-layer Information," in Proc. of the International Multi-conference on Measurement, Modeling, and Evaluation of Computer-Communication Systems (MMB), Aachen, Germany, September 2001.

[Fik99] Fikouras N., El Malki K., and Cvetkovic S., "Performance Evaluation of TCP over Mobile IP," In Proc. of the International Symposium on Personal Indoor and Mobile Radio Communications 1999 (PIMRC), Osaka, Japan, September 1999.

[Gof00] Goff T., Moronski J., Phatak D., Gupta V., "Freeze-TCP: A True End-to-End TCP Enhancement Mechanism for Mobile Environments," INFOCOM'00, Tel-Aviv, Israel, pp. 1537-1545, 2000.

[Gua04] Guan Sheng-Uei and Lim Sok-Seng, "Modeling adaptable multimedia and self-modifying protocol execution," Future Generation Computer Systems, Vol. 20, No. 1, pp. 123-143, Jan

2004.

[Gus01] Gustafsson E., et al. “Mobile IPv4 Regional Registration” draft-ietf-mobileip-reg-tunnel-05, IETF, September 2001.

[Hau04] Huang Y-W., Yu F., Hang C., Tsai C-H., Lee D-T., Kuo S-Y., "Securing Web Application Code by Static Analysis and Runtime Protection," Proc. of the 13th int. conference on WWW (WWW2004), pp. 40-52, 2004.

[Hay98] Hayden M., "The Ensemble system," Technical Report TR98-1662, Department of Computer Science, Cornell University, Jan. 8, 1998.

[Hil98] Hiltunen M. A., and Schlichting R. D., "A configurable membership service," IEEE Transactions on Computers, Vol 47, No 5, pp. 573–586, 1998.

[Hri02] Hristea C., and Tobagi F., “A network infrastructure for IP mobility support in metropolitan areas,” Computer Networks, 38, pp.181-206, 2002.

[Hut91] Hutchinson N. C., and Peterson L. L., "The x-Kernel: An Architecture for Implementing Network Protocols," IEEE Transactions on Software Engineering, Vol. 17, No. 1, pp. 64–76, Jan. 1991.

[Jac88] Jacobson V., “Congestion Avoidance and Control,” Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988.

[Jac90] Jacobson V., “Modified TCP Congestion Avoidance Algorithm,” end2end-interest mailing list, April 1990.

[Jon98] Jonkman R., Evans J., and Frost V., "Netspec: A Tool for Network Experimentation and Measurement", University of Kansas, 1998. http://www.ittc.ku.edu/netspec/

[Ke00] Ke J. and Williamson C., “Towards a Rate-Based TCP Protocol for the Web,” Proc. of the 8th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecomm. Systems, 2000.

[Kha01] Khan J. and Q. Gu, “Network Aware Symbiotic Video Transcoding for Instream Rate Adaptation on Interactive Transport Control,” IEEE NCA’01, Oct. 2001, Cambridge, MA, pp. 201-213.

[Kha02] Khan J., R. Zaghal, and Q. Gu, “Rate Control in an MPEG-2 Video Rate Transcoder For Transport Feedback based Quality-Rate Tradeoff,” PV2002, Pittsburgh, PA, April 2002.

[Kha03a] Khan J., Zaghal R., and Gu Q., “Symbiotic Streaming of Elastic Traffic on Interactive Transport,” IEEE ISCC'03, Antalya, Turkey, July 2003.

[Kha03b] Khan J., and Zaghal R., “Jitter and Delay Reduction for Time Sensitive Elastic Traffic for TCP-interactive based World Wide Video Streaming over ABone,” Proc. of the 12th IEEE-ICCCN 2003, Dallas, Texas, Oct. 2003, pp.311-318.

[Kha03c] Khan J., R. Zaghal, and Q. Gu, “Dynamic QoS Adaptation for Time Sensitive Traffic with Transientware,” IASTED WOC'03, Banff, Canada, July 2003.

[Kul98] Kulkarni, A.B. Minden G.J., Hill, R., Wijata, Y., Gopinath, A., Sheth, S., Wahhab, F., Pindi, H., and Nagarajan, A., “Implementation of a Prototype Active Network”, First International Conference on Open Architectures and Network Programming (OPENARCH), San Francisco, 1998.

[Lee95] Lee Beng-Ong, "Wide Area ATM Network Experiments using Emulated Traffic Sources," Master's Thesis, University of Kansas, Lawrence, Kansas, 1995.

[Mal96] Malloth C., "Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks," PhD thesis, Federal Institute of Technology, Lausanne (EPFL), 1996.

[Mat96] Mathis M., Mahdavi J., Floyd S., and Romanow A., "TCP Selective Acknowledgment Options," IETF, RFC 2018, 1996.

[Men03] Mena S., Cuvellier X., Grégoire C., and Schiper A., "Appia vs. Cactus: Comparing Protocol Composition Frameworks," 22nd International Symposium on Reliable Distributed Systems (SRDS'03), Florence, Italy, pp. 189-200, October, 2003.

[Mir01] Miranda H., Pinto A., and Rodrigues L., "Appia, a flexible protocol kernel supporting multiple coordinated channels," In Proceedings of The 21st Int'l Conf. on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, USA, pp. 707 – 710, April 2001.

[Mir99] Miranda H., and Rodrigues L., "Communication support for multiple QoS requirements," In 3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal, April 1999.

[Pax98] Paxson V., Almes G., Mahdavi J., and Mathis M., "Framework for IP Performance Metric," RFC 2330, 1998.

[Per00] Perkins C.E., and Johnson D.E., "Route Optimization in Mobile IP," IETF, draft-ietf-mobileip-optim-10.txt, 2000.

