To Appear in the Journal of of Computer Communications,
Elsevier Science Direct (Accepted May 2006)

Interactive Transparent Networking: Protocol Meta-modeling based on EFSM

Javed I. Khan and Raid Y. Zaghal
Networking and Media Communications Research Laboratories
Department of Computer Science, Kent State University
233 MSB, Kent, OH 44242
javed|rzaghal@cs.kent.edu

Abstract—the extensibility and evolution of network services and protocols had become a major research issue in recent years. The 'programmable' and 'active' network paradigms have been trying to solve the problems emanating from the immutable organization of network software layers by allowing arbitrary custom codes to be embedded inside network layers. In this work, we propose a new approach for building extensible network systems to support cross-layer optimization. The fundamental idea is to perform a simple, light-weight meta-engineering on the classical OSI protocols' organization to make it *interactive* and *transparent*. The protocols become (*interactive*) since they can provide event notification to service subscribers, and they become (*transparent*) since they also allow controlled access to their state information. Actual protocol extensions (or modifications) can then be performed at the application space by what we call *Transientware Modules*. This organization provides the infrastructure needed for easy and practical extensions of the current network services and it becomes much easier to address other difficult issues like security and flexibility. We call this mechanism *Interactive Transparent Networking* (*InTraN*) and we call the extended kernel *InTraN*-enabled. We have realized a FreeBSD implementation of the extensible *InTraN*-enabled kernel. In this paper, we present a formal EFSM-based model for the proposed meta-engineering and illustrate the principles through a real example of TCP extension. Then, we demonstrate how it can be used to realize equivalents of other protocol modifications by showing the *InTraN* model of 'Snoop' [4].

Keywords: interactive transparent networking, active networks, interactive TCP, protocol meta-engineering.

1. Introduction

Traditional network software stack has been designed with a pseudo-layered organization. Each layer is intended to offer a specific sub-service in the overall task of high level communication between application end-points. This organization used to be a blessing but no longer. These network layers, the organization of the services and their specific implementations now turn out to be quite rigid and frustratingly immutable. The increased sophistication and complex communication needs of the applications as well as the increased diversification of the underlying network infrastructure now require more dynamic and informed adaptive extension (or modification) of the existing protocols. Some situations even call for new functionalities and services as well [6]. The end-to-end paradigm has tried to address this demand by proposing application layer extensions (such as RTP [29]). Almost by definition any 'network' adaptation requires awareness about the dynamics of events and the status within the 'network'. Fundamentally, network adaptation requires triggers that originate from the network infrastructure. However, the end-to-end approach faces difficulty due to the classical 'black box' design. Network software layers have been designed as a closed box from the application's point of view [28]. Therefore, to circumvent this problem, end-to-end solutions try to estimate an approximation of network states by external measurements such as probe-packets [1, 8, and 25]. A key advantage is that end-to-end modules can run as application level components. Thus, they are dramatically easy to realize without requiring any modification in the network software layers. Early end-to-end solutions were therefore quite successful and inspiring. However, as more complicated solutions were attempted, it became evident that there is often a substantial handicap in the timeliness and the type of information that can be estimated from application layer functions only. The network remains completely non-communicative and silent from all its end-points. Some states and statistics are indeed readily available in the end-point lower layer modules (such as RTT, loss information, power level, etc). But, because of the black box design, this information remains out of reach from the end-to-end modules. These upper modules have to play a frustrating game of guessing these network states by indirect and often redundant means. In contrast to end-to-end approach, researchers however have proposed various direct and custom modifications (or enhancements) to network protocols. However, these enhancements were viewed as requiring serious changes within networking software layers. Thus a majority of otherwise bright ideas have faced a serious acceptance problem. Most of these solutions achieved very limited success towards real implementation and deployment. The paradigm of *active* and *programmable* networking attempted to find a holistic framework for custom modifications within network software layers [5, 9]. In a way, such customized networking approach can be considered as diametrically opposite to the end-to-end approach. To make protocols more adaptive, it allows installing modifications/extensions right into the network layer where all the events (triggers) and state information are readily available. Unfortunately, this approach introduces another set of even more serious problems. Typically the network system space has not been designed for multi-user execution environment. Thus, issues like resource sharing and security has remained unresolved in *active* and *programmable* network systems. Apparently both of these two approaches have the noble goal towards building evolvable, flexible, and extensible network services and have very attractive properties. But, as of today, both are still facing their unique set of formidable difficulties.

Is it possible to combine the best of both? It seems an innovative solution may be formulated if a distinction can be made between the information trigger needed to initiate adaptation, and the actual action code. The programmable network took the approach of keeping both within network layers, while the end-to-end networking approach took the approach of keeping both at the application layer. It seems there might be a third approach which may be able to keep the best of both.

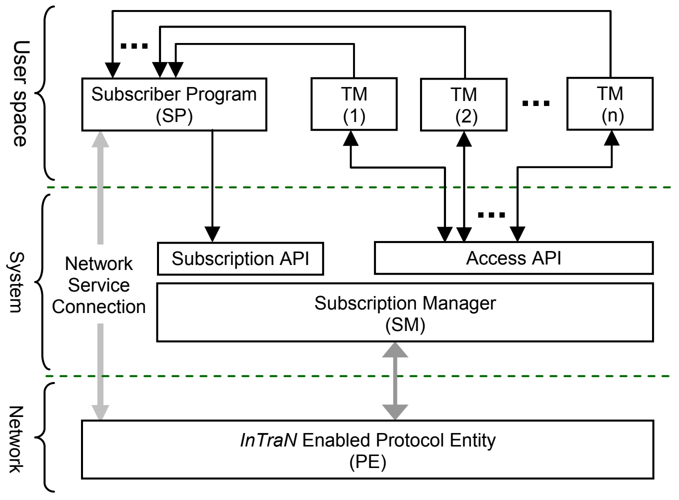
In this research we present an experimental system which takes advantage of this distinction. It seems the need for changing and embedding custom components inside network software stems from two realities: first, we are seeing that the data in every communication pathway has to be processed in custom and adaptive ways. No finite set of pre-agreed, *fixed* protocols may ever foresee and satisfy all cases. Secondly, these customizations are very dynamic and communication-case specific. Many of the triggers for the accompanying custom actions originate right inside the network software. However, instead of embedding codes to create customized actions within network layers, the new approach suggests creating a mechanism to *pull-up* the required state information (of the service or protocol) into the application layer. Then, the actual actions can be formulated by programmable components running at the application layer. We call the new paradigm *Interactive Transparent Networking* (*InTraN*). The scheme creates handles to be able to *push-down* the generated actions into the target network layer if needed. This relieves lower network layers from housing costly custom components and from dealing with complex issues regarding security and resource sharing. The attraction is that the application space already has a well developed provision to run custom codes, share resources, and handle security issues for managing multiple trust domains, etc, much of that can be reused.

What are the potential costs? Yes, *InTraN* still requires some reorganization in the network software. However, in contrast to creating a network software organization which can house custom codes, as proposed in active networking, it requires creating a network software organization which can facilitate service state information exchange. Some meta-engineering of the protocol is still required, though it is much lighter. Compared to the active networking approach there is also a potential concern about performance degradation since actions are performed in relatively upper layer, though it is expected to be much faster than the end-to-end approach since much of the state information can now be derived from the local network protocol end-point. However, the potential advantage of easy extensibility seems to be very lucrative.

