MODELING OF TRANSMISSION CONTROL PROTOCOL USING ROBUST OBJECT CALCULUS

By

KISHORE D. CHAKRAVADHANULA

A thesis submitted in partial fulfillment of
the requirement for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

December 2000

To the faculty of Washington State University:

The members of the Committee appointed to examine the thesis of KISHORE D. CHAKRAVADHANULA find it satisfactory and recommend that it be accepted.

_____

Chair

_____

_____

ACKNOWLEDGEMENT

I would like to thank my advisor, Dr. John Shovic, for giving me an opportunity to work under him. I would like to further thank him for all the help, guidance and motivation he has provided me. I would like to thank Jack Hagemeister for all the technical guidance he has provided me. I would like to express my gratitude to Dr. David Bakken, my committee member for the time and effort involved in serving as committee members.

I would like to thank my Secure and Reliable Embedded Systems (SRES) office mates Harald, Ioanna, Geir, Rebecca and Ryan. Special thanks to Jincheng Zhang (Joe), for all the time and effort he has spent in helping me.

Finally, I wish to thank my parents and brother, whose support and confidence have carried me all these years.

MODELING OF TRANSMISSION CONTROL PROTOCOL USING ROBUST OBJECT CALCULUS

Abstract

by Kishore D. Chakravadhanula, M.S.
Washington State University
December 2000

Chair: John C. Shovic

Distributed real-time systems are growing at an exponential rate. These systems vary from being very simple using 4-bit processors to highly complex cluster systems. The need for high amount of efficiency and power saving has seen the growth of a number of communication protocols. The inherent complexity, heterogeneity, concurrency, non-determinism, security issues and real-time requirements of these distributed real-time systems hinder their development. There is a need for modeling tools or languages using which, one can mathematically validate and verify the above mentioned features of the system against the requirements.

Formal methods supply the underlying theoretical base for concurrency in object-oriented systems. Robust Object Calculus (ROC) is a formal concurrent language, which has been mechanized into Higher Order Logic (HOL). This would lend HOL semantics to ROC expressions, providing a rich provable logic to reason about distributed systems, as HOL permits the specification and verification of abstract systems with logical

statements about events and states. By reasoning with simple simulations instead of complex systems, the verification scheme becomes much more practical.

The objective of this thesis is to study the feasibility of modeling distributed real-time systems using ROC and make suggestions for over coming the deficiencies in ROC, if any, with regard to modeling these systems. As a specific example, popular protocol of the widely used TCP/IP stack, Transmission Control Protocol (TCP), is modeled using Robust Object Calculus.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The end of twentieth century has seen the growth of the Internet in a tremendous way, which is still growing in an exponential way. This has seen the emergence of distributed networks, architectures and technologies in a big way, and has brought distributed computing to every home. On the other spectrum, the rapid development in the field of ASICs and chip technology has meant that more and more gadgets used in day-to-day activities have begun to become more sophisticated, cheaper, widely used and be found everywhere. This has led to embedded systems becoming a more integral part of our lives, ever more subtly. With advances in network technologies like never before, these embedded gadgets are talking to each other, over the Internet, wireless mediums etc. One can program the Television to record a program in a Video Cassette Recorder (VCR) (which is at home) sitting at work place through the Internet or walk into a room with a Personal Digital Assistant (PDA) and automatically identify a printer and print out the driving directions! The world is changing fast.

Thus, today we have number of computing appliances, mostly embedded in nature, on many different kinds of distributed networks. There are a number of communication protocols out in the open, and more and more research is being done on them to find more power efficient and faster communicating protocols. With all these

gadget talking, using services of one another, there are numerous security issues cropping up, at an alarming rate and menacing propositions, probably faster than the growth of the distributed systems and networks themselves. Not far behind all these is the good old fundamental embedded systems requirement, reliability of the systems. The emergence of mission critical Internet enterprises such as E-Commerce and Telemedicine establishes the need for high assurance distributed computing as important area of research. And all these means enormous amount of complexity in the software, which runs them. The biggest problem of all these kinds of systems is that they suffer from inherent complexity, concurrency and non-determinism. And due to the heterogeneity of these systems, in operating systems, programming languages, underlying hardware etc, achieving interoperability becomes extremely difficult.

1.1 Motivation

Research is being done to address these concerning issues. Work done by the research community has been to address these issues separately. Some of the research has sometimes been in a very specialized manner and not being extended to encompass sister problems. Industry attitude to some of these problems has been more of "fix-and-forget" attitude. What they fail to understand is that such an attitude could compromise the security and reliability of the systems even more.

The motivation for this research comes from the theoretical work being done on the Meta-Object Operating System Environment (MOOSE) at the University of Tulsa and the work being done at Secure and Reliable Embedded Systems (SRES) Lab of Washington State University, Pullman. We, at SRES Lab, are looking at addressing all

these problems together, through the use of Formal Methods. Formal methods provide the much-needed mathematical proofs to establish or understand the problems associated with communication protocols, distributed network architecture, embedded software or the whole system software. Thus, with a proper framework where formal methods can be used to model the system under study, one can identify and address the security or reliability issues associated with the system. The research community and the industry have recognized the need and usefulness of formal methods. Formal methods have been successful in supplying the theoretical underpinnings for concurrency in object-oriented systems.

A low level concurrent language can provide formal semantics for higher-level sequential concurrent languages. These higher-level languages would be used for developing the software needed for the distributed systems or protocols. The low level concurrent language described here is called the Robust Object Calculus (ROC). ROC is part of the MOOSE framework. Upon ROC rests the newly developed Meta-Object Model (MOM), which can be used by language developers to construct concurrent object-oriented programming languages for application programmers. Since MOM has ROC semantics, it is permits verifiability.

1.2 Contributions

This thesis undertakes a study of the feasibility of using ROC to model real time systems. ROC is a very powerful and versatile formal calculus. It has the best features of Lambda and Pi-calculus, along with encapsulation, which gives it the name “robust”. As a particular case, one of the mostly popular and widely used transport layer protocol, the

Transmission Control Protocol (TCP), has been selected. A study of TCP along with its ROC model is presented in this thesis.

The research includes understanding and drawing conclusions from the ROC models of various algorithms and sub protocols of TCP. It pin points the strengths of ROC, as a tool for modeling real-time distributed systems. It also proposes a few enhancements required in ROC, to make ROC more versatile and capable of modeling real time systems. For example, there is no notion of "time" in ROC. This is very much needed to model real time protocols and systems. Finally, a briefing is provided about how the ROC models can be extended into the MOM layer.

1.3 Thesis Organization

The remainder of this thesis is as follows: Chapter II provides background information, related work and the presents Robust Object Calculus (ROC) in more detail. Chapter III introduces Transmission Control Protocol (TCP) and models of some of the various methods of connection establishment and termination. Chapter IV deals with the models of the various data communication schemes TCP uses. Chapter V deals with the timers used by TCP. Concluding remarks and recommendations for future work are given in Chapter VI.

CHAPTER II

BACKGROUND AND RELATED WORK

There are a number of complex issues related to the distributed real time systems. The issued and the work being done on them is presented. A brief history of various popular formal languages and the reason behind our choice of Robust Object Calculus as the formal language for our models is described. The remainder of the chapter introduces the ROC syntax, its capabilities and the ROC Virtual Machine (ROCVM) to test our ROC models.

2.1 Issues in Distributed Real Time Systems

There are a number of issues related with distributed systems, which are most often than not heterogeneous in nature. The hosts on a distributed systems may be running on different hardware platforms, using different programming languages, using different kinds of databases and even use different communication protocols. All these bring a host of critical issues with them. These issues are: interoperability, transparency, concurrency, integrity, reliability and security [COU01]. Interoperability between software built using different languages, on different hardware and using different communication protocols have been addressed by Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) and Microsoft's Object Linking

and Embedding (OLE). Transparency in accessing resources or code is highly desired and strived for. This is also addressed by CORBA. Distributed systems are inherently concurrent in that each component has its own processing unit [TAN95]. Co-ordination, synchronization and modeling of concurrent processes must be considered. All these fail if the system isn't *consistent* and *correct*, which forms the Integrity. Reliability is realized by satisfying criteria for *safety* and *liveness* [COU01, TAN95]. A system is safe if it doesn't "do anything bad" while liveness is concerned with the system performing the function it is supposed to. A secure system must offer *authentication*, *access control*, *data integrity* and *secure communications* [OPE92]. Authentication service guarantees the identity of subjects. This is necessary for access control, auditing and other security services. Access control does the role of authorization of the subject to access resources or avail services. Access control in distributed systems is made more difficult by disparate security models that do not interoperate. Data integrity eliminates the potential for data tampering. Distributed real time systems adds an additional issue to all the above mentioned, namely *timeliness* [BUR97]. Timeliness means, the functionality has to be met with in certain time constraints.

2.2 Formal Methods

Formal methods provide mathematical techniques for specifying and verifying system properties [CLA96, DIL90, SAI96, WIN90]. Used properly, they can help design new systems, prove their correctness, or find ambiguities and inconsistencies in existing systems. Formal specification languages are used to define and prove important properties about computational systems. The syntax of a formal specification language is

given explicitly from a syntactic domain of symbols. The semantics are derived from the domain of discourse; e.g., semantics for a language dealing with concurrent processes would consider state and event sequences among other things.

Distributed systems require formal specification languages to verify safety and liveness properties [HOA85, MIL92, MPW89]. The obvious similarities between concurrent systems and distributed systems have led to the use of concurrency formalisms for this purpose. Hoare's CSP [HOA85] was one of the first formal specification languages designed for concurrent systems. Milner developed CCS [MIL89] and the π-calculus [MIL92] for modeling concurrent processes. These formalisms have been used to specify parallel object-oriented languages [PIE95, WAL91]. Majority of the concurrent object systems and object-oriented programming languages employ π-calculus as a formal model. But, π-calculus is not well suited to modeling concurrently executing objects because it does not readily support complex message passing.

The Robust Object Calculus (ROC) [THR96] is a new formalism for concurrently executing objects. It extends Nierstrasz's Object Calculus (OC) [NIE91] by providing robust agent encapsulation. Such formalism is needed to provide the foundation for practical distributed system verification. While ROC is well suited to providing formal operational semantics for distributed object systems, a more expressive logic is required for reasoning and verification. The Higher Order Logic (HOL) [GOR93] satisfies this requirement by providing an-expressive set-theoretic higher order logic of types. HOL's type system permits the expression of abstractions. It has been applied to hardware verification and, more recently, to distributed system verification. A definitional mechanization formally defines the operational semantics using HOL expressions; i.e.,

anything that can be modeled with the formal operational semantics has a corresponding HOL semantics [ALV91, CHO95]. The advantage of this approach is that no new axioms are introduced, preventing inconsistencies. This approach provides a flexible verification scheme where systems are formally modeled using operational semantics and then verified using HOL theorem provers.

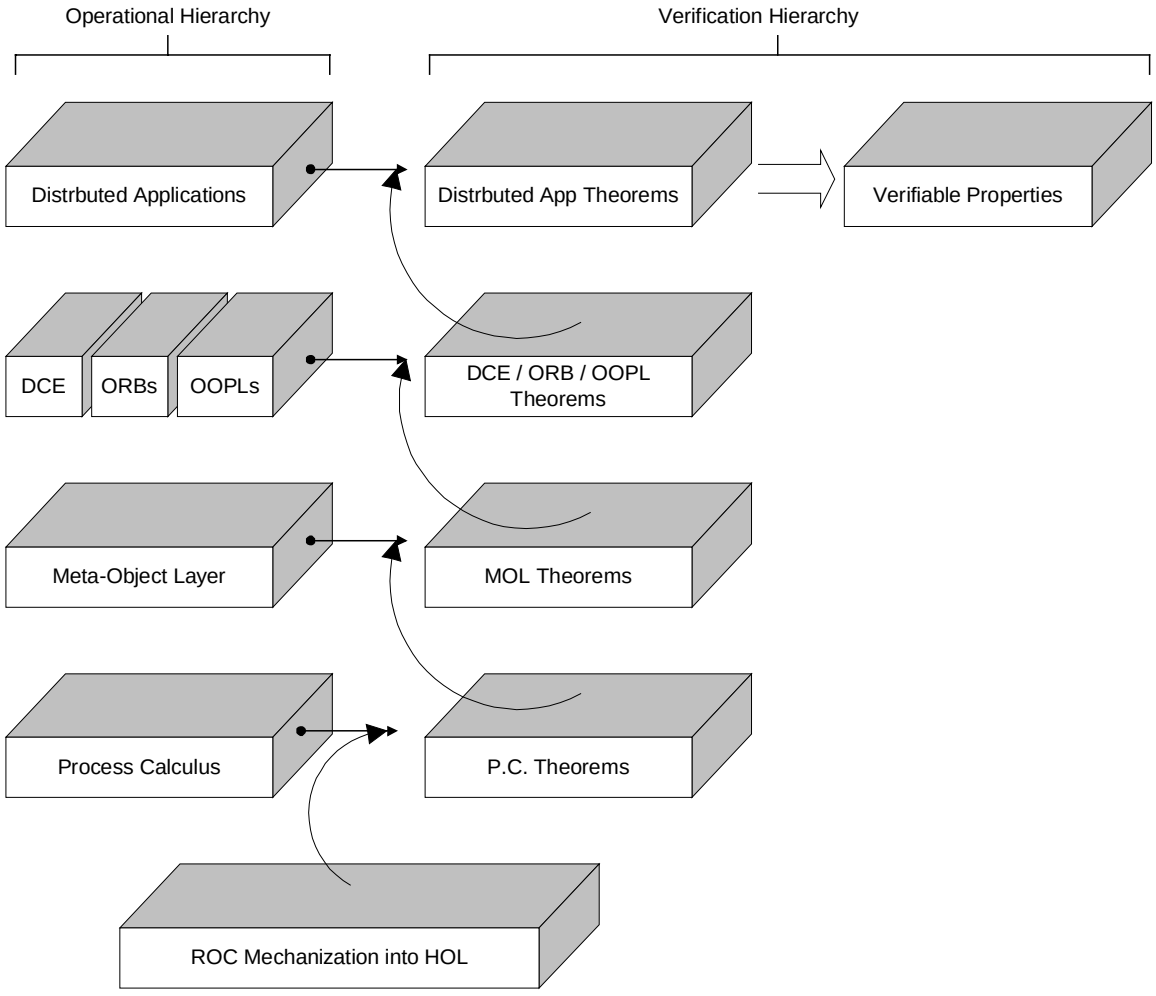

Figure 2.1: Layered Semantics of MOOSE

2.3 Robust Object Calculus

The Robust Object Calculus (ROC) is the foundation of the MOOSE architecture. It is a special process calculus tailored to modeling concurrent objects. ROC is used both as an execution model and as a basis for system verification. The Robust Object Calculus (ROC) supplies formal operational semantics to all layers in the MOOSE operational hierarchy. ROC supports complex message passing and a robust form of encapsulation for concurrently executing objects. Furthermore, the concepts of simulation and bi-similarity are definable for ROC systems, demonstrating its viability as a formal system. The operational semantics of ROC are translated into a logical system to support mechanical reasoning. Thus, ROC serves as a foundation for both the operational and the verification hierarchies [HAL97].