[Per01] Perkins C.E., "IP Mobility Support for IPv4," revised draft-ietf-mobileip-rfc2002-bis-03-txt, 2001.

[Per96] Perkins C., "IP Mobility Support," RFC2002, IETF, October 1996.

[Pet99] Peterson L., "NodeOS Interface Specification", Technical Report, Active Networks NodeOS Working Group, February 2, 1999

[Pos81] Postel J., "Transmission Control Protocol," RFC 793, September 1981.

[Pra00] Pradhan P., Chiueh T., and Neogi A., "Aggregate TCP Congestion Control Using Multiple Network Probing," Proc. of the 20th International Conference on Distributed Computing Systems, ICDCS 2000.

[Ram00] Raman S., "A Framework for Interactive Multicast Data Transport in the Internet," Ph.D. thesis, UC-Berkeley, May 2000.

[Ram99] Ramjee R., et al. "HAWAII: A Domain-Based Approach for Supporting Mobility in Wide-area Wireless Networks," Proc. IEEE Int'l Conf. Network Protocols, 1999.

[Rej00] Rejaie R., M. Handley, and D. Estrin, "Architectural Considerations for Playback of Quality Adaptive Video over the Internet," Proc. of the IEEE ICON 2000.

[Sal84] Saltzer J., Reed D., and Clark D.D., "End-to-end arguments in system design," ACM Trans. Computer Systems, Vol. 2, No. 4, pp. 277-288, Nov 1984.

[Sch03] Schulzrinne H., Casner S., Frederick R., and Jacobson V., "RTP: A Transport Protocol for Real-Time Applications," RFC 3550, July 2003.

[SDLfrm] SDL Forum Society. SDL specification (z.100 11/99). http://www.sdl-forum.org.

[Sis98] Sisalem D. and Wolisz A., "Towards TCP-Friendly Adaptive Multimedia Applications Based on RTP," Proc. of the 4th IEEE Symposium on Computers and Communications, 1998.

[SiT99] Singh R., Tay Y., Teo W., and Yeow S., "RAT: A Quick (And Dirty?) Push for Mobility Support," 2nd IEEE Workshop on Mobile Computer Systems and Applications, pp. 32, Feb. 1999.

[SiV99] Sinha P., Venkitaraman N., Sivakumar R., and Bharghavan V., "WTCP: A reliable transport protocol for wireless wide-area networks," Proceedings of ACM Mobicom'99, Seattle, WA, pp. 231–241.

[Sri99] Srisuresh P., and Holdrege M., "IP Network Address Translator (NAT) Terminology and Considerations," RFC2663, 1999.

[Ste94] Stevens W. R., "TCP/IP Illustrated, Volume 1: The Protocols," Addison-Wesley, 1994.

[Ten96] Tennenhouse, D., and Wetherall, D. “Towards an Active Network Architecture”, Proceedings, Multimedia Computing and Networking, San Jose, CA, 1996.

[Tur93] Turner Kenneth J., "Using Formal Description Techniques-An Introduction to Estelle, LOTOS and SDL," John Wiley and Sons Ltd., 1993, ISBN 0-471-93455-0.

[Van93] Van Renesse R., Birman K., Cooper R., Glade B., and Stephenson P., "The Horus System. In Reliable Distributed Computing with the Isis Toolkit," IEEE Computer Society Press, pp. 133–147. 1993.

[Van96] Van Renesse R., Birman K. P., Glade B. B., Guo K., Hayden M., Hickey T., Malki D., Vaysburd A., and Vogels W., "Horus: A flexible group communications system," Technical Report TR95-1500, Department of Computer Science, Cornell University, Apr 1996.

[Vin97] Vinoski, S.,“CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments,” IEEE Communications
Magazine, Vol. 14, No. 2, February, 1997.

[Wet98] Wetherall, D., Guttag, J. and Tennenhouse, D., “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”, Proc. IEEE OPENARCH'98, San Francisco, CA, April 1998.

[Wid01] Widmer J., Denda R., and Mauve M., “A survey on TCP-friendly congestion control,” IEEE Network, vol. 15, pp. 28-37, May-June 2001.

[Wol97] Wolfinger B., “On the potential of FEC algorithms in building fault-tolerant distributed applications to support high QoS video communications,” Proc. of the sixteenth annual ACM symposium on principles of distributed computing, 1997, pp. 129-138.

[Wu01] Wu Jon Chung-Shien, Cheng Chieh-Wen, Ma Gin-Kou, Huang Nen-Fu, "Intelligent Handoff For Mobile Wireless Internet," Mobile Networks and Applications, Vol. 6, No. 1, pp. 67-79, Jan 2001.

[Yac00] Yacoub S., and Ammar H., "Finite State Machine Patterns," Pattern Languages of Program Design 4, pp. 413 – 440, Addison-Wesley Longman, 2000.

[Yav95] Yavatkar R., Bhagawat N., "Improving end-to-end performance of TCP over mobile internetworks," Proc. of The Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, pp. 146-152, 1995.

[Yem96] Yemini, Y., and Da Silva, S, "Towards Programmable Networks", IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, L'Aquila, Italy, October, 1996.

[Yok02] Yokota H, et al., “Link Layer Assisted Handoff Method over Wireless LAN Networks,” Proc. of MOBICOM ’02, Sept. 2002.

[Zag03] Zaghal R., and Khan J., “Event Model and Application Programming Interface of TCP Interactive,” Technical Report ‘TR2003-02-02’, Feb. 2003.

[Zag05] Zaghal R., Khan J., “EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno,” Technical Report TR2005-07-22, July 2005. http://www.medianet.kent.edu/technicalreports.html