We have recently implemented a *transparent FreeBSD* based on the *InTraN* paradigm. As an instance of interactive protocols, we have also implemented an interactive version of an otherwise legacy TCP, which we call *interactive-TCP* (iTCP). There are already some excellent proposals for network protocol enhancements and extensions. However, despite their functional advantages, many of these creative solutions faced a crippling deployment problem because they required highly individualized changes within the network software. To test the extension power of our approach, we have selected one such instance based on TCP derivatives—the 'Snoop' protocol proposed by Balakrishnan et al. [4]. We demonstrate how it can be easily implemented at the application level and operate on demand within the *InTraN* paradigm. Through this real implementation we explore the reorganization requirement in network software layers, the formal meta-engineering process of the involved protocols, and

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 paper.
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.



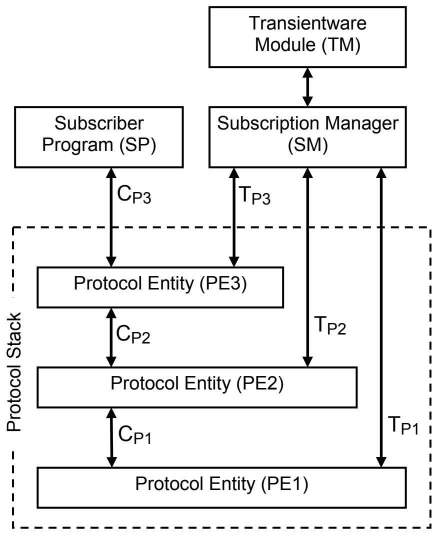


Figure 2. *InTraN* channel extension.

Table 2. Types of Transientware Modules

TM Type	Definition
Signal-Only	A *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	A *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 $v \in V''_P$).

the transientware based extension mechanisms. Along that, we expose the performance and security issues associated with this new approach and provide a comparative picture.

The paper is organized as follows: in section 2 we formally explain the *InTraN* paradigm using SDL language. In section 3 we present iTCP—a simplified version of TCP with *InTraN* meta-engineering. In section 4 we show how the *InTraN* paradigm can be used to model 'Snoop'. In section 5 we discuss performance and security issues and we conclude in section 6.

2. Interactive Transparent Networking (InTraN)

2.1 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)[1] [12, 30] to formally describe (a) the protocol meta-engineering process and (b) the network software organization needed to support the interactivity and transparency. First let us briefly review SDL. The programming model used by SDL is based on extended finite state machines (EFSM) [7, 12]. 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 this EFSM 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 [12]. Upon receiving an event, the machine checks a predicate that is composed of variables, logical operators (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).

[1] SDL 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.

2.2 A framework for the *InTraN* paradigm

2.2.1 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 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*.

*TM*s are also classified into three types based on their access privileges to protocol's internal variables. These are shown in table 2. A *TM* is granted read-only access to a subset of *PE*'s local data (variables). In certain circumstances the *TM* is allowed even to modify a subset of these accessible variables as long as 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 3 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 $E'_P \subseteq E_P$.

2.2.2 SP-SM Interfacing: Subscription Mechanism

The *IntTraN* framework 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 4. 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

Table 3. Types of variables and their access privileges

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 4. Subscription API

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

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 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(*V*) request. If the flag (F) is (true), then (*val*) is valid, otherwise, the *TM* just ignores (*val*).
Invoke(*TM*)	The *SM* invokes a registered *TM* after receiving an *Event()* signal
Finish(*TM*)	The *TM* signals 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 activate/deactivate the event in *PE* which 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*

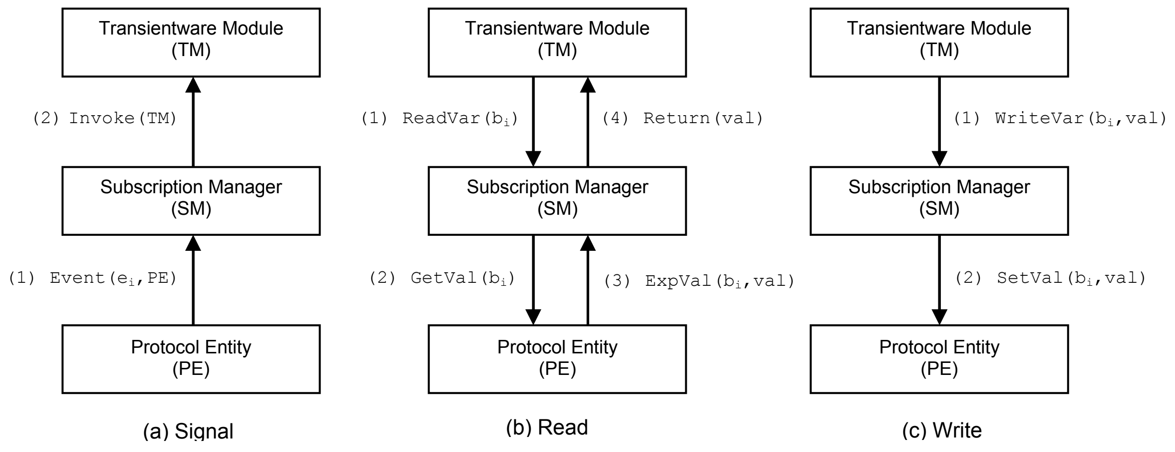


Figure 3. *TM* interfacing between the *PE* and the *TM* for three scenarios.

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 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.

2.2.3 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. 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 3 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*, the signal Event(e_i, P) is sent to the *SM* indicating the event type and the protocol. The *SM* looks into its subscription lists to find which *TM*s are currently bound to such (event, protocol) pair, and then it activates them by the Invoke(*TM*) 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*. When the *TM* finishes, and before it is terminated, it sends a Finish(*TM*) request to the *SM*. The *SM* then removes the *TM-Instance* record of the terminating *TM*.

(b) Read Access

When a *TM* wants to read the value of a certain variable v_i from the underlying *PE*, it sends a ReadVar(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 the *TM* that issued the read request is eligible (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 the *TM*. 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 the *TM* 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* framework 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 intention of the designer. Therefore, protocol modifications can be realized through a group of carefully designed *TM*s which can manipulate certain properties of the EFSM through interaction; (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*. The *TM* makes a write request and passes the variable name and its new value to the *SM* via a WriteVar (v_i, *val*) operation. If v_i is modifiable and the *TM* is *Read-Write* type, the *SM* generates a signal SetVal (v_i, *val*) to the *PE*. Otherwise, it simply ignores this WriteVar() request. When the EFSM of the *PE* consumes the SetVal() signal, it simply runs the assignment $v_i := val$.

2.2.4 Protocol Meta-Engineering

The meta-engineering of a *PE* involves adding new events and transitions to its EFSM. 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 4 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:

1) A new transition triggered by the signal SetVal(d_i, *val*).
2) A new transition triggered by the signal GetVal(d_i).
3) A new transition triggered by the signal SetFlag(E_i, *val*)