2.3.1 Communication in ROC

Communication in ROC is based on pattern matching. Complex message passing is achieved by matching values to patterns. Two agents may communicate when a value exposed by one matches a pattern exposed by the other. Values are nested tuples of names. Patterns are nested tuples of names and wildcards. The following notation is used: $a, b, c \in \boldsymbol{A}$ (agents); $m, n, p \in \boldsymbol{N}$ (names); $u, v \in \boldsymbol{\varsigma}$ (values); $x, y, z \in \boldsymbol{\Xi}$ (patterns) [THR97].

Definition 2.1 A value *is a name, an agent, a tuple of values, or a bindable value. A value does not contain wildcards at any level. The BNF definition of a value is:*

$$v ::= n \mid a \mid [v_1, v_2, \ldots, v_j] \mid v\#$$

Definition 2.2 A pattern is a value, wildcard, or tuple of patterns. The BNF definition of a pattern is:

$$x ::= v \mid n? \mid [x_1, x_2, \ldots, x_i]$$

A wildcard, i.e., a placeholder for a value, is denoted by $n?$ ($n \in$.). A bindable value, which can be bound to a placeholder, is denoted by $n\#$ ($n \in$.). Bindable values are those values, which are transmittable to agents with matching wildcard patterns. Values not tagged with ``#" are called unbindable values. They cannot be matched by a wildcard [THR97].

2.3.2 Pattern Matching

The matching of patterns and values is accomplished by the "~" operator. The semantics of the bindable symbol "#" are clarified in the matching rules below. Unbindable values can match bindable values, but cannot match wildcards.

Definition 2.3 The match operator is denoted by "~". The matching rules are:

$$v\# \sim v\# \qquad v \sim v$$

$$v\# \sim v \qquad v \sim v\#$$

$$v\# \sim n? \qquad v_i \sim x_i, \ \forall i$$

The last matching rule applies to a tuple of values where the tuple is either bindable or unbindable. This condition is denoted by "(#)".

Pattern matching is achieved by applying the rules given above. For example, $[m, [n]] \sim [m, [n]]$. On the other hand, $[m, n] \not\sim [m, [n]]$.

Wildcards can match bindable values. For example, $[m, [n]\#] \sim [m, [n]]$ and

$[m, [n]\#] \sim [m, p?]$. On the other hand, $[m, [n]] \not\sim [m, p?]$.

Free occurrences of the wildcard are replaced with the value inside the accepting-agent. When $v \sim x$, the notation $a\{v/x\}$ is used to denote that all wildcards in x are replaced by the corresponding sub values of v in the agent a. For example, when $v\#$ binds to $n?$ in $n? \rightarrow a$, each free occurrence of n in a is replaced by v [THR97].

2.3.3 ROC Syntax

The ROC syntax is derived from Nierstrasz's Object Calculus (OC). The symbol "#" distinguishes bindable values from unbindable values. The non-deterministic choice operator "+" from π-calculus is added along with "*", a new left-preferential composition operator. The BNF syntax is shown below [THR96, THR97].

$a ::= a \,\&\, a$	(concurrent composition)
$\mid a * a$	(left preferential composition)
$\mid n := a$	(recursion)
$\mid a + a$	(non-deterministic choice)
$\mid a \mid a$	(left preferential choice)
$\mid x \rightarrow a$	(input)
$\mid v \,\hat{}\, a$	(output)
$\mid a \,@\, v$	(application)
$\mid n \setminus a$	(new name n in a)
$\mid n$	(name)
$\mid \mathbf{nil}$	(empty agent)

The following notation is used in the syntax definition: $a, b, c \in \boldsymbol{A}$ (agents); $m, n, p \in \boldsymbol{N}$ (names); $u, v \in \boldsymbol{\varsigma}$ (values); $x, y, z \in \boldsymbol{\Xi}$ (patterns). Note, that the set of values is a proper subset of the set of patterns, i.e., V#X. The operators "&", "&", ":=", "|", "+", "$\rightarrow$", "^" and "\" are right associative. This order indicates the binding precedence from loosest to tightest. The application operator, "@", is left associative. Its binding precedence is tighter than "^" and looser than "\". Agent communication may occur when an input pattern matches an output value.

Structural congruence rules are used for manipulating expressions. While they do not represent system activity, they can be used to transform stable expressions into reducible states [THR97]. Note that := is overloaded to also mean $\stackrel{def}{\equiv}$.

1. $a \,\&\, b \equiv b \,\&\, a,\ a \,\&\, (b \,\&\, c) \equiv (a \,\&\, b) \,\&\, c$
2. $a + b \equiv b + a,\ a + (b + c) \equiv (a + b) + c$
3. $n := a \equiv a\{(n := a) / n?\}$
4. $n \setminus a \equiv a,\ n \notin fn(a)$
5. $n \setminus m \setminus a \equiv m \setminus n \setminus a$
6. $n \setminus a \square b \equiv n \setminus (a \square b),\ n \notin fn(b),\ a \mid n \setminus b \equiv n \setminus (a \mid b),\ n \notin fn(a)$

   where $\square \in \{\&, *, |, +, @\}$
7. $a \,\&\, \mathbf{nil} \equiv a,\ \mathbf{nil}@v \equiv \mathbf{nil}$
8. $n := a \equiv n' := a\{n' / n?\},\ n' \notin fn(a)$
9. $n \setminus a \equiv n' \setminus a\{n' / n?\},\ n' \notin fn(a)$
10. $x \rightarrow a \equiv x\{n' / n?\} \rightarrow a\{n' / n?\},\ n' \notin fn(x, a)$

Rules 1 and 2 address the commutativity and associativity of parallel composition and choice, respectively. Rule 3 is for expanding recursive agents. Rule 4 stipulates when a restriction can be discarded (*fn*(*a*) denotes the set of free names in *a*). Rule 5 formalizes the commutativity of restriction. Rule 6 describes scope extrusion. Scope extrusion expands the scope of a restriction to proximal agents. Rule 7 shows the effect of **nil** and any value applied to **nil**. Rules 8-10 define α-conversion for agents which substitutes globally unique names for local names. They are useful when scope extrusion is necessary.

2.3.4 Communication and Reduction Rules

Communication and reduction inference rules supply the semantics for agent expressions. Actions in the ROC universe are comprised of communication, reduction and binding. The rules are similar to those in OC except for an additional rule to handle non-deterministic choice [THR96, THR97].

Definition 2.4 Communication offers *are written as* $\overset{\alpha}{\alpha}$ *, where* α *is either v (for input) or u (for output). Reduction is written as* α *. Communication offers and reduction are defined by the following rules:*

In : $\dfrac{v \sim x}{x \to a \overset{v}{\alpha}\ a\{v/x\}}$ **Out :** $\dfrac{}{v\langle a \rangle \overset{u}{\alpha}\ a}$

Conc : $\dfrac{a \overset{\alpha}{\alpha}\ a'}{a \,\&\, b \overset{\alpha}{\alpha}\ a' \,\&\, b}$ **Choice :** $\dfrac{a \overset{\alpha}{\alpha}\ a'}{a + b \overset{\alpha}{\alpha}\ a' + b}$

ConcIf : $\dfrac{a \overset{\alpha}{\alpha} a'}{a * b \overset{\alpha}{\alpha} a' * b}$ **ConcElse :** $\dfrac{OS(a), DNO(a,\alpha), b \overset{\alpha}{\alpha} b'}{a * b \overset{\alpha}{\alpha} a * b'}$

ChoiceIf : $\dfrac{a \overset{\alpha}{\alpha} a'}{a \mid b \overset{\alpha}{\alpha} a'}$ **ChoiceElse :** $\dfrac{OS(a), DNO(a,\alpha), b \overset{\alpha}{\alpha} b'}{a \mid b \overset{\alpha}{\alpha} b'}$

Comm : $\dfrac{a \overset{v}{\alpha} a', b \overset{u}{\alpha} b'}{a \,\&\, b \alpha\ a' \,\&\, b'}$ **Apply :** $\dfrac{a \overset{v}{\alpha} a'}{a @ b \alpha\ a'}$

Left : $\dfrac{a \alpha\ a'}{a \times b \alpha\ a' \times b}, \times \in \{\&, *, |, +, @\}$ **Right :** $\dfrac{b \alpha\ b'}{a \times b \alpha\ a \times b'}, \times \in \{\&, *, |, +, \backslash\}$

Struct : $\dfrac{a \equiv b, b\ \alpha\ b', b' \equiv a'}{a\ \alpha\ a'}$

Conc provides concurrency by allowing "composed" agents to reduce independently. **Choice** allows only one of several possible activities. The **ConcIf** (**ChoiceIf**) and **ConcElse** (**ChoiceElse**) rules prefer the reduction of the left side over the right side under the concurrency (choice) operator. This is important for handling "default" actions (which is lacking in the π-calculus).

The **Else** rules use the decidable predicates *OS*(*a*) and *DNO*(*a*, α) which require the left agents be *Offer Stable* and at the same time not offer α. *OS*(*a*) states that the offers of agent *a* are stable, so that it is impossible for the agent to further internally evolve and produce new offers, even though the agent may not be completely stable. It is not necessary to look at any offers beyond guards to satisfy this predicate. It is necessary to look into unguarded applications (@), since an offer may come from within an

application if the applied value can be consumed. $DNO(a, \alpha)$ simply means that a cannot offer α.

Comm (global communication) matches complementary communication offers and reduces the system by realizing the communication. **Comm** works in concert with the **In** and **Out** rules. Apply (local communication) requires agent a to eventually accept value v. It cannot communicate externally using **Comm** until value v is consumed. If v is never consumed, then agent $a@v$ is effectively dead. The **Left** and **Right** rules allow for activity within (or under) the various operators. The **Struct** rule allows an expression to be manipulated using the structural congruence rules so that further activity can occur when a "stable" agent is structurally congruent to a reducible one. The **Struct** rule manifests symmetrical communication rules for **Conc** and **Choice** where $a \,\&\, b \equiv b \,\&\, a$ and $a + b \equiv b + a$.

2.3.5 ROC and Encapsulation

ROC extends OC by supporting robust agent encapsulation. The encapsulation property is critical to object-oriented systems. It mandates that private services and values in objects be inaccessible to external objects, thereby ensuring that objects have well-defined interfaces. From the point of view of process calculi, this necessitates a higher level of communication control between agents [THR97].

The π-calculus achieves this control by using restriction (unique-naming) to create globally unique labels for communication ports. This works because there is no wildcard matching, i.e., only identical ports can match. Unfortunately, this severely hinders its capacity for complex message passing.

Encapsulation must be guaranteed by the construction of the agent itself and not by assumptions made about external agents. ROC allows such guarantees to be made without sacrificing the ability to model complex message passing. Unbindable values in ROC make robust agent encapsulation possible [THR97].

In general, an agent is encapsulated using restriction to create an unbindable globally unique identifier for the agent. Exposed values containing the unbindable identifier are guaranteed to match only patterns containing the globally unique identifier. Such a pattern can exist outside the encapsulated agent only if the agent has transmitted its identifier to an external agent by making it bindable to an exposed value. Multiple identifiers can be used to achieve nested encapsulation, which is essential to most object systems [THR97].

2.3.6 Behavioral Equivalence

Theories of behavioral equivalence are important for system specification and verification. By constructing a simple model that is behaviorally equivalent to a more complex model, it is possible to better understand complex system behavior. Furthermore, behavioral equivalence forms the basis of sub typing, interoperability and substitutability [HAL97].

Behavioral equivalence theories are built on formal definitions of simulation and bisimulation. Two basic forms of simulation are defined. One is a strict form of simulation that considers all actions. The other considers only the *observable behavior* of *agents*. The observable behavior of an agent refers to any communication involving that

agent and its environment (which includes all agents external to the agent). Internal communications are not considered observable [HAL97].

The notion of observational communication is formalized by defining $P \overset{\alpha}{\Rightarrow} Q$, which states that P reduces to Q after some observable communication:

$$P \overset{\alpha}{\Rightarrow} Q \overset{def}{=} P\ \alpha\ {*}\alpha\ \alpha\ {*}\ Q$$

The notion "$\alpha\ *$" indicates an arbitrary number of internal communications (reductions).

Relation predicates of type *S* are predicates on ROC expression pairs. Strong simulation and weak simulation are predicates that operate on relation predicates of type *S*. Strong simulation mandates that each potential communication with agent *P* has a counterpart in *Q*, i.e., *P* is simulated by *Q*, for each *P* and *Q* in *S*. Weak simulation is based on observable communications. The definitions below formalize notions of strong simulation and weak simulation [HAL97]:

$$S : ROC \rightarrow ROC \rightarrow Bool$$

$$StrongSimulation : (ROC \rightarrow ROC \rightarrow Bool) \rightarrow Bool$$

$$WeakSimulation : (ROC \rightarrow ROC \rightarrow Bool) \rightarrow Bool$$

$$StrongSimulation\ S \overset{def}{=} \forall P, Q.S\ P\ Q \supset (\forall P'.P\ \alpha\ P' \supset \exists Q'.Q\ \alpha\ Q') \wedge$$

$$(\forall \alpha.\forall P'.P\ \overset{\alpha}{\alpha}\ P' \supset (\exists Q'.Q\ \overset{\alpha}{\alpha}\ Q' \wedge S\ P'\ Q'))$$

$$WeakSimulation\ S \overset{def}{=} \forall P, Q.S\ P\ Q \supset \forall \alpha.\forall P'.P \overset{\alpha}{\Rightarrow} P' \supset$$

$$(\exists Q'.Q \overset{\alpha}{\Rightarrow} Q' \wedge S\ P'\ Q').$$

The predicates *StronglyBisimilar* and *WeaklyBisimilar* convey analogous notions of bisimilarity. The definitions below are patterned after notions of bisimulation found in [MIL89].

$$\mathit{StronglyBisimilar} : \mathit{ROC} \rightarrow \mathit{ROC} \rightarrow \mathit{Bool}$$

$$\mathit{WeaklyBisimilar} : \mathit{ROC} \rightarrow \mathit{ROC} \rightarrow \mathit{Bool}$$

$$\mathit{StronglyBisimilar}\ \mathrm{P}\ \mathrm{Q} \stackrel{def}{=} \exists S.S\ P\ Q\ \Lambda$$

$$\mathit{StrongSimulation}\ S\ \Lambda\ \mathit{StrongSimulation}\ (\lambda x\ y.S\ y\ x)$$

$$\mathit{Weakly\ Bisimilar}\ P\ Q \stackrel{def}{=} \exists S.S\ P\ Q\ \Lambda$$

$$\mathit{WeakSimulation}\ S\ \Lambda\ \mathit{WeakSimulation}\ (\lambda x\ y.S\ y\ x)$$

These definitions facilitate mechanical reasoning about the behavioral equivalence of ROC systems. Mechanizing ROC within a more expressive formal system, e.g., Higher Order Logic (HOL), expands the possibilities for system verification and providing a rich logic for reasoning about distributed object systems. HOL permits the specification and verification of abstract system properties, which are not suitably modeled, but more adequately represented with logical statements about events and states. By reasoning with simple simulations instead of complex systems, the verification scheme becomes much more practical. Models and languages given ROC semantics inherit this potential for verification [HAL97].