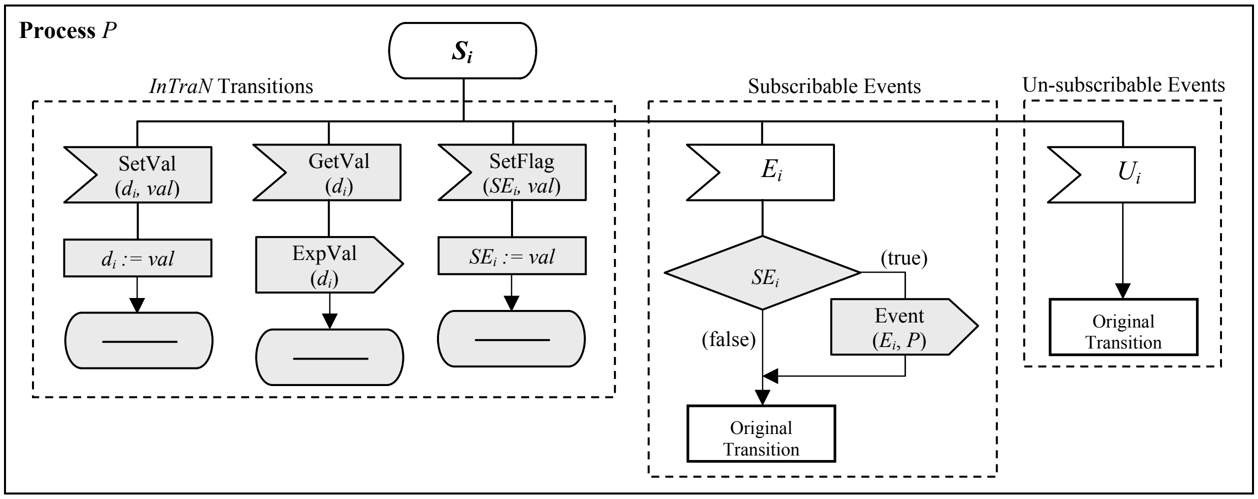
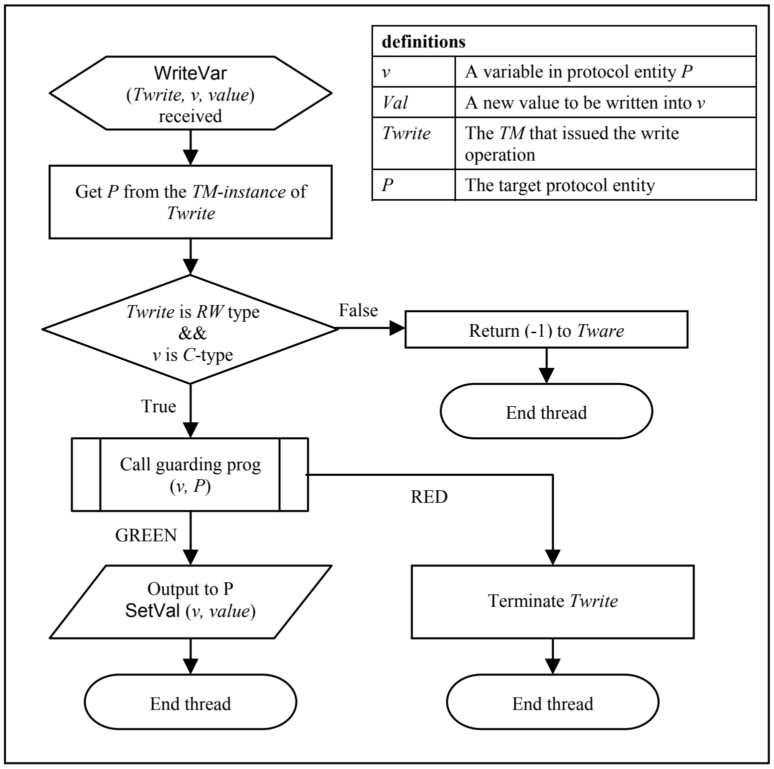


Figure 4. Protocol meta-engineering extension.



WriteVar (*Twrite, v, value*) received
Get *P* from the *TM-instance* of *Twrite*
Twrite is *RW* type && *v* is *C*-type
False
Return (-1) to *Tware*
End thread
True
Call guarding prog (*v*, *P*)
RED
GREEN
Output to P SetVal (*v*, *value*)
Terminate *Twrite*
End thread
End thread

definitions	
v	A variable in protocol entity *P*
Val	A new value to be written into *v*
Twrite	The *TM* that issued the write operation
P	The target protocol entity

Figure 5. The *SM* thread that handles WriteVar() operation.

4) 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 4. 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 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, 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 E_i signaling service. The SetVal() and GetVal() signals correspond to the write-access and read-access operations which were described in the previous sub-section.

2.2.5 Security Model

Since *InTraN* 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). We have to be concerned only when a *TM* is allowed to write to protocol's internal variables (i.e., *Read-Write* mode).

Here, we propose a security model which allows controlled access to protocol's variables while maintaining system stability. We define two role types who can be involved in any *InTraN*-based solution: (1) protocol designer, and (2) *TM* designer. The components given by the protocol designer must run with super-user access. He basically performs the one time meta-engineering of 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. The *TM* designer can be any user; s/he implements a particular protocol solution/extension by coding one or more *TM*s. S/he uses the facilities offered by the underlying *InTraN*-enabled system to implement the intended solution.

It can be seen that only when a *TM* of type *Read-Write* tries to update a *C* type variable, then system security (or protocol 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 (e.g., wrong type declaration), and (2) a malicious *TM* of type *Read-Write*. When a system is running with a dangerous combination, the operating system can optionally activate 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 an *index* variable) 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 an operating system utility. 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 3. This is determinable by static impact analysis of the protocols EFSM. This way, the integrity of the *InTraN*-enabled system can be preserved even in the presence of potential 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 [14]). The *SM* can be programmed to initiate a special thread program to handle the WriteVar() operation and the *dangerous combination*. Figure 5 describes the basic algorithm: Assuming a *TM* called (*Twrite*) has issued the following write operation: WriteVal(*Twrite, v, value*). First, the *SM* consults the *TM-instance* of *Twire* 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 *Twrite* is *Read-Write* type *TM* and *v* is *C* type). 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 approve 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 *Twrite*. Let *N* be the number of *PE*s in the system 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.

Besides this operation, the application and the *TM*s run in the user space of the native operating system with all the usual security protections. Thus, the rest of the *InTraN*-enabled system remains as secure as the original non-extended system. If we compare this to the active networking model for example, we cannot find a peer security strategy.

3. Interactive-TCP (iTCP)

We demonstrate the *InTraN* principles outlined in the previous section by means of a simplified TCP protocol. First, we formally describe the abstract protocol using SDL, and then we augment the protocol by adding *InTraN* components. [32] 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 new version features the slow start/congestion avoidance mechanism [15], and the fast retransmit/fast recovery mechanism [16].

3.1 The SDL model

The simplified TCP service 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 6 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 writer application to the TE, and the channel *SR* carries the *AppRead* signal from the RE to the reader 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 [appx1] (please see last 4 pages) 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 details are hidden inside procedure calls, e.g., CalcRTO().

The EFSM of this system is depicted in figure 7 and is described as:

- *S = {Slow Start, Data Transfer, Fast Recovery, Closed Window},*
- s_0 *= Slow Start,*
- *E = {AppWrite, RecvACK, rexmt timeout},*

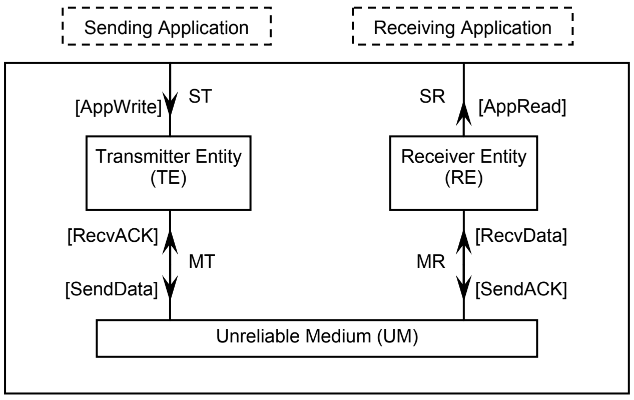


Figure 6. Half-duplex TCP service composition.

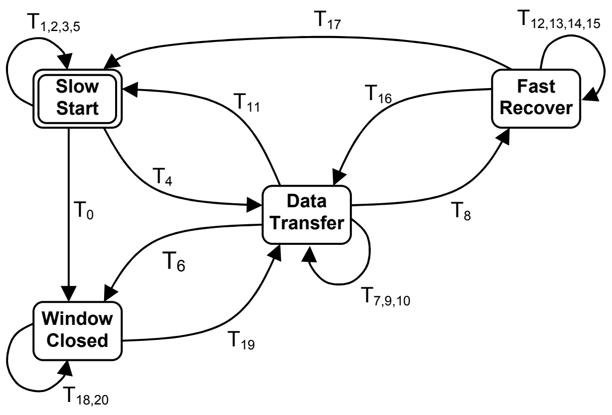


Figure 7. System's EFSM.

- *O = {SendData},*
- *V = {seqno, ackno, RAW, dACK, pACK, FRFlag, RTO, rexmt, Cwnd, Swnd, LU, LS, ExpBoff}.*
- *f = {T_0, T_1, …, T_{20}}. These transitions are labeled in figure appx1.*

3.2 EFSM of iTCP

Interactive-TCP is a real interactive transport protocol based on the *InTraN* framework. We wanted to track two events in TCP: '*retransmission timer timeout*' and '*receiving third duplicate ACK*'. Both events signify packet loss and usually cause TCP to trigger congestion control procedures.

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 4 (i.e., by adding three transitions for the *GetVal, SetVal, and SetFlag* events, and modifying existing transitions of subscribable events in every state).

Where *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* and *RT*) which cannot be included in the set *B* or *C*.

3.3 The *TMs*

We have experimented iTCP with elastic video traffic by allowing an adaptive video transcoder [8] to intercept the video stream and modify the generation bit rate based on the *InTraN* service feedback. We designed the following simple scheme: when the network is congested (e.g., '*RexmtTimeout*' event has occurred) *InTraN* triggers a *TM* to reduce the generation bit rate of the video transcoder. When the network recovers from congestion, another *TM* orders the transcoder to resume transmission at the normal bit rate. We have realized the scheme by writing two *TM*s: TM-rexmt and TM-ack. The algorithms of these *TMs* are shown in figure 8. We let the Subscriber Program—in this case the video transcoder—subscribe by running: Bind

(1) TM-rexmt (Type: Signal-Only)

```
bitRate := bitRate * RRF;
Finish(TM-rexmt);
```

(2) TM-ack (Type: Read-Only)

```
ReadVar(dACK);
If (dACK = 3)
        bitRate := bitRate * RRF;
Else{
        ReadVar(Swnd);
        ReadVar(RAW);
        If (Swnd >= RAW)
                bitRate := Normal Rate;
}
Finish(TM-ack);
```

Were:

bitRate: is a parameter used by the transcoder to decide the generation bit rate of the next video frame.
RRF: is the Rate Reduction Factor (less than 1).
dACK: the number of duplicate ACKs received at TCP sender.
Swnd: TCP sender window size.
RAW: Receiver Advertised Window. The window size advertised by the TCP receiver in the last ACK.

Figure 8. Video Transcoder *TMs*

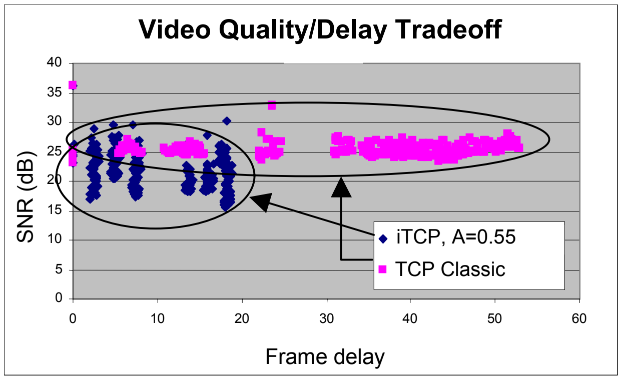


Figure 9. Quality/delay tradeoff offered by iTCP. Frame delivery delay was dramatically reduced by controlled trade-off of the SNR quality.

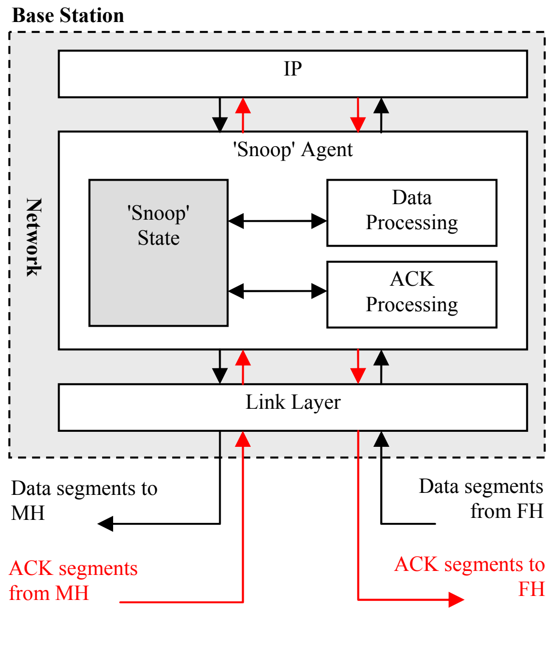


Figure 10. Conventional 'Snoop'.

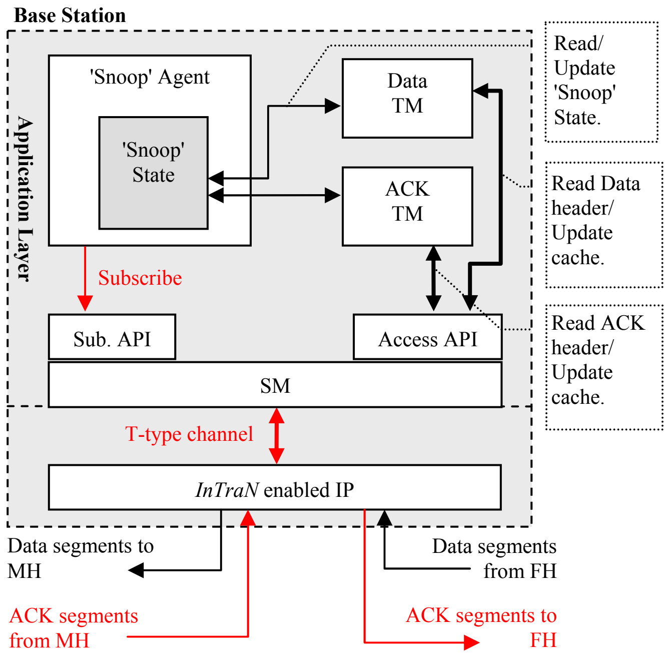


Figure 11. Interactive 'Snoop' (i'Snoop').

(*RexmtTimeout*, *TCP*, *TM-rexmt*); Bind (*RecvACK*, *TCP*, *TM-ack*).

3.4 Performance

The scheme proved to be effective and had shown substantial gain in video performance metrics like frame-wise end-to-end delay and referential jitter. Figure 9 shows a sample of performance results which highlights the tradeoff offered by iTCP. Here, each video frame is plotted as a point in the video quality/frame delay plane. As can be seen from the region of the two QoS distributions, in classical TCP, although frames have been generated with SNR quality ranging between 22-29 dB, but many of these frames were lost in transport, and were never delivered. In contrast, the proposed iTCP (with a bit Rate Reduction Factor RRF=0.55) delivered all the frames with 15-17 delay guaranteed at 15-26 dB quality. Fundamentally, what *InTraN* solution has offered is a qualitatively (as opposed to the quantitative improvements offered by unaware solutions) new empowering mechanism, where the catastrophic frame delay can be traded off for acceptable reduction in SNR quality. We have published detailed performance analysis on iTCP and related experiments in [18, 19].

4. Protocol Modeling Example

In this section we briefly describe the well-known 'Snoop' protocol [4] which has been proposed—among many other schemes in the literature—to improve TCP performance over wireless links. Then, we show how it can be re-modeled via 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 [2, 3, and 11].