2.4 ROC Virtual Machine

A ROC virtual machine (ROCVM) is necessary to not only test the accuracy of the models developed in ROC. The ROCVM is designed to be a platform independent

distributed execution model for ROC expressions. From a user's point of view, ROCVM is a single machine. The distributed virtual machine will consist of many ROC nodes that may be running on numerous platforms. Each incarnation of a ROC node is tied to a particular ROC virtual machine name so that multiple nodes with different names may be running on the same machine. This makes it possible for multiple ROCVMs to run on the same set of machines without name space conflicts [THR97].

ROC nodes are the heart of the ROC virtual machine since they are where reduction of ROC expressions occurs. Communication offers are transferred between ROC nodes, effectively linking the nodes into one large reduction machine for ROC expressions. Most of the ROC design and implementation efforts have concentrated on the reduction engine found in each node. ROCVM accepts agent definition as input and tries to resolve all references. All reductions are performed on the agent trees until it reaches a stable state. A stable state occurs when no communications are possible. While executing ROC expressions, ROCVM simulates the execution of ROC-modeled applications in distributed environments [THR97].

The ROCVM consists of visualization module too. This module provides a graphical and intuitive way to observe the systems modeled with ROC. Visual representations of ROC entities, such as agents, values and names are created in both Tree Graph View and Tree Map View. The Tree Map View presents a more natural view of ROC agent tree [ZHA00]. Figure 2.2 and Figure 2.3 depict the Tree Graph View and Tree Map View respectively.

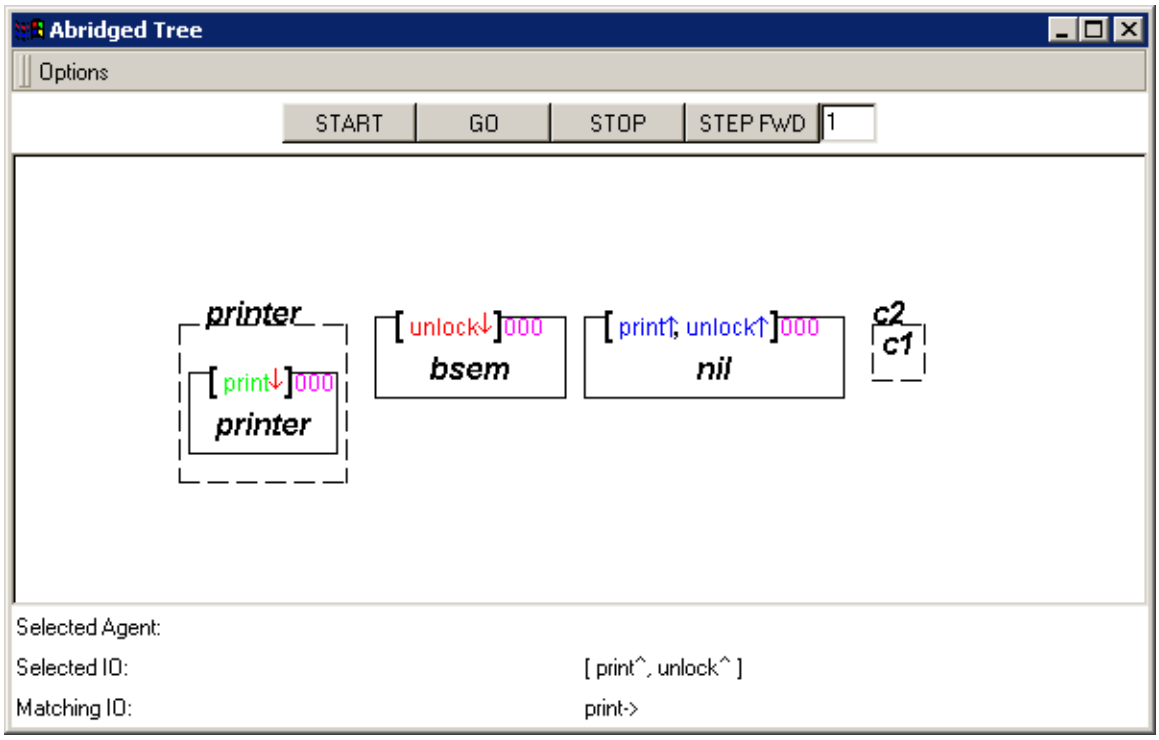

Figure 2.2: Tree Map View of ROC System

Figure 2.3: System View of ROC System. Tree Graph View is on the left pane.

The Tree Graph View is just like common hierarchical file structure viewer, which is widely used in modern operating systems. The root system agent is at the top left and all descendents go down and to the right. The Tree Map View is more natural view. Here, the screen is recursively divided into rectangular boxes. Thus, the box corresponding to a high level node contains the boxes corresponding to its descendants [ZHA00].

2.5 Numbers in ROC

We would encounter the use of numbers while doing the models of TCP. Numbers are inherently not supported as a primitive type in ROC. So, they have to be encoded so that they can be viewed as any other kind of agents. We follow the approach used in lambda calculus [BAR84] to model numbers and arithmetic.

First, we need Boolean values encoded as agents. Booleans are used in practice for making a choice between two alternatives, so:

true := (?[a?, b?] -> a);

false := (?[a?, b?] -> b);

With this interpretation, we can also define:

neg := (?a? -> a @ #[false#, true#])

and := (?[a?, b?] -> a @ [b#, a#])

or := (?[a?, b?] -> a @ #[a#, b#])

The encoding of natural numbers differs only slightly from the standard one. Instead of viewing an expression such as 1 + 2 as a function applied to the values 1 and 2, we interpret it as syntactic sugar for applying the tuple (+, 2) to the agent 1, i.e., 1@(+, 2). That is, + is not a function but merely a name serving as a message selector. This allows

us to later define other kinds of agents that are not numbers, but that also understand messages of the form (+, a), exactly as one would when defining new classes in an object-oriented programming language.

We can now encode the natural numbers as an abstraction over two values: a Boolean value indicating if the number is 0, and the number's predecessor, if any:

```
nat := (?[z?, p?] -> (  ?iszero -> z |
                        ?pred -> p |
                        ?succ -> nat @ #[false#, null#] |
                       )
       );
0 := (nat @ #[true#, iszero])
1 := (nat @ #[false#, iszero])
```

It is easy to see that *0 @ iszero -> true* and *0 @ succ @ iszero -> false.*

CHAPTER III

TCP CONNECTION ESTABLISHMENT AND TERMNATION

Transmission Control Protocol is by the far the most popular communication protocol at the transport layer. It is widely used, especially in the Internet, besides in applications like Telnet, Rlogin, FTP, SMTP etc. TCP provides a *reliable, connection-oriented, byte stream, full-duplex* transport layer service. It can be described as a sliding window protocol without selective or negative acknowledgements [STE99]. TCP packetizes the user data into segments, sets a timeout any time it sends data, acknowledges data received by the other end, reorders out-of-order data, discards duplicate data, provides end-to-end flow control, and calculates and verifies a mandatory end-to-end checksum.

3.1 TCP Header

Figure 3.1 shows the format of Transmission Control Protocol (TCP) header. It is normally 20 bytes long unless options are present. Each TCP segment contains the source and destination *port number* to identify the sending and receiving applications. These values along with the source and destination IP address in the IP header uniquely identify each connection. The combination of IP address and port number represents a *socket*. Thus, a *socket pair* specifies the two end points that uniquely identify each TCP connection in an internet. *Sequence number* identifies the byte in the stream of data from

the sending TCP to the receiving TCP that the first byte of data in this segment represents. When a new connection is being established, the SYN flag is turned on and the sequence number field contains the *initial sequence number*. If the ACK flag is turned on, then, the *acknowledgement number* contains the next sequence number that the sender of the acknowledgement expects to receive, which is sequence number plus one. The *header length* gives the length of the header in 32-bit words.

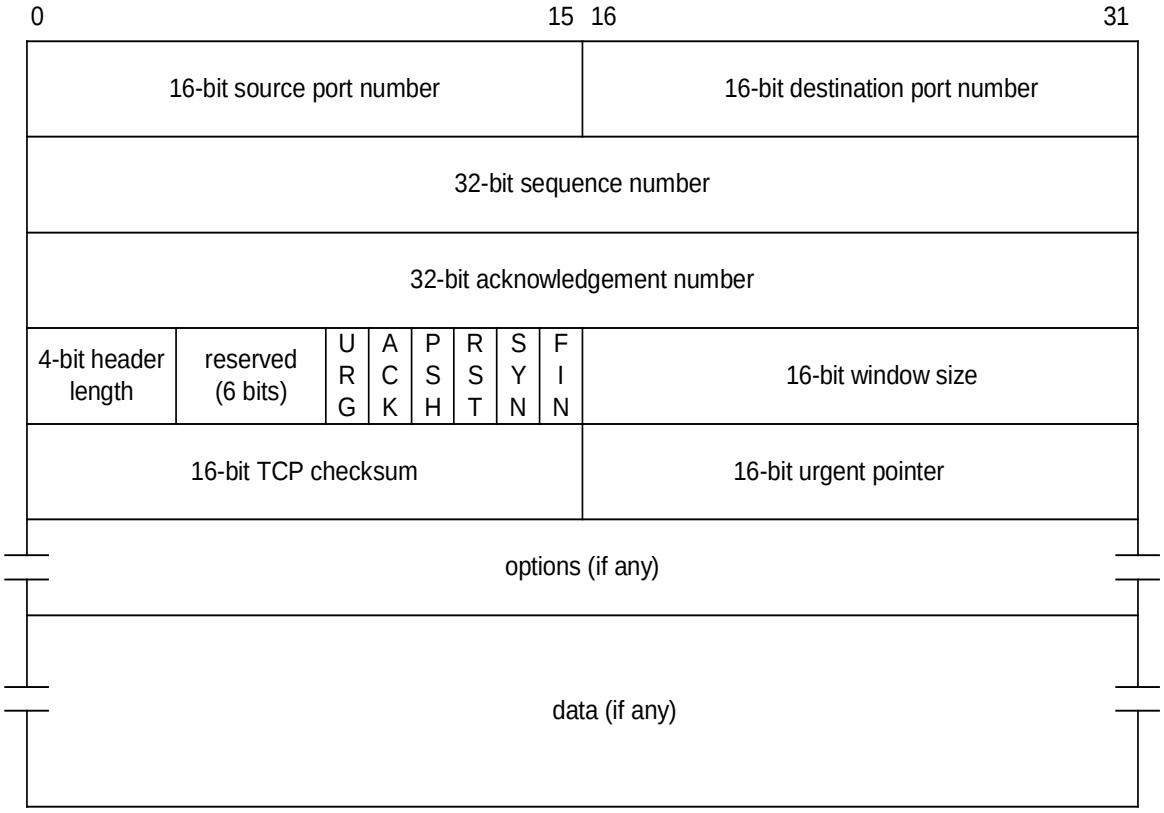

Figure 3.1: TCP Header

There are six flags bits. URG flag is set to indicate that the *urgent pointer* is valid. ACK flag is set to indicate that the *acknowledgement number* is valid. A set PSH flag indicates that the receiver should pass this data to the application immediately, while a set RST

flag resets the connection. SYN flag is set to synchronize sequence numbers to initiate a connection and FIN flag is set to indicate that the sender is finished with sending data. The number of bytes the receiver is willing to receive is advertised using the *window size*. It is used for providing flow control. The *checksum* is computed on the TCP header and TCP data, and sent along with the packet, which is verified by the receiver. *Urgent pointer* is the positive offset that must be added to the sequence number field of the segment to yield the sequence number of the last byte of urgent data. Among the *options*, the most common option field is *Maximum Segment size* (*MSS*), which describes the maximum segment sized segment the sender wants to receive.

The ROC representation of TCP is shown below:

TCPMessage (srcPort, dstPort, seqNum, ackNum, hdrLen, rsvd, FLAGS,

wndSize, chksum, urgPtr, OPTIONS, data) :=

#[srcPort#, dstPort#, seqNum#, ackNum#, hdrLen#, rsvd#, FLAGS#, wndSize#,

chksum#, urgPtr#, OPTIONS#, data#] ^ nil;

FLAGS := [URG# | null#, ACK# | null#, PSH# | null#, RST# | null#,

SYN# | null#, FIN# | null#]#;

OPTIONS := [EOPL#, NOP#, MSS#, WSF#, TS#]#;

EOPL := [kind#]#;

NOP := [kind#]#;

MSS := [kind#, len#, mss#]#;

WSF := [kind#, len#, shiftcount#]#;

TS := [kind#, len#, tsValue#, tsEchoReply#]#;

3.2 TCP Connection Establishment Protocol

TCP has to establish connection before data exchange can take place. To establish a TCP connection:

1. The requesting end (normally called the *client*) sends a SYN segment specifying the port number of the *server* that the client wants to connect to, and the client's *initial sequence number* (ISN).
2. The server responds with its own SYN segment containing the server's initial sequence number. The server also acknowledges the client's SYN by ACKing the client's ISN plus one. A SYN consumes one sequence number.
3. The client must acknowledge this SYN from the server by ACKing the server's ISN plus one.