4.1 “Snoop”

First we explain the basic idea of [4] which is the core Snoop. In a later section we discuss some of its modifications [13, 20, 21, 31, and 35]. The 'Snoop' protocol introduced a module, called 'Snoop', at the base station that monitors the wireless link for every packet that passes through in both directions.

The 'Snoop' module maintains a cache of TCP packets sent from the fixed host (FH) that have not yet been acknowledged by the mobile host (MH). 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 locally retransmits the lost packet if it has it in the cache. Thus, the base station can hide the packet loss from the fixed host, therefore avoiding its invocation of an unnecessary congestion control mechanism. Figure 10 describes the architecture of the classic 'Snoop' protocol. For brevity, it shows the part of 'Snoop' that handles one direction of the traffic only (Data segments from FH to MH and ACK segments from MH to FH). As it can be seen, it requires substantial extension inside network software layers. However, with *InTraN* the same functionality can be achieved at application layer with an *InTraN*-enabled IP protocol (or iIP). Figure 11 shows this *InTraN* implementation of 'Snoop'—which we call 'iSnoop'. In this scheme, the 'Snoop Agent' (shown in figure 11) subscribes two iIP events: an ACK received from MH event (Recv_ACK), and data segment received from FH event (Recv_DAT). 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* and subscribe with the *InTraN* service. The rest of the work is now done by the *TM*s. The *Snoop State* is similar to that of conventional 'Snoop'. The *TM-Data* handles the (Recv_DAT) event and implements the *Data processing* algorithm of 'Snoop'. The *TM-ACK* handles the (Recv_ACK) event and implements the *ACK processing* algorithm of 'Snoop'.

As can be seen this implementation also keeps the cached data segments in the TCP buffer just like the classical 'Snoop' and thus retains the same performance

Table 6. Cost parameters for Snoop/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

Table 7. Algebraic overhead cost of Snoop and iSnoop for three scenarios of wireless link properties

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	$SNOOP_{free}+C_{hoff}\ (8T / nR)$	$iSNOOP_{free}+C_{hoff}\ (8T / nR)$

Table 8. Running modes for the getrusage() experiment

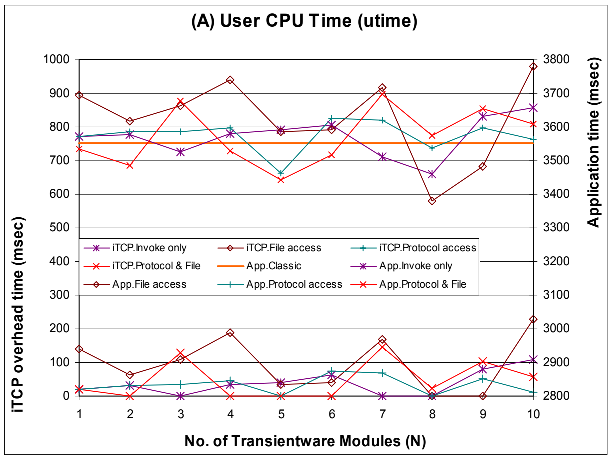
Mode name		Description
Classic TCP		No interactivity overhead. This is the reference case.
iTCP modes	**Invoke only**	Subscribe with a *Signal-only* type *TM*. The *TM* does not perform any Read/Write operations.
	File access	Subscribe with a *Signal-only* type *TM*. We let the *TM* open a disk file 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.

Table 9. iTCP's CPU time overhead

	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 10. iTCP's 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%



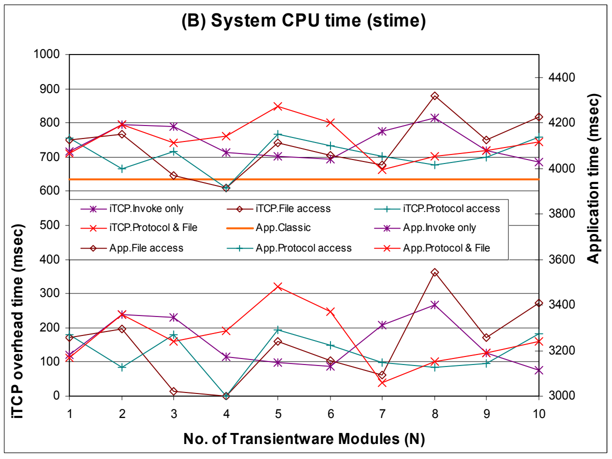
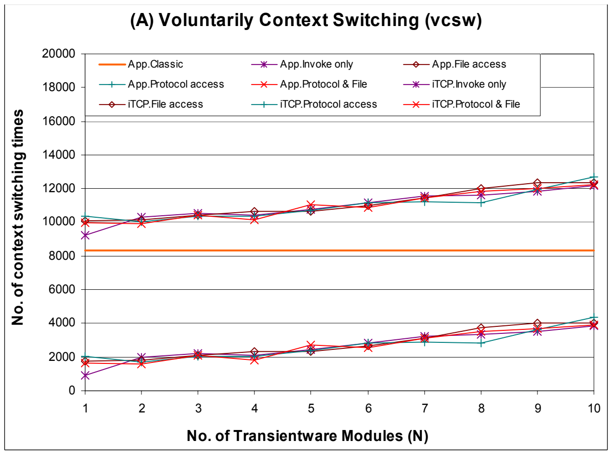


Figure 14. CPU time: iTCP overhead vs. application



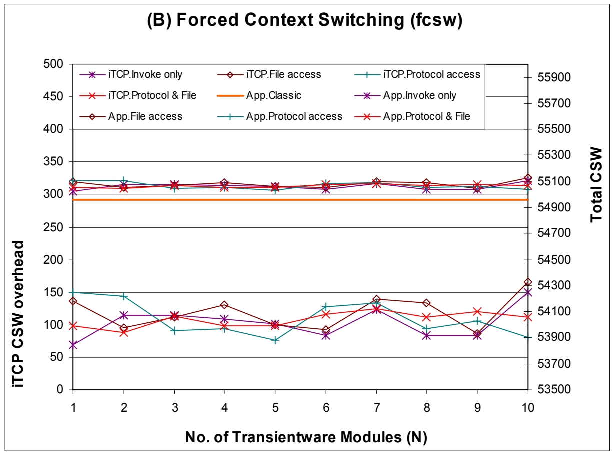


Figure 15. Context switching: iTCP overhead vs. application

advantage. The Snoop algorithms are described in detail in [4].

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. As can be seen the entire 'Snoop' logic has been implemented in application layer. The 'Snoop' protocol uses a different mechanism to handle traffic on the opposite direction. This too can be easily modeled with the *InTraN* paradigm in a similar fashion. The advantages of the *InTraN* engineering does not stop at the classical 'Snoop', indeed, more advanced versions of the basic 'Snoop' strategy can also be implemented with equal ease. For example, more advanced error correction mechanism—such as forward error correction (FEC) [24] can be implemented as well with the same effort without requiring a new round of standardization/modification in network layers.

4.2 'Snoop' Variants & InTraN

It is interesting to note, that many improvements and modifications on the original Snoop have been proposed since its first publication in 1995 [13, 20, 21, 31, and 35]. The evolutionary history of Snoop is a good example that highlights the importance of meta-engineering provisioning. Some of these newer techniques are variations of Snoop and few are expansions to Snoop. The later keeps all the basic features of Snoop and adds new ones, while the former substitutes. For example, [35] introduces a TCP-SACK aware Snoop. Their work was motivated by observing several versions of TCP with and without Snoop. They found that the Snoop protocol improved the performance of TCP Vegas considerably but in the case of TCP SACK, the effect of using the Snoop protocol was actually negative. The algorithm proposed in [35] helps Snoop differentiate between an ordinary ACK and a SACK block. In the case of an ordinary ACK the SACK-Aware Snoop retransmits only the packet as suggested by the sequence number of the duplicate ACK. However, in the case of a SACK block the protocol retransmits all the packets indicated by the SACK block. [13] has extended this idea by employing a SNACK mechanism on the BS and the MH to provide explicit information on multiple packet losses over the wireless link. Two protocol components are used: a SNACK-Snoop deployed at the BS and a SNACK-TCP deployed at the MH. Another enhancement is found in [20], an ARQ Snoop Agent is inserted between TCP and MAC layers at both the sender and the receiver to exploit the ARQ link layer information for a more efficient acknowledgement of TCP packet delivery (e.g., When a TCP packet is successfully delivered at the link level, the TCP ACK for the transport layer will not be sent through the channel, but it will be automatically generated locally at the sender side). It is worth noting that all of these variants can be easily modeled with the *InTraN* EFSM-based paradigm especially that most of them are natively event-based.