These three segments complete the connection establishment. This is often call the *three-way handshake* [STE99]. The side that sends the first SYN is said to perform an *active open* and the side, which receives this SYN and sends the next SYN, performs a *passive open*. When each side sends its SYN to establish the connection, it chooses an initial sequence number for that connection. The ISN should change over time, so that each connection has a different ISN. RFC 793 specifies that the ISN should be viewed as a 32-bit counter that increments by one every 4 microseconds [POS81]. The purpose of this sequence numbers is to prevent packets that get delayed in the network from being delivered later and then misinterpreted as part of an existing connection.

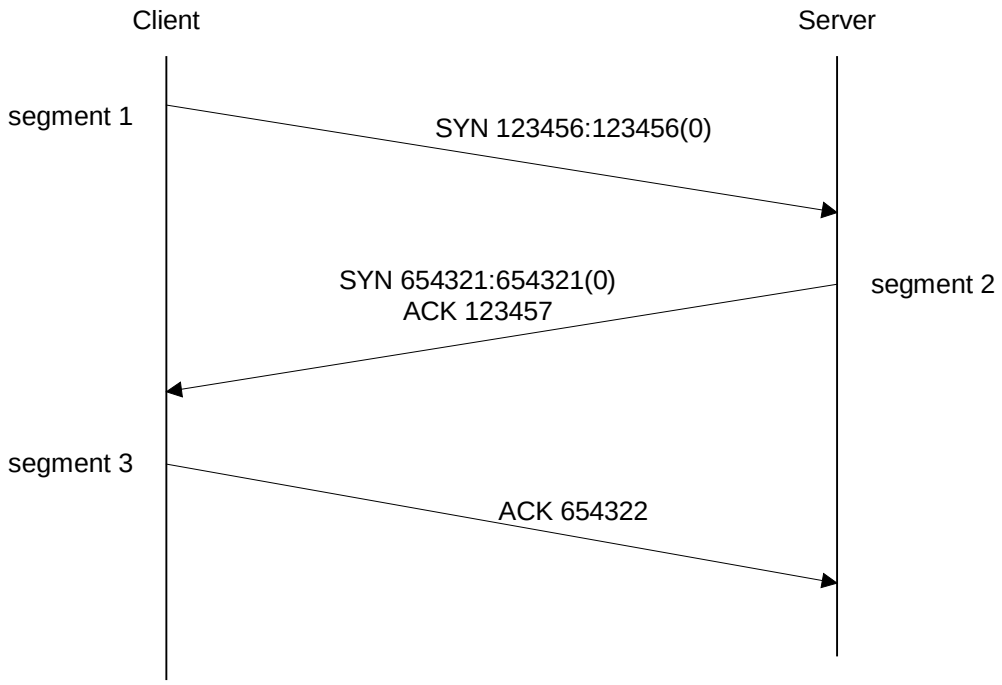

Figure 3.2: TCP Connection Establishment

System := (Client & Server);

Client := (#[cPort#, sPort, cSeqNum#, null#, [null#, SYN#]] ^ cSynSent);

cSynSent := (?[sPort, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb);

Server := (?[cPort?, sPort, cSeqNum?, cAckNum?, [null, SYN]] ->

#[sPort#, cPort#, sSeqNum#, cSeqNum1#, [ACK#, SYN#]] ^ sSynRcvd);

sSynRcvd := (?[cPort?, sPort, cSeqNum1?, sSeqNum1, [ACK, null]] -> sEstb);

The above is the ROC model of the TCP connection establishment. Only those fields and flags, which make are required for the understanding of the connection establishment are

shown here. So, we have only the port numbers, which determine the application on a specified host, sequence numbers and acknowledgement numbers, ACK and SYN flags only. The Client outputs a SYN segment and moves to SYN_SENT (SYN segment sent) state. A SYN segment consists of an initial sequence number with the SYN flag set. The server receives the SYN segment, moves to SYN_RCVD (SYN segment received) state, and acknowledges the receipt of the segment with SYN-ACK segment. The SYN-ACK segment has both the SYN and ACK flags set, the server's initial sequence number and client's sequence number plus one (represented by *cSeqNum1*). The client receives the server's SYN-ACK segment and responds with an ACK segment, which contains the client's sequence number as the initial sequence number plus one and the acknowledgement number as server's initial sequence number plus one. The client sends this packet and moves to the ESTB (connection established) state. When the server receives the ACK segment, it also moves to the ESTB (connection established) state.

From this model we can talk a few things about the TCP protocol's connection establishment. It is observable from (*cPort?*, *sPort*) pair that the server is ready to receive connections from any client trying to contact it (as it should be). On the other hand, the application on the client receives the segments only from the specified server talking with the specified *instance* of the application (represented by (*sPort*, *cPort*)). Thus, the connection establishment is secure, given that the underlying network protocol is secure. This is so, because the TCP checks only the port number. A segment intended for an application in a different machine using the same port number as this machine, could receives segments intended for some one else. The connection establishment is a reliable since it is possible for the server or client to not only confirm if the other end has

received the segments, but also keep track of any missing packets with the help of acknowledgement numbers (this would be discussed in more detail in TCP's data communication).

One can see the power of ROC in that it can use agents to encapsulate the state, events in the form of values or agents can be communicated to drive from one state to another. In ROC, agents themselves can be communicated. This is possible due to the encapsulation capability of ROC, which is a very useful property to model distributed concurrent systems.

3.3 TCP Connection Termination Protocol

While it takes three segments to establish a connection, it takes four to terminate a connection. This is caused by TCP's *half-close*. Since a TCP connection is full duplex, each direction must be shut down independently. The rule is that either end can send a FIN when it is done sending data. When a TCP receives a FIN, it must notify the application that the other end has terminated that direction of data flow. The receipt of FIN only means that there will be no more data flowing in that direction. A TCP can still receive data after receiving a FIN. While it is possible to take advantage of this half-close, in practice few applications use it. The normal scenario is shown in the figure 3.3 [STE99].

The end that issues the first FIN is said to perform the *active close* and the other end, the one, which receives the FIN, performs the *passive close.* Usually, it is the client, which performs the active close. When the server receives the FIN it sends back an ACK of the received sequence number plus one (segment 2). A FIN consumes a sequence

number just like a SYN. The server then sends a FIN (segment 3), which the client TCP ACKs by incrementing the received sequence number by one (segment 4).

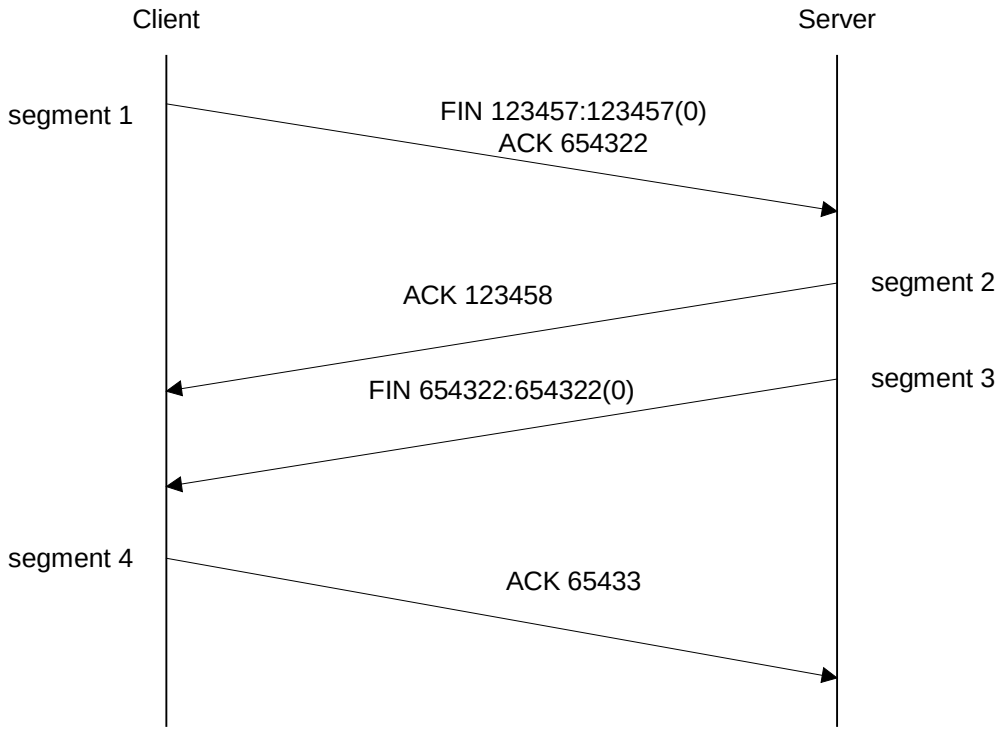

Figure 3.3: TCP Connection Termination

The ROC model of the connection termination can be as shown.

System := (cEstb & sEstb);

cEstb := (#[cPort#, sPort, cSeqNum#, null#, [null#, FIN#]] ^ cFinWait1);

cFinWait1 := (?[sPort, cPort, sSeqNum?, cSeqNum1, [ACK, null]] ->

?[sPort, cPort, sSeqNum1?, cSeqNum1, [null, FIN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum2, [ACK, null#]] ^

cTimeWait);

cTimeWait := (nil);

sEstb := (?[cPort?, sPort, cSeqNum?, cAckNum?, [null, FIN?]] ->

#[sPort#, cPort, sSeqNum#, cSeqNum1, [ACK#, null#]] ^

sCloseWait);

sCloseWait := (#[sPort#, cPort, sSeqNum1#, cSeqNum1, [null#, FIN#]] ^ sLastAck);

sLastAck := (?[cPort, sPort, cSeqNum?, sSeqNum2, [ACK, null]] -> nil);

The connection termination representation is very similar to the TCP connection establishment representation, from a ROC modeling perspective. As with connection establishment, there is authentication of the ports, there is authorization based on the current states at the client and server sides, flags and acknowledgement numbers.

3.4 Timeout of Connection Establishment

There are several instances where the connection cannot be established. We know that the client's TCP would keep sending SYN segments to try to setup the connection. The client's TCP cannot keep sending SYN segments forever. It would timeout after a few attempts. For most Berkeley derived systems, this time limit, for keep trying to establish a connection before giving up, has been set for 75 seconds.

System := (Client & Timer & Channel);

Client := (#[cPort#, sPort, cSeqNum#, null#, [null#, SYN#]]# ^ cSynSent);

cSynSent := ((?[sPort?, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->

```
                    #[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb)
            | (?Timeout -> ?Timeout -> ?Timeout -> nil)
        );
Timer := (#Timeout ^ #Timeout ^ #Timeout ^ Timer);
Channel := (?PKT? -> Channel);
```

Above is the ROC model of timeout of TCP connection. We have a client who is trying to establish a connection, a timer and the channel. The timer is TCP's 500ms timer. The client sends a SYN segment, sets the timer and waits for an ACK. If the ACK doesn't return in 6 seconds, again a SYN segment is sent. This time the client waits for about 24 seconds for the ACK segment. If it doesn't get the ACK, then it would resend the SYN again and wait for 48 more seconds. But, usually, the client ends the connection establishment trials after a total of 75 seconds (implementation dependent).

Here, while modeling the connection establishment timeout, we run into the first problem of ROC when it comes to model real time systems, the absence of a "notion of time". To get around this, we had to model the timeout in the manner shown above. So, if ROC is to be successful in modeling of real time systems, it has to be modified to include "time" into it.

3.5 TCP State Transition Diagram

Shown in figure 3.4 is the state transition chart of a TCP client or server. A normal transition path of TCP client is showed using dark solid arrows and the TCP server's transition path using dark dashed arrows. The two transitions leading to the

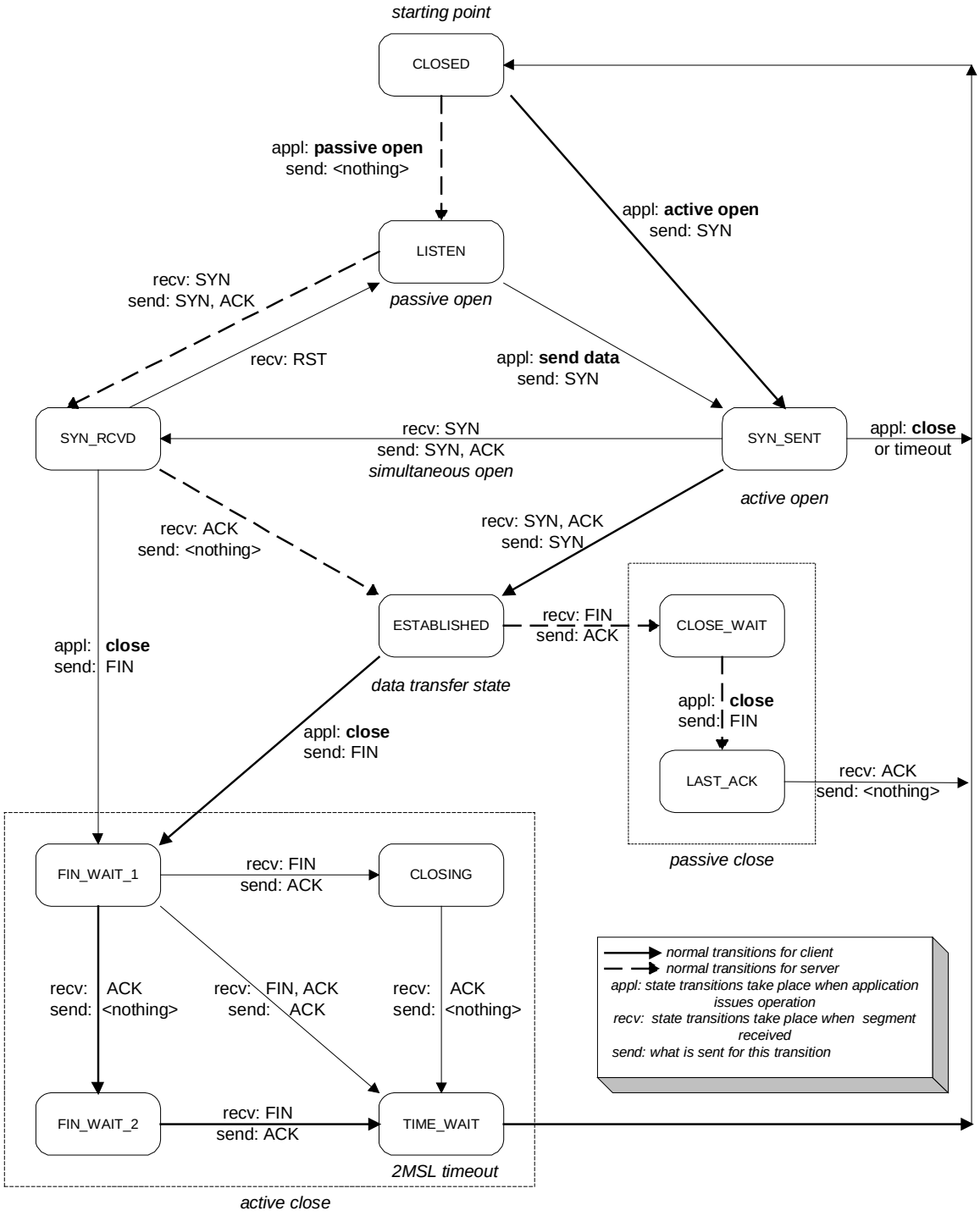

Figure 3.4: TCP State Transition Diagram

ESTABLISHED state correspond to opening a connection, and the two transitions leading from the ESTABLISHED state are for the termination of a connection. The ESTABLISHED state is where data transfer takes place between two ends in both directions.

The state transition from LISTEN to SYN_SENT though legal is not supported by many TCP implementations. The transition from SYN_RCVD back to LISTEN is valid only if the SYN_RCVD state was entered from the LISTEN state (the normal state), not from the SYN_SENT state (a simultaneous open). This means, if we perform a passive open (enter LISTEN), receive a SYN, send a SYN with an ACK (enter SYN_RCVD), and then receive a reset instead of an ACK, the end point returns to the LISTEN state and waits for another connection request to arrive.

A server would do a passive open and go to the LISTEN state, where it would wait for client connection requests. A normal client would perform an active open by sending a SYN to the server and move to the SYN_SENT. The server would reply with its own SYN and ACK of the client's SYN it received, and move to SYN_RCVD state. When the client receives the ACK from server and the server's SYN, it would acknowledge server's SYN with an ACK and move to the ESTABLISHED state. The server would also move to the ESTABLISHED state once it receives the client's ACK. Now, they can exchange data.

When the client is done with sending or requesting information, and it is time for tearing down the connection, it would send a FIN segment, indicating to the server it's intent of termination of the connection, and in the process move to the FIN_WAIT_1 state. Usually, the server would also be willing to terminate the connection. So, when it

receives the FIN, it would send an ACK back to the client and move to the CLOSE_WAIT state. The server would then send its FIN segment and move to LAST_ACK state, where it would be waiting for the "last" ACK from the client, in this connection. When it receives it would move back to the CLOSED or LISTEN state. On the other side, the client who is waiting in the FIN_WAIT_1 state, would move to the FIN_WAIT_2 state, when it receives the server's ACK sent in reply to client's FIN segment. When the client receives the server's FIN, it would move to the TIME_WAIT state, where it would wait for a minimum period of 2*MSL* (*maximum segment life*).

Below is the ROC model of the TCP state transition. Here again, only those flags and fields, which are relevant for explaining the concept are shown.

System := (Client & Server);

Client := (#[cPort#,sPort,[null#,SYN#,null#]] ^ cSynSent & cMsgHdlr);

cMsgHdlr := (?[sP?,cPort,[A?,S?,F?]] -> (

cSynSent @ #[sP#,cPort,[A#,S#,F#]] |

cEstb @ #[sP#,cPort,[A#,S#,F#]] |

cFinWait1 @ #[sP#,cPort,[A#,S#,F#]] |

cFinWait2 @ #[sP#,cPort,[A#,S#,F#]] |

cTimeWait @ #[sP#,cPort,[A#,S#,F#]] |

cClosing @ #[sP#,cPort,[A#,S#,F#]]

) & cMsgHdlr);

```
cSynSent := (?[sPort?,cPort,[ACK,SYN,null]] ->
                #[cPort#,sPort,[ACK#,null#,null#]] ^ cEstb);
cEstb := (#[cPort#,sPort,[null#,null#,FIN#]] ^ cFinWait1);
cFinWait1 := ( (?[sPort?,cPort,[ACK,null?,null?]] -> cFinWait2) +
        (?[sPort?,cPort,[null?,null?,FIN]] ->
                [cPort#,sPort,[ACK#,null#,null#]] ^ cClosing) +
        (?[sPort?,cPort,[ACK,null?,FIN]] ->
                 #[cPort#,sPort,[ACK#,null#,null#]] ^ cTimeWait) );
cFinWait2 := (?[sPort?,cPort,[null?,null?,FIN]] ->
                        #[cPort#,sPort,[ACK#,null#,null#]] ^ cTimeWait);
cClosing := (?[sPort?,cPort,[ACK,null?,null?]] -> cTimeWait);
cTimeWait := cClosed;
cClosed := (nil);


Server := (sListen & sMsgHdlr);


sMsgHdlr := (?[cP?,sPort,[A?,S?,F?]] -> (
        sSynRcvd @ #[cP#,sPort,[A#,S#,F#]] |
        sListen @ #[cP#,sPort,[A#,S#,F#]] |
        sEstb @ #[cP#,sPort,[A#,S#,F#]] |
        sCloseWait @ #[cP#,sPort,[A#,S#,F#]] |
```

sLastAck @ #[cP#,sPort,[A#,S#,F#]]

) & sMsgHdlr);

sListen := (?[cPort?,sPort,[null?,SYN,null?]] ->

#[sPort#,cPort,[ACK,SYN,null#]] ^ sSynRcvd);

sSynRcvd := (?[cPort?,sPort,[ACK,null?,null?]] -> sEstb);

sEstb := (?[cPort?,sPort,[null?,null?,FIN]] ->

#[sPort#,cPort,[ACK#,null#,null#]] ^ sCloseWait);

sCloseWait := (#[sPort#,cPort,[null#,null#,FIN#]] ^ sLastAck);

sLastAck := (?[cPort?,sPort,[ACK,null?,null?]] -> sClosed);

sClosed := (nil);

There is a client and a server in the system. A server forks two process, one, which listens on a port for incoming TCP segments (*sListen*) and the other, which handles the incoming segments (*sMsgHdlr*,) received from *sListen*. On the client side too there is a segment handler (*cMsgHdlr*). This agent receives the segments for client and drives the state transition on the client side. The above is a complete representation of the TCP connection establishment and termination.

3.6 2MSL Wait State

The TIME_WAIT state, also known as the 2MSL wait state, is the maximum amount of time any segment can exist in the network before being discarded. This time is bounded, because TCP segments are transmitted as IP datagrams, and IP datagrams have a Time-

To-Live (TTL) field that limits its lifetime. RFC 793 specifies the MSL as 2 minutes [POS81]. The rule for the MSL value implementation is: when TCP performs an active close, and sends the final ACK, that connection must stay in the TIME_WAIT state for twice the MSL. This lets TCP resend the final ACL in case this ACK is lost (in which case, the other end will time out and retransmit its final FIN) [STE99]. Another effect of this 2MSL wait is that while the TCP connection is in the 2MSL wait, the socket pair defining that connection (client IP address, client port number, server IP address, server port number) cannot be reused. That connection can only be reused when the 2MSL wait is over.

The implication of the above statement is visible in case of clients and not servers. If a client is terminated and restarted, it wouldn't have problems, because clients use any available port. The same is not true with servers, which bind to well-known ports. So, if a server is terminated and restarted immediately, then it would take from 1 to 4 minutes before the server can be restarted. The 2MSL provides protection against delayed segments from an earlier incarnation of a connection from being interpreted as part of new connection that uses the same local and foreign IP address and port numbers. But this works only if a host with connections in the 2MSL wait does not crash. If, the host with ports in the 2MSL wait crashes and reboots within MSL seconds (many embedded systems can do this), and immediately establishes new connections using the same local and foreign IP addresses and port numbers corresponding to the local ports that were in the 2MSL wait before crash, then delayed segments from previous connection can be misinterpreted as those belonging to the new connection. To protect from this scenario,

RFC 793 states that TCP should not create any connections for MSL seconds after rebooting, which is called the *quiet time* [POS81].

3.7 Reset Segments

RST flag in the TCP header stands for “reset”. In general, a reset is sent by TCP whenever a segment arrives that doesn’t appear correct for the referenced connection (a connection specified by the destination IP address and port number, and the source IP address and port number, which is nothing but the socket [POS81]). A common cause for generating a reset is when a connection request arrives and no process is listening on the destination port.

System := (Client & Server);

Client := (#[cPort#, somePort#, cSeqNum#, null#, [null#, SYN#]]# ^ cSynSent);

cSynSent := (

(?[sPort?, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb) |

(?[sPort?, cPort, anySeqNum?, anyAckNum?, [RST]] -> nil)

);

cEstb := (#OUT# ^ nil);

Server := (

(?[cP?, sPort, cSeqNum?, cAckNum?, [null, SYN]] ->

#[sPort#, cP#, sSeqNum#, cSeqNum1#, [ACK#, SYN#]] ^ sSynRcvd) |

(?[cP?, nonExistentPort?, anySeq?, anyAck?, [A?, SYN?]] ->

#[nonExistentPort#, cPort, null#, null#, [RST#]] ^ Server)

);

sSynRcvd := (?[cPort?, sPort, cSeqNum1?, sSeqNum1, [ACK, null]] -> sEstb);

sEstb := (#OUT# ^ nil);

Another case when a reset would be sent would when a connection is abruptly ended. This is called *abortive release* (as against the *orderly release* in the normal connection termination). In this case, any queued data is thrown away and a reset is immediately. An example would be, when you press CTRL-D doing your telnet session.

3.8 TCP Simultaneous Open

It is possible, although improbable, for two applications to both perform an active open to each other at the same time. Each end must transmit a SYN, and the SYNs must pass each other on the network. It is also required that each application has a local port number which is well known to the other end [STE99]. This is called *simultaneous open*. TCP is designed to handle simultaneous opens and the rule is that only one connection results from this, not two connections.

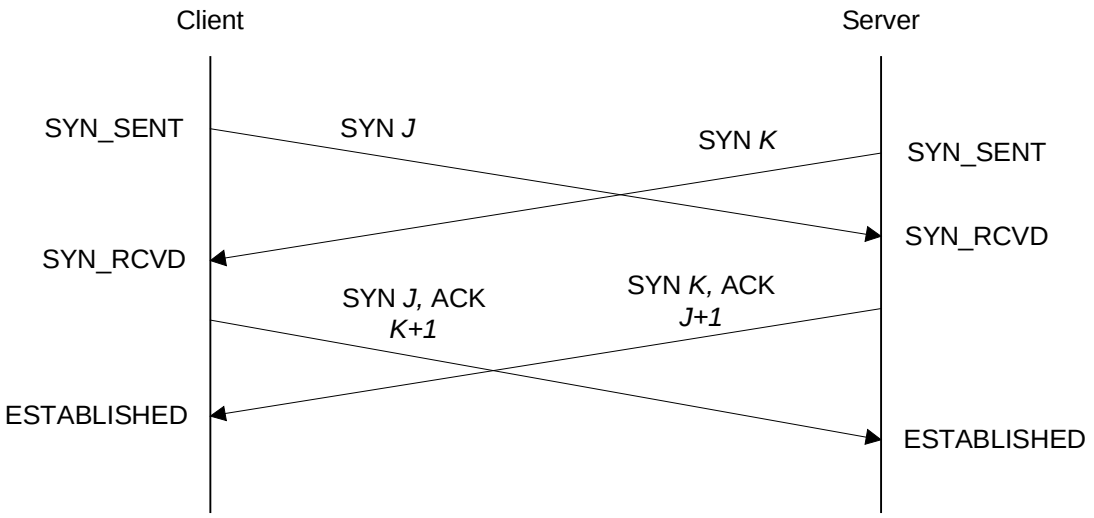

Figure 3.5: TCP Simultaneous Open

In a simultaneous open, both ends send a SYN at about the same time, entering the SYN_SENT state. When each end receives the SYN, the state changes to SYN_RCVD and each end resends the SYN and acknowledges the received SYN. When each end receives the SYN plus the ACK, the state changes to ESTABLISHED.

System := (Client & Server);

Client := (#[cPort#, sPort, cSeqNum#, null#, [null#, SYN#]]# ^ cSynSent);

cSynSent := ((?[sPort?, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb)

| (?Timeout -> ?Timeout -> ?Timeout -> nil));

Server := Client;

3.9 TCP Simultaneous Close

Simultaneous close results when both ends perform an active close. TCP allows simultaneous close. Both ends go from ESTABLISHED to FIN_WAIT_1 when the application issues the close. This causes both FINs to be sent, and they probably pass

each other on the network. When the FIN is received, each end transitions from FIN_WAIT_1 to the CLOSING state, and each state sends its final ACK. When each end receives the final ACK, the state changes to TIME_WAIT.

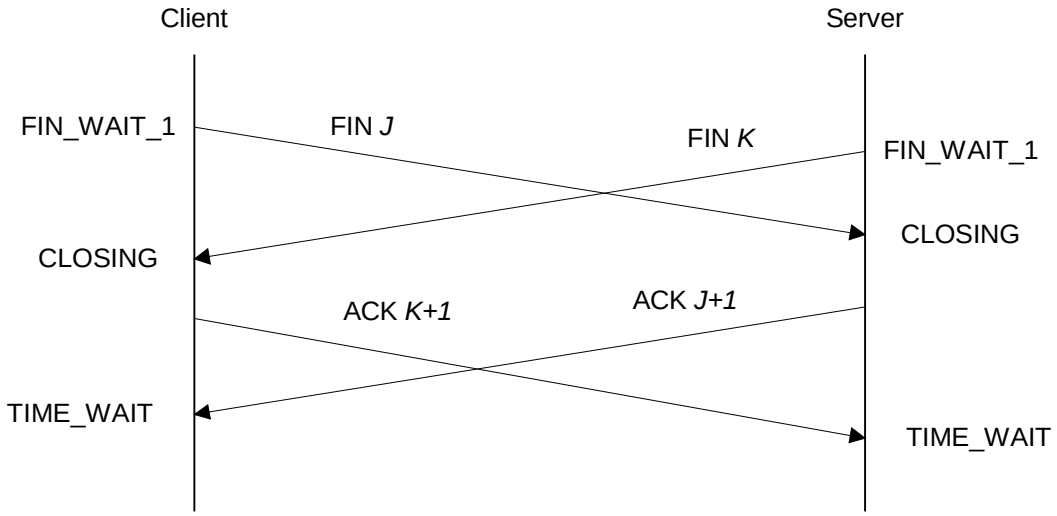

Figure 3.6: TCP Simultaneous Close

System := (cEstb & sEstb);

cEstb := (#[cPort#, sPort, cSeqNum#, null#, [null#, FIN#]] ^ cFinWait1);

cFinWait1 := (?[sPort, cPort, sSeqNum?, cSeqNum1, [ACK, null]] ->

?[sPort, cPort, sSeqNum1?, cSeqNum1, [null, FIN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum2, [ACK, null#]] ^

cTimeWait);

cTimeWait := (nil);

sEstb := (cEstb);

CHAPTER IV

DATA FLOW IN TCP

Studies of TCP traffic [CAC91] usually find that on a packet count basis about half of all TCP segments contain bulk data (FTP, electronic mail, Usenet news) and other half contain interactive data (Telnet and Rlogin, for example). On a byte-count basis the ratio is around 90% bulk data and 10% interactive, since bulk data segments tend to be fill sized (normally 512 bytes of user data), while interactive data tends to be much smaller. (The above-mentioned study found that90% of Telnet and Rlogin packets carry less than 10 bytes of user data).