The implementation of these Snoop enhancements would require permanent changes in the network layers. Unfortunately, these are yet to see any real deployment despite the benefits they offered. In sharp contrast to the conventional approach to protocol reengineering, the base Snoop including these recent enhancements can be easily deployed using *InTraN*. For enhancements it will require modifications only at application layer. [20] Will require *TM*s to subscribe a new event. Only *TM*s have to be substituted under *InTraN* paradigm of meta-engineering.

This further illustrates the crucial advantage of the idea behind the *InTraN* paradigm—to provide an alternative way to model and implement protocols' extensions or modifications. Not only it reduces deployment cost, but also it provides the new capability to easily switch between implementations. Often, there is no decidedly winner alternate, different versions wins in different cases.

5. Performance Issues

5.1. 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 6 we show several quantities that define cost variables and wireless link characteristics. The first column in table 7 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). Besides the reference case, the other two scenarios are identical in both protocols.

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 8.

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 14 and the latter two are plotted in figure 15.

A) CPU time Analysis
In figure 14 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. We can see that (*utime*) overhead—figure 14(A)—varied between 0 and 220 msec, while (*stime*) overhead—figure 14(B)—varied between 0 and 360 msec. This is a small percentage of the total running time in both cases as we show in table 9. In the table we also show the standard deviation (SD) of the iTCP overhead over the 10 runs. We could not determine a consistent pattern of CPU time overhead as *N* increases. This means that once the *InTraN* service has been activated, *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 and showed a higher (SD).

B) Context Switching Analysis
In figure 15 we show context switching overhead and in table 10 we show the overhead as a percentage of the total context switching. 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 that it added to (*fcsw*)—between 70 to 170 context switches. Percentage wise, as shown in table 10, 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.

5.2. Security and practice

As we saw in the previous section, any *InTraN* based solution will incur a small overhead cost on the application and the OS levels. But this can be justified for the practical gains allowed by employing the *InTraN* paradigm. Since *TM*s run in the application space, they will enjoy a well developed environment that has been fine-tuned to run custom codes, share resources, and manage security issues. Actually, the security issue is of great importance in such engagement. For example, running "Active Modules" inside the network (e.g. Active and Programmable Networks [10]), 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. Conversely, within the *InTraN* paradigm, the *Subscriber Programs* and *TM*s can only access internal network services through the API extension, and by imposing the appropriate access restrictions on these entities, we can guarantee a certain security level. Furthermore, since these 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.

6. Concluding Remarks

The Interactive Transparent Networking (*InTraN*) paradigm can offer two fundamental benefits, (i) it becomes much easier and practical to implement and deploy the modeled protocol on a real network, and (ii) new extensions or alternative algorithms—invoked as application level Transientware Modules (*TM*s)—can be experimented with the new protocols without changing the underlying infrastructure. For example, a protocol like 'Snoop' 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 performed extensive performance evaluations [17, 26] and have also proposed many other creative schemes [27, 33]. The proposed transparency via interaction and the 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 to upgrade a particular *TM* to another improved one.

The work has been supported by the DARPA Research Grant F30602-99-1-0515.

7. References

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

[2] 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.

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

[4] Balakrishnan H., Seshan S., and Katz R., "Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks," ACM Wireless Networks, Vol. 1, 1995.

[5] 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.

[6] Blumenthal M.S., 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.

[7] 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.

[8] 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.

[9] Campbell A., Meer H., Kounavis M., Miki K., Vicente J., and Villela D., "A Survey of Programmable Networks," ACM Computer Communications Review, April 1999

[10] 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.

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

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

[13] Fanglei Sun; Li, V. O. K. and Liew, S. C., "Design of SNACK mechanism for wireless TCP with new Snoop," IEEE Wireless Communications and Networking Conference, Atlanta, GA, USA, March 2004.

[14] Huang Y-W., Yu F., Hang C., Tsai C-H., Lee D-T., and 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.

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

[16] Jacobson V., "Modified TCP Congestion Avoidance Algorithm," end2end-interest mailing list, April 1990.

[17] Jianxuan X., Labrador M., Guizani M., "Performance Evaluation of TCP over Optical Channels and Heterogeneous Networks," Cluster Computing, Vol. 7, Issue 3, pp. 225-238, July 2004.

[18] 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.

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

[20] Kliazovich, D. and Graneill, F., “A cross-layer scheme for TCP performance improvement in wireless LANs,” GLOBECOM '04. IEEE Global Telecommunications Conference, Dallas, TX, USA, Dec. 2004.

[21] Kui-Fai Leung and Yeung, K. L., “G-Snoop: enhancing TCP performance over wireless networks,” Proceedings. ISCC’04. 9th International Symposium on Computers and Communications, Alexandria, Egypt, July 2004.

[22] Lee S.B., Ahn G.S., Campbell A.T., “Improving UDP and TCP performance in mobile ad hoc networks with INSIGNIA,” IEEE Communications Magazine, Vol. 39, Issue 6, pp. 156-165, June 2001.

[23] Lin H.,Das S.K., “Performance study of link layer and MAC layer protocols to support TCP in 3G CDMA systems,” IEEE Transactions on Mobile Computing, Vol. 4, Issue 5, pp. 489 – 501, Oct. 2005.

[24] Luby M., Vicisano L., Gemmell J., Rizzo L., Handley M., and Crowcroft J., "Forward Error Correction (FEC) Building Block," RFC 3452, Dec. 2002.

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

[26] Racherla G., Radhakrishnan S., and Sekharan C., "Performance evaluation of wireless TCP with rerouting in mobile networks," Computer Communications, Vol. 26, No. 6, pp. 542-551, April 2003.

[27] Ratnam K., Matta I., "WTCP: an efficient mechanism for improving wireless access to TCP services," International Journal of Communication Systems, Vol. 16, Issue 1, pp. 47 – 62, February 2003.

[28] Saltzer J., Reed D., and Clark D.D., “End-to-end arguments in system design.” ACM Trans. Comput. Syst., Vol. 2, No. 4, Nov., pp. 277-288, 1984.

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

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

[31] Seung-Chan Lim; Woo-Jae Kim and Young-Joo Suh, “Rate-adaptive Snoop cache allocation to guarantee TCP fairness in wireless networks,” IEEE 60th Vehicular Technology Conference. VTC2004-Fall, Los Angeles, CA, USA, Sept. 2004.

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

[33] Vaidya N., Mehta M., Perkins C., and Montenegro G., "Delayed duplicate acknowledgements: a TCP-

[34] Van Der Schaar M., Sai Shankar N., “Cross-layer wireless multimedia transmission: challenges, principles, and new paradigms,” IEEE Wireless Communications, Vol. 12, Issue 4, pp. 50 – 58, Aug. 2005.

[35] Vangala, S. and Labrador, M., “The TCP SACK-aware Snoop protocol for TCP over wireless networks”, IEEE 58th Vehicular Technology Conference. VTC 2003-Fall, Orlando, FL, USA, Oct. 2003.