TCP obviously handles both types of data, but different algorithms come into play for each. In this chapter, we look at how TCP handles interactive data transfer and model the algorithms that are used in ROC. The topics dealt with here are the working of delayed acknowledgements and Nagle algorithm, to reduce the number of small packets being transferred across wide area networks (WAN).

4.1 Interactive Input

To study the TCP's interactive input, we need to consider an application, which uses TCP's interactive data transfer, such as Rlogin. For each interactive keystroke, TCP

normally generates a data packet. That is the key strokes are sent from the client to the server 1 byte at a time (not one line at a time) [STE99]. Rlogin application has the remote system echo the characters that the client types. Therefore, four segments are generated for each keystroke:

1. The interactive key stroke from the client
2. An acknowledgement of the key stroke from the server
3. The echo of the keystroke from the server
4. An acknowledgement of the echo from the client.

This data transfer is shown in Figure 5.1.

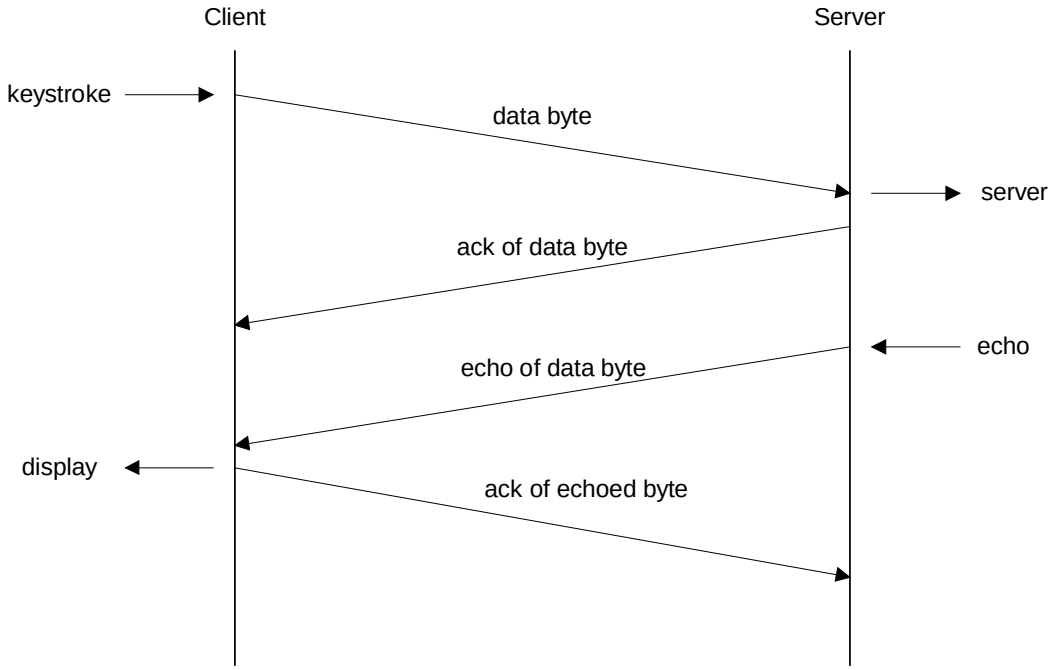

Figure 4.1: Remote Echo of Interactive Keystroke

System := (Client & Server);

Client := (#databyte# ^ ?ackdatabyte? -> ?echodatabyte? -> Client);

Server := (?databyte? -> #ackdatabyte# ^ #echodatabyte# ^ Server);

Ignoring all the details of the TCP header, such as, source and destination ports, sequence number, window sizes, flags etc, we can represent the interactive keystrokes in ROC in a very simple manner as shown above. The data byte sent by the client is accepted by the server, which in return outputs the acknowledgement and an echo. The client receives the acknowledgement and echo and transfers it to the application for display.

4.2 Delayed Acknowledgements

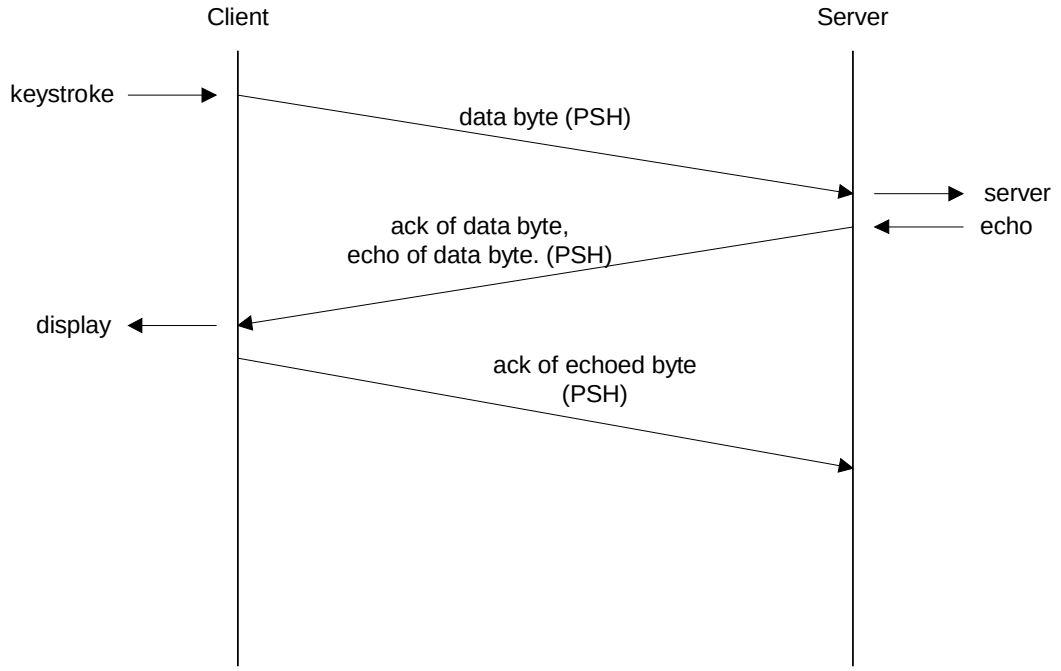

Figure 4.2: Delayed Acknowledgements

Normally, TCP doesn't send the ACK the moment it receives data. Instead, it delays the ACK, hoping to have data going in the same direction as the ACK, so the ACK can be sent along with the data. This is sometimes called having the ACK *piggyback* with the

data. This reduces the number of segments sent across the network. Most implementations of TCP use a 200-ms delay — that is, TCP will delay an ACK up to 200ms to see if there is data to send with the ACK.

The ROC model of the delayed acknowledgement is as follows:

```
System := (Client & Server & SAppln);

SAppln := (#timeout# ^ #sDataByte# ^ SAppln);

Client := (#[cPort,sPort,cSeqNum#,null#,[null#,PSH#],cDataByte#] ^ cClientSent);
cClientSent := (?[sPort, cPort, sSeqN?, sAckN?, [A?, P?], sData?] -> (
        cClientAckPshRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#]
        cClientAckRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#]      ));
cClientAckRcvd := (?[sPort, cPort, sSeqNum?, sAckNum?, [ACK, null], sDataByte?] ->
                cClientAckPshRcvd);
cClientAckPshRcvd := (?[sPort, cPort, sSeqNum?, sAckNum?, [ACK, PSH],
                        sDataByte?] -> Client);

Server := (?[cPort, sPort, cSeqN?, cAckN?, [A?, P?], cData?] -> (
        sTimeoutHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#] +
        sDataByteHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#]
        ) );
sTimeoutHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->
                ?timeout ->
```

#[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, null#], null#] ^

sDataByteHndl @ #[cPort, sPort, cSeqNum#, cAckNum#, [null, PSH#], cData#]);

sDataByteHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->

?sDataByte ->

#[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, PSH#], sDataByte#] ^

Server);

In this system, we have a client, a server and a server side application. The server side application does the part of echoing in case of Rlogin. The client TCP receives data bytes from a client side application, which we haven't shown (as it is understood). For clarity sake, we have shown and used only those flags and fields, which are important. Thus, we have, the client and server's port numbers, client and server sequence numbers and acknowledgement numbers, client and server data bytes, and finally ACK and PSH flags.

The client outputs a packet, which has some data (represented as *cDataBbyte*). At the server, after receiving the packet, either of the following two things could happen. Either, there could be a data byte available to reply to the client, and the ACK would piggyback ride with the data byte, or the 200-ms timer would expire and an ACK packet would be sent. This would be followed up by, a data byte packet from the server side to the client side. Note that the push (PSH) flags are set. This is to indicate to TCP that the data byte has to be immediately delivered to the application. Here, we are simulating, both the timeout situation and data byte to be sent from the server back to the client using

Sappln agent. On the server side, if a *timeout* occurs, then the *sTimeoutHndl* would output an ACK packet, with acknowledgement number set to client's sequence number plus one (*cSeqNum1*), and move to the *sDataByteHndl* state (state modeling using agents), which would then wait for a data byte from the server side application. This is then sent to the client. On the client side, if the client receives an ACK segment with no data byte(s), then it indicates there was no piggyback ride, and so the client moves to a state where it waits for the data byte from the server. On the receipt of this data byte segment, the state returns back to *Client* state.

4.3 Nagle Algorithm

TCP interactive data produces 1byte segments. That is, each packet is 41 bytes long (20 bytes of TCP and IP header each). These small packets, called *tinygrams,* are normally not a problem on LANs, since most LANs are not congested. But these tinygrams can add to congestion on WANs. A simple solution was proposed in RFC 896 [NAG84], called the *Nagle Algorithm*.

The solution says that when a TCP connection has outstanding data that has not yet been acknowledged, small segments cannot be sent until the outstanding data is acknowledged. Instead, small amounts of data are collected by TCP and sent in a single segment when the acknowledgement arrives. The beauty of this algorithm is that it is self-clocking: the faster the ACKs comeback, the faster the data is sent. But on a slow WAN, where it is desired to reduce the number of tinygrams, fewer segments are sent. To encounter this algorithm, the data should be typed in at a rate faster than the round-

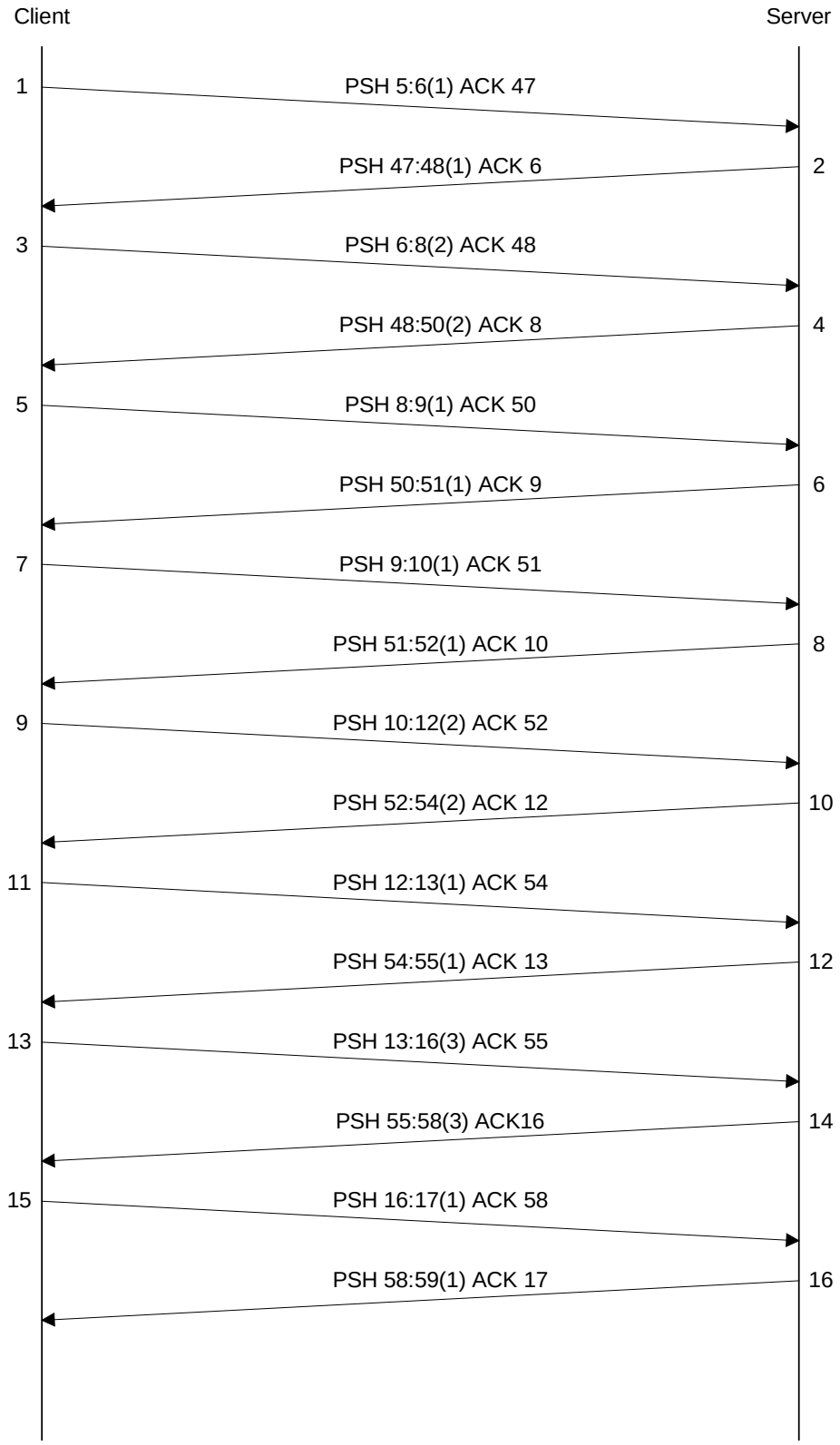

Figure 4.3: Data Flow Showing Nagle Algorithm

trip time. So this algorithm is rarely encountered when sending data between two hosts on a LAN.

Referring to Figure 5.3, which depicts the data flow between two machines on a WAN, where the Nagle algorithm comes into play. The first thing one can notice that there are no delayed ACKs. This is because there is always data ready to be send before the delayed ACK timer expires. One can also notice the varying amounts of data (number of bytes) being sent from left to right: 1, 2, 1, 1, 2, 1, 3 and 1. This is because the client is collecting the data to send and doesn't send until the previously sent data has been acknowledged. Thus, by using the Nagle algorithm, only 8 segments are sent instead of 12 segments.

```
System := (Client & Server & ClientAppln & ServerAppln);

ClientAppln := (#cDataByte# ^ ClientAppln);
ServerAppln := (#timeout ^ #sDataByte ^ ServerAppln );

Client := (?cDataByte? -> #[cPort, sPort, cSeqNum#, null#, [null#,PSH#], cDataByte#]
                        ^ nil & cMsgHndlr);
cMsgHndlr := (
        (?cDataByte? ->
            cClientSent @ #cDataByte# |
            cClientAckRcvd @ #cDataByte# |
            cClientAckPshRcvd @ #cDataByte#  ) |
```

```
        (?[sPort, cPort, sSeqN?, sAckN?, [A?, P?], sData?] ->
            cClientSent @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#]] |
            cClientAckRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#]] |
            cClientAckPshRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#]]
                )
    );

cClientSent := (
    (?[sPort, cPort, sSeqN?, sAckN?, [A?, P?], sData?] -> (
    cClientAckPshRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#] |
    cClientAckRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#] )
    )
        | (?cDataByte? -> cClientSent)
    );
cClientAckRcvd := (
    (?[sPort, cPort, sSeqNum?, cAckNum?, [ACK, null], sDataByte?] ->
    cClientAckPshRcvd)
    | (?cDataByte? -> cClientSent)
    );
cClientAckPshRcvd := (
    (?[sPort, cPort, sSeqNum?, cAckNum?, [ACK, PSH], sDataByte?] ->
    Client) | (?cDataByte? -> cClientSent)
    );
```

```
Server := (?[cPort, sPort, cSeqN?, cAckN?, [A?, P?], cData?] -> (
        sTimeoutHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#] +
        sDataByteHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#]
        ) );
sTimeoutHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->
        ?timeout ->
        #[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, null#], null#] ^
        sDataByteHndl  @ #[cPort, sPort, cSeqNum#, cAckNum#, [null, PSH#],
                                        cData#]);

sDataByteHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->
        ?sDataByte ->
        #[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, PSH#], sDataByte#] ^
                        Server);
```

4.4 Normal Data Flow

Normal data exchange takes place in the ESTABLISHED state of TCP connection. Before entering this state, the maximum segment sizes (MSS) have been exchanged. The window sizes on either side of the connection are also known. The sender keeps sending until all the data is sent. In the process, if the receiver buffers were full, then the server would pause, until there is more room available in receiver buffers, before resuming

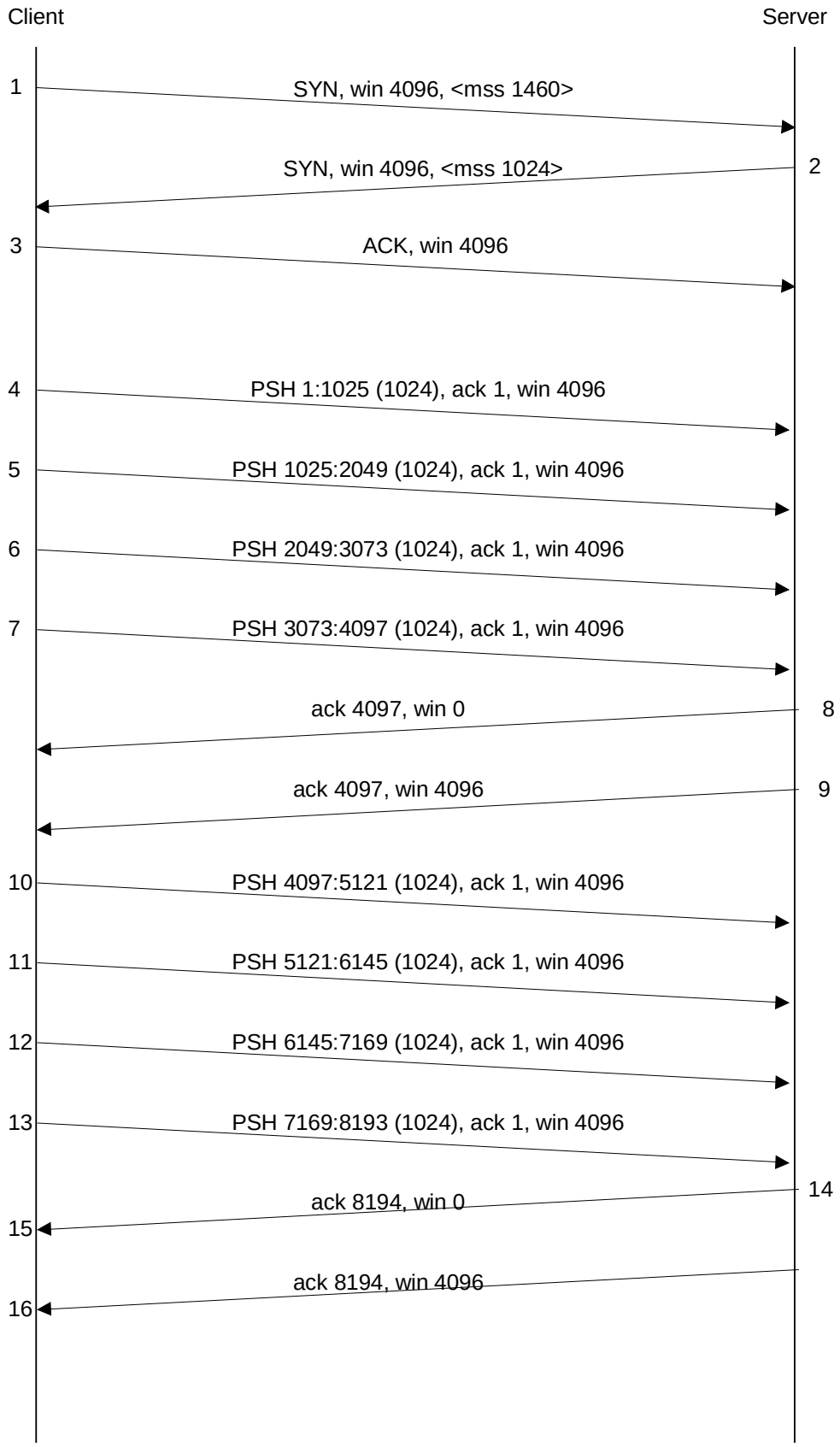

Figure 4.4: TCP Normal Data Flow

resending of packets. On the receiver of the other host, TCP processes the incoming segments. When a segment is processed, the connection is marked to generate delayed ACK. This ACK would piggyback ride if any data to be sent. Else, when the timer expires or when there is more than one outstanding segment to be ACK-ed, the delayed ACK is generated and sent. The delayed ACK acknowledges two segments, that is, the ACKs are cumulative. This strategy is commonly known as "ACK every other segment". Sometimes, delayed ACK may not be generated even if there is more than one outstanding segment.

The window sizes play an important role in flow control in TCP. It prevents a fast sender from drowning a slow receiver with large quantities of data. If the receiver advertises a window size of zero, then the sender waits until the receiver is able to further receiver data, before sending data. This is illustrated in figure 4.4.

Below is the ROC representation of normal TCP data flow. Since ROC doesn't have numbers as primitives, it is not possible to model normal TCP data flow in ROC. This is because of the arithmetic requirements. The following model assumes numbers as being primitives in ROC.

System := (host1Sender & host1Receiver & host1app & host2Sender & host2Receiver);

host1Sender := (localhost2winsize @ iszero -> ((?false -> #MSG# ^ HostSender &

localhost2winsize @ # [-, host1datasize#]) +

(?true -> ?wakeup? -> #MSG# ^ HostSender &

localhost2winsize @ # [-, host1datasize#])));

MSG := (#[host1Port#, host2Port, host1SeqNum#, host2SeqNum @ #[+, 1#],

```
        #host1winsize#, #host1datasize#, #host1data] );
host1Receiver :=( ?[host2Port, host1Port, host2seqNum?, host1SeqNum @ #[+, 1#],
        localhost2winsize?, host2datasize?, host2data?) ->
        (buffer @#[host2datasize#, host2data#] &
        host1winsize @ (-, host2datasize#) )  );
host1app := (?host2data? -> host1winsize @ (+, host2datasize#));
host2sender := (host1sender);
host2receiver := (host1receiver);
```

The receiver on host 1 would receive the data and decrease the window size by the size of the data after placing the data in the buffers. Also, the receiver records the host2 sender's window size. Thus host 1 sender would know the amount of data that can be sent without being dropped. When the application reads the data, it would signal TCP to reset the window size by the size of the data read. The synchronization and race condition prevention is not shown here, though it is trivial to represent. The host 1 sender would send data only if the host 2 receiver's window size is not zero. After sending the data, it would decrement the local copy of host 2's receiver window size by the amount of data sent.

4.5 TCP Slow Start

In the figure 4.4 the sender starts off by injecting multiple segments into the network, up to the window size advertised by the receiver. While this happens when both the sending and receiving hosts are on the same LAN, it doesn't happen when the sender and receiver

are spread across WANs, as this would result in congestion at routers, on the network and reduce the throughput of a TCP connection drastically [JAC88]. The solution to this is TCP's slow start algorithm. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgements are returned by the other end. It adds a new window to the sender's TCP: *congestion window,* called *cwnd*. When a new connection is established with a host on another network, the congestion window is initialized to one segment (the segment size announced by the other end). The sender can transmit up to the minimum of the congestion window and the advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the *cwnd* is doubled to two, and so two segments can be sent. When each of these two segments is acknowledged, the *cwnd* is increased to four. This provides an exponential increase. At some point, intermediate packets will stop dropping packets, which tells the sender that the *cwnd* has got too large.

Thus the ROC expressions for slow start on the sender side are:

HSender & HReceiver & segmentsToSend & acksToRecv & cwnd;

cwnd := cwnd @ #[<, advertisedWnd#];

HSender := (segmentsToSend @ iszero) & (false? -> (TCPSegment# ^ HSender &

segmentsToSend @ #[-, 1#]));

HReceiver := [acknum?, advertisedWnd?, data?] -> segmentsToSend @

#[+, (seqNum @ #[-, acknum#]) @ #[÷, MSS#]]

We have the *cwnd* set to the minimum of *cwnd* and advertised window. The sender checks if it is allowed to send segments or pause for ACKs before it can start sending. When it is allowed to send, it sends a segment and decrements the number of segments to send. The receiver would receive the data and compute the segments to send by dividing the difference of last sent sequence number and received acknowledgement number by the Maximum Segment Size value. This model cannot be simulated on the ROCVM, as ROC doesn't support numerals.

4.6 Congestion Avoidance Algorithm

Slow start, which was described in section 4.5, is the way to initiate data flow across the connection. But at some point, the limit of an intervening router would be reached and packets would be dropped. Congestion avoidance algorithm is the way to deal with lost packets and is described in [JAC88]. The assumption of the algorithm is that 99% of packet loss is due to network congestion. There are two indications of a packet loss: timeout occurring and the receipt of duplicate ACKs. Congestion avoidance and slow start are independent algorithms with different objectives, but implemented together. This is because, when congestion occurs, the transmission rate has to be slowed down and then invoke slow start to get going again. Congestion avoidance is flow control imposed by sender, which is based on sender's assessment of network congestion, while the receiver's advertised window is flow control imposed by the receiver, which is based on the amount of available buffer space.

Congestion avoidance and slow start require that two variables be maintained for each connection: a congestion window, *cwnd*, and a slow start threshold size, *ssthresh*. The combined algorithm works as follows [STE99]:

1. Initialization for a given connection sets *cwnd* to one segment and the *ssthresh* to 65535 bytes.
2. The TCP output routine never sends more than the minimum of *cwnd* and the receiver's advertised window.
3. When congestion occurs, one-half of the current window size, which is the minimum of *cwnd* and receiver's advertised window, is saved in *ssthresh*. Additionally, if the congestion is indicated by timeout, then *cwnd* is set to 1 (slow start).
4. When new data is acknowledged, *cwnd* is increased, based on whether slow start has to be performed or congestion avoidance. If *cwnd* is less than or equal to *ssthresh*, slow start is performed until half way to where congestion occurred (one-half of the window size that caused the congestion was recorded in *ssthresh*), and then congestion avoidance algorithm takes over. Congestion avoidance dictates that *cwnd* be incremented by 1/*cwnd* each time an ACK is received. This is an additive increase, compared to slow start's exponential increase.

Assuming that numbers are primitives in ROC, one can model the congestion avoidance algorithm as follows:

receiver := (?timeout -> (ssthresh :=cwnd @ (÷, ½) & cwnd := 1))

| (?duplicateACK -> (ssthresh := cwnd @ (÷, ½)))

| (?properACK? ->

(cwnd @ (<=, ssthresh) -> ?true -> cwnd @ (×, 2))

+(cwnd @ (+, 1/cwnd)));

In this model, all the TCP header details are left out and only the necessary aspects are shown. Obviously, this model cannot be tested on the ROCVM, because ROC doesn't support numerals as primitives.

4.7 Fast Retransmit and Fast Recovery Algorithms

TCP is required to generate an immediate acknowledgement (a duplicate ACK) when an out-of-order segment is received. This duplicate ACK should not be delayed, whose purpose is to let the other end know that a segment was received out of order and to tell it what sequence number is expected. Since, the duplicate ACK could be due to a lost segment or reordering of segments, a small number of duplicates ACKs are awaited, usually three. If three or more duplicate ACKs are received it is a strong indication that a segment has been lost. A retransmission of the missing segment, without waiting for the retransmission timer to expire, is performed. This is the *fast retransmit* algorithm. This is followed by *congestion avoidance* and not *slow start*. This is the *fast recovery* algorithm. These algorithms are implemented together as follows [STE99]:

1. When the third duplicate ACK is received, *ssthresh* is set to one-half of the minimum of the current congestion window (*cwnd*) and the receiver's advertised window. The missing segment is retransmitted. *cwnd* is set to *ssthresh* plus three times the segment size.