[36] Xi Zhang, Jia Tang, Hsiao-Hwa Chen, Song Ci, Guizani M., “Cross-layer-based modeling for quality of service guarantees in mobile wireless networks,” IEEE Communications Magazine, Vol. 44, Issue 1, pp. 100-106, Jan. 2006.

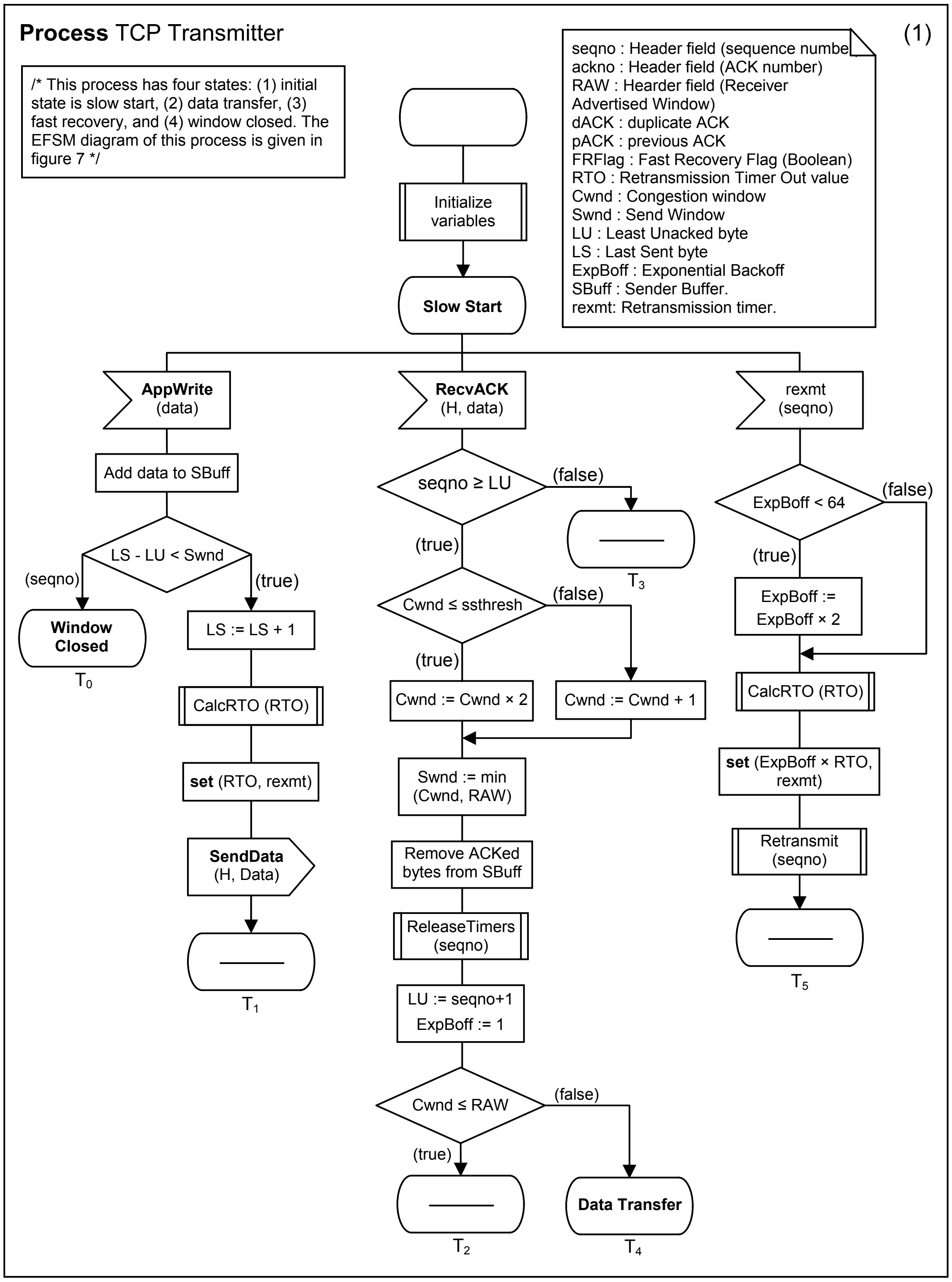


Figure appx1. SDL description of a simple TCP

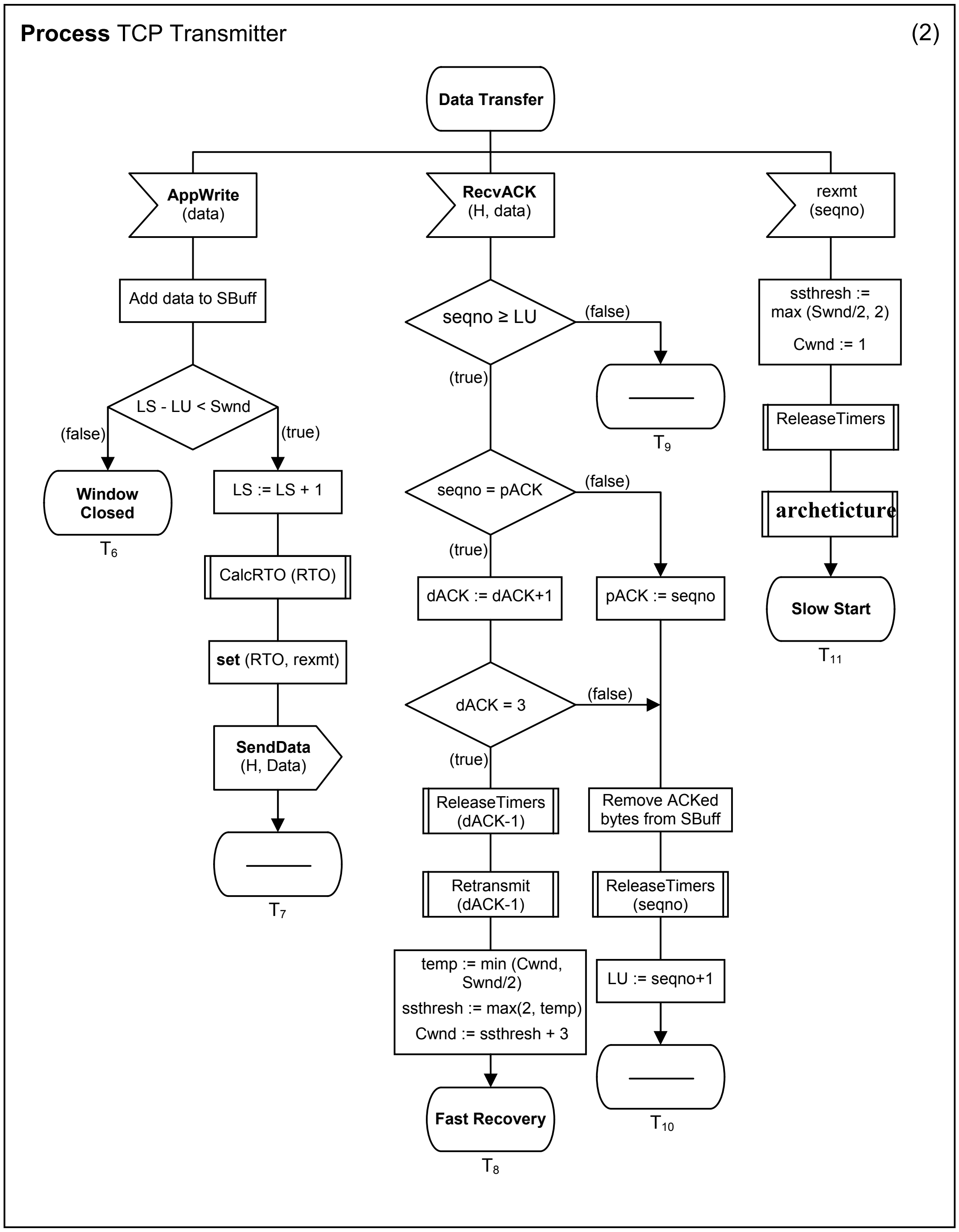


Figure appx1. (continued)

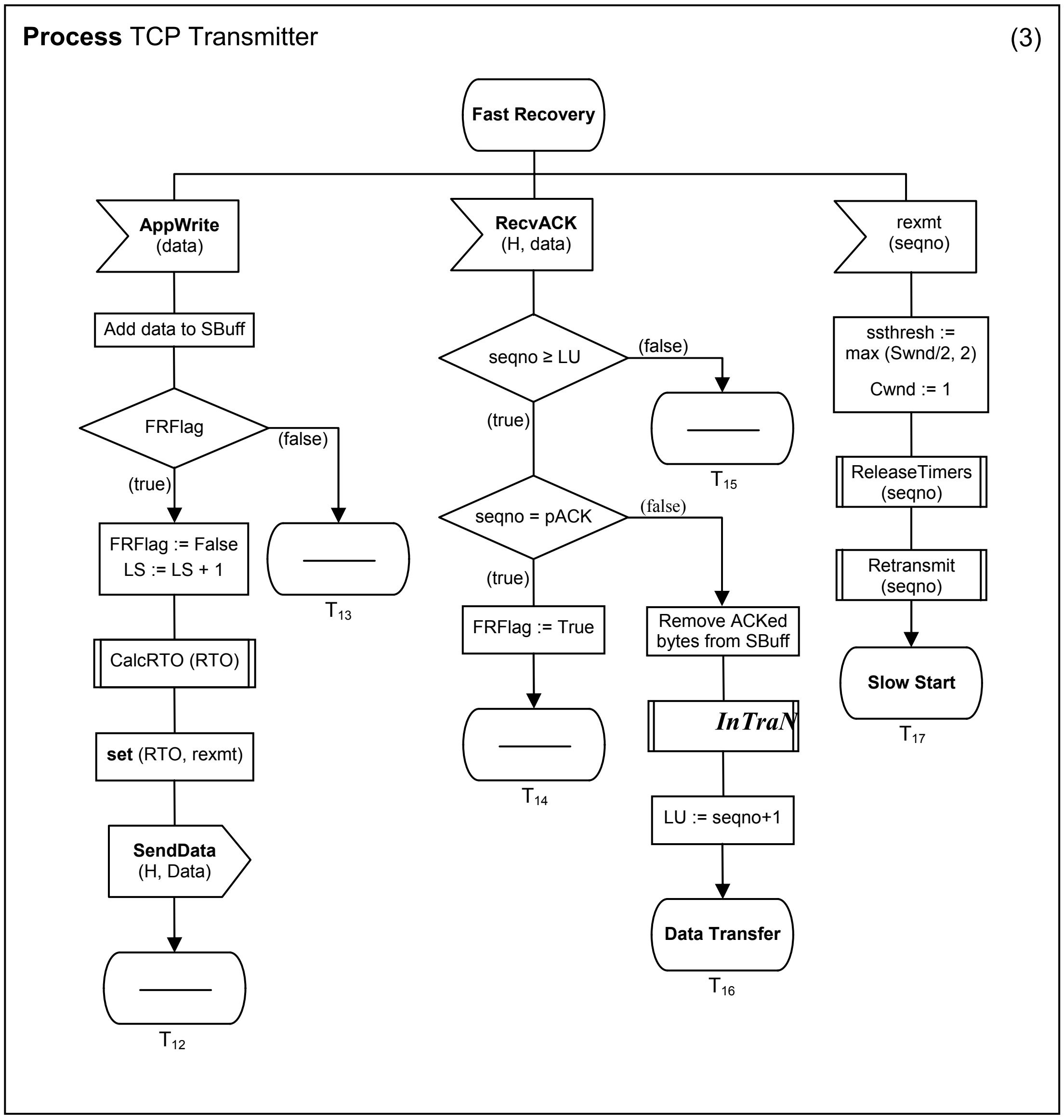


Figure appx1. (continued)

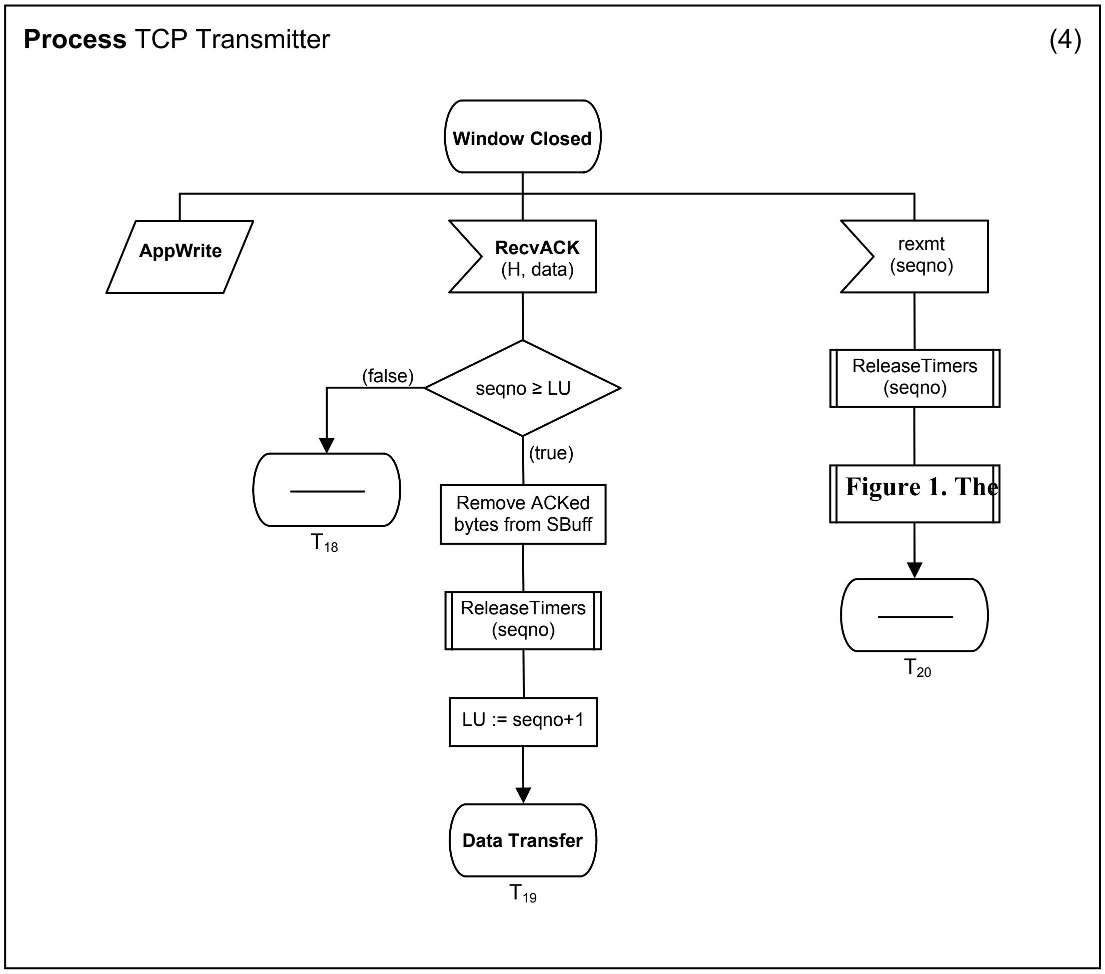


Figure appx1. (continued)