2. Each time another duplicate ACK arrives, *cwnd* is incremented by the segment size and a packet is transmitted, if allowed.
3. When the next ACK arrives that acknowledges new data, *cwnd* is set to *ssthresh*. This ought to be the ACK of the retransmission from step 1. Additionally, this ACK acknowledges all the intermediate segments sent between the lost packet and the receipt of the third duplicate ACK. This step is the congestion avoidance.

Assuming that numbers are primitives in ROC, one can model the congestion avoidance algorithm as follows:

```
receiver := ( ?timeout -> (ssthresh :=cwnd @ (÷, ½) & cwnd := 1))
    | (    (?duplicateACK ->numberofDupAcks @ (==, 3) ->
                    (   ?true -> (ssthresh := cwnd @ (÷, ½)) &
                    transmitter @ #[missingSegment] &
                    cwnd := ssthresh @ (+, segmentsize @ (×, 3) )  )
                    | (?false -> cwnd @ (+, segmentsize) )    )
    | (?properACK? ->
        ( cwnd @ (<=, ssthresh) -> ?true -> cwnd @ (×, 2) )
                    +(  cwnd @ (+, 1/cwnd) )  );
```

In this model, all the TCP header details are left out and only the necessary aspects are shown. Obviously, this model cannot be tested on the ROCVM, because ROC doesn't support numerals as primitives.

CHAPTER V

TCP TIMERS

TCP provides a reliable transport layer. One of the ways it provides reliability is for each to acknowledge the data it receives from the other end. But data segments and acknowledgements could get lost. TCP handles this by setting a timeout when it sends data, and if the data isn't acknowledged when the timeout expires, it retransmits the data. TCP manages four different timers for each connection.

1. A *retransmission* timer is used when expecting an acknowledgement from the other end.
2. A *persist* timer keeps window size information flowing even if the other end closes its receive window.
3. A *keepalive* timer detects when the other end on an otherwise idle connection crashes or reboots.
4. A *2MSL* timer measures the time a connection has been in the TIME_WAIT state.

5.1 TCP Retransmission Timer

With the use of acknowledgements and retransmission timer, a TCP connection is able to provide reliability to the data being communicated. A TCP connection would set a

retransmission timer for the segment immediately after sending the segment. It expects an acknowledgement (ACK segment) for the segment sent. If the ACK segment doesn't arrives before the retransmission timer expires, the data segment is resent, and the retransmission timer is set to a time double the previous time with an upper limit of 64 seconds. This doubling is called *exponential backoff*. Thus, first time the retransmission timer would expire at about 1.5 seconds. There after, the timer expires at 3, 6, 12, 24, 48 and 64 seconds apart. The motivation for doubling of the retransmission timer is to take into account any delays on the network and the application side. At some point, usually tunable, the connection would stop retransmission and reset the connection, though some of today's TCP implementations are persistent.

```
HostSender := (  (?Retransmit -> TCPSegment# ^ HostSender) +
                    (TCPSegment# ^ GetDataState & RetransmissionTimer ) );
RetransmissionTimer := (?RESET? -> nil | Timeout);
HostReceiver := ( (?TCPSegmentACK? -> StoreData &
                        RetransmissionTimer @ #[RESET#] ) +
                        (?Timeout -> HostSender @ #[Retransmit]) );
```

The sender on the host would send a TCP segment and try to get further data to send. At the same time, it would start a retransmission timer, which would be counting down from a set value. If the host receiver receives an ACK for the segment transmitted, it would reset the retransmission timer. If the retransmission timer expires before being reset, it would force the Host sender to retransmit the TCP segment once again.

5.2 TCP Persist Timer

TCP receiver performs flow control by specifying the amount of data it is willing to accept from the sender using the window size. If the window size goes to zero, the sender is effectively stopped from transmitting data until the window size becomes nonzero. The re-opening of a shut window in TCP is indicated by the transmission of an ACK segment with the new available window size. Since, acknowledgements are not reliably transmitted (TCP does not ACK acknowledgements. Only data segments are ACK-ed), if ACK segments get lost, both sides would end up waiting for the other side to transmit and would result in a deadlock. This is modeled in ROC below.

System := (Sender & Receiver & Channel);

Receiver := (#[winsize0#, [ACK]] ^ #[winsize1024#, [ACK]] ^ Receiver);

Channel := (?[winsize1024, [ACK]] -> nil);

Sender := (?[winsize0, [ACK]] -> SenderPause);

SenderPause := (?[winsize1024, [ACK]] -> #[data#] ^ Sender);

In the above model, we introduce the channel to consume the ACK segment with window size 1024 to simulate the lost-ACK. The sender, which has entered the pause state, waits for the receiver to signal the opening of the window, while the receiver is waiting for data and thus we have a deadlock.

To prevent this, the sender uses a *persist timer* that causes it to query the receiver periodically, to find out if the window has been increased. These segments are called *window probes*. These window probes contain one byte of data. TCP is always allowed to send one byte of data beyond a closed window. The ACK sent in reply to this window probe does not ACK the one byte of data sent as part of the window probe. Therefore,

the one byte of data is kept on being retransmitted until persist timer is reset. The normal TCP exponential back off is used when calculating the persist timer. Thus, the first timeout would occur after 1.5 seconds on a typical LAN connection. This is followed by timeouts at 3 seconds, 6 seconds, 12 seconds etc, with an upper limit of 60 seconds. The characteristic of TCP persist timer is that it never gives up sending window probes as long as the connection exists.

System := (Sender & PreReceiver & PersistTimer & Channel);

PersistTimer := (#[expSeqMinus1#] ^ PersistTimer);

PreReceiver := (#[winsize0#, [ACK]] ^ #[winsize1024#, [ACK]] ^ Receiver);

Receiver := ((?[expSeqMinus1] -> #[winsize1024#, [ACK]] ^ Receiver) |

(?[expSeq, data?] -> Receiver));

Channel := (?[winsize1024, [ACK]] -> nil);

Sender := (?[winsize0, [ACK]] -> SenderWait);

SenderWait := (?[winsize1024, [ACK]] -> #[data#] ^ Sender);

5.3 TCP Keepalive Timer

TCP connections do not have polling, which means a connection once established remains for hours, days or months even if the intermediate routers crash and reboot, phone lines go down and come up, as long as neither hosts at both ends of the connection reboot (this assumes that both client and server side applications don't terminate). There are however instances when a server would like to know if the client's host has either crashed and is down, or crashed and rebooted. The *keepalive timer*, a feature of many implementations provides this capability. The keepalive feature is intended for server

applications that might tie up resources on behalf of a client, and want to know if the client host crashes.

Keepalive timers are not part of TCP specification. In fact, the Host requirement RFC provides three reasons not to use them: (1) they cause perfectly good connections to be dropped during transient failures, (2) they consume unnecessary bandwidth, and (3) they cost money on an internet that charges by the packet. Nevertheless, many implementations provide the keepalive timer. It has been a controversial feature. The host RFC says that an implementation may provide the keepalive timer, but it must not be enabled unless the application specifically says so. Also, the keepalive interval must be configurable, but it must default to no less than two hours. A keepalive probe segment contains no data and none of the flags are set.

The keepalive timer works as follows. If there is no activity on a given connection for 2 hours, then the server sends a probe segment to the client. The client host must be in one of the three states:

1. The client host is still up and running and reachable from the server. In this case, the client's TCP responds normally to the server and the server will reset the keepalive timer for 2 hours in the future. Whenever there is application traffic, this timer is reset.
2. The client's host has crashed and is either down or rebooting. In either case, its TCP is not responding. The server will not receive a response for its probe and it times out after 75 seconds. The server sends a total of 10 such probes, each 75 seconds apart, and if it doesn't receive a response, then the server considers the client host to be down and reset the connection.

3. The client host has crashed and rebooted. The server would receive a response, but it would be a reset (RST), causing the connection to terminate.

Unlike the SYN, FIN, RST segments, which consume a sequence number, probe segment does not consume any sequence numbers. The probe segment contains a sequence number, which is one less than the expected sequence number. It is this incorrect sequence number that forces the client to respond with an ACK to the keepalive probe, which tells the server the next sequence number the client is expecting.

```
2hourTimer := (#2hourTimeout ^ 2hourTimer);
75secondTimer := (#75secondTimeout ^ 75secondTimer);
Channel := (?Any? -> Channel);

Server := (?2hourTimeout ->
            #[sPort, cPort, sSeqNumMinus1#, cAckNum#, [null#, null#, null#]] ^
            sProbeSent & 75secondTimer
            );
sProbeSent := ( (?[cPort, sPort, cSeqN?, sSeqNum, [ACK, null, null]] -> Server) |
      (?75secondTimeout ->
            #[cPort, sPort, sSeqNumMinus1#, cAckNum#, [null#, null#, null#]] ^
      sProbeSent) );

Client := ( (?[sPort, cPort, sSeqNum, cAckNum, [A?, P?, F?]] ->  clientNextState) |
   (?[sPort, cPort, sSomeSeq?, cAckNum, [A?, P?, F?]] ->
```

#[cPort, sPort, cSeqNum#, sSeqNum#, [ACK#, null#, null#]] ^ Client));

5.4 TCP 2MSL Timer

TCP identifies each byte of data with a 32-bit unsigned sequence number. Since the sequence number space is finite, the same sequence number is reused after 4,294,967,296 bytes have been transmitted [STE99]. One of the segments could get delayed in the network and reappear later, and interfere with an existing connection. Any delayed segments that arrive for a connection while it is in the 2MSL wait are discarded. Since, the connection defined by the socket pair in the 2MSL wait cannot be reused during this time period, when we do establish a valid connection, delayed segments from an earlier incarnation of this connection cannot be misinterpreted as being part of the new connection. The recommended value of the 2MSL is 2 minutes (giving a 2MSL of 240 seconds), but most implementations use an MSL value of 30 seconds.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

The first section of this chapter contains the conclusions of the research and the second section highlights upon future work to be done on the ROC process calculus to adapt it better for modeling of distributed real-time systems.

6.1 Conclusions

Modeling real-time distributed systems is non trivial. The modeling language or tool should be powerful and rich with capabilities to model concurrency, non-deterministic behavior, complex message passing for communication, state of the system and real-time issues. We have studied the feasibility of ROC as a modeling language for real-time distributed systems and communication protocols using popular transport layer protocol, Transmission Control Protocol. The following are the advantages that ROC possesses, which would be handy in modeling:

- ✓ Communication between agents is a primitive operation in ROC.
- ✓ ROC supports complex message passing with its tuple-based communication.
- ✓ The functional and process paradigms are unified in ROC. This is very handy for both local and remote communication.

These three features make ROC highly desirable for modeling communication systems and protocols.

- ✓ ROC has the added advantage of object encapsulation. Thus, using this feature, not only values, but also agents can be communicated.
- ✓ ROC possesses capability to model concurrency and non-determinism.
- ✓ ROC also provides two primitives not found in other calculus, namely, non-deterministic choice and left preferential concurrent composition.

Together, these make ROC a powerful modeling language for distributed object systems.

- ✓ And with its background in Higher Order Logic, using which one can specification and verification is possible and provable with mathematical logic and reason, ROC is ideal for modeling distributed systems.

While ROC has many powerful capabilities required for modeling real-time distributed systems and communication protocols, it has a few shortcomings too. Currently, ROC can model concurrency, states (agents are used to model state information), message passing and non-determinism. The shortcomings are:

- ✖ Numbers are not primitives in ROC.

  Numbers, the basis for all Math, are indispensable when it comes to model many protocol features. While modeling the TCP protocol, there have been many instances where numbers were needed. For example, in the acknowledgement of segments at the receiver side, there is a need to identify a specific sequence number, acknowledge any outstanding data, etc. For all this, numbers would be

very useful. There are a number of timers used in TCP. There is a need to set these timers and also ways by which they could expire at appropriate times. If the timers are to expire based on ticks, then to count the number of ticks that have occurred or to decrement the current time at the occurrence of every tick needs support for numbers in ROC. Numbers form the basic foundation of all systems and hence, there is a strong necessity for making them primitives in ROC.

- Multicast communication is not possible in ROC, which is extremely important for modeling many communication protocols and distributed systems.

  Modeling of protocols like Internet Protocol (IP) requires support for multicast. Currently in ROC, messages are produced one at a time, and all messages have to be consumed. Thus, to simulate multicast, one has to output the same message a number of times to all the agents of the group. But this is not multicast as the time the message sent to one agent differs from another. Multicast support would be extremely powerful tool in ROC to model many of the communication protocols. If an external clock, like Network Time Protocol (NTP) time is being used to provide time to the distributed systems, then to there is a need for multicasting the ticks to each of the agents of the distributed system.

- In ROC, if an agent produces a message, then the agent cannot undergo any further reduction until the message produced has been consumed. In many cases, this is a necessity (synchronous operations). But in many other cases this becomes a restriction (asynchronous operations) in modeling distributed systems. There is a need to have support in ROC by which asynchronous and synchronous operations can be modeled with message passing.

- With the lack of facility to model time dependent issues, one cannot successfully model real-time systems, which is all about timing.

  We have seen while modeling timers and many other TCP algorithms that, without means to specify time in the system, it is not possible to conveniently and accurately model the algorithm or protocol. There is no way for validating if a real-time system can meet time deadlines if there is no support for time or timers. Making time as a primitive in ROC could make specification of real time systems easy. This could then translate directly into HOL theorems and we could verify the timeliness of a real-time system. Alternately, time could be introduced by implementing synchronizations with a global clock, which needs numerals and multicast support.

6.2 ROC Extensibility to MOM

Formal methods are powerful tools for high assurance computing. However, making them practical is an extremely difficult task. Often, applying formal methods requires a great deal of expertise that developers simply do not have. The Meta-Object Model (MOM), an intermediate layer in the MOOSE framework designed with ROC. MOM can be used to model a variety of object-oriented systems. MOM can be used to express the behavior of virtually any distributed object system. A capability-based access control scheme integrated into MOM permits the development of secure systems. Because, it can serve as a common substrate between disparate object systems, it holds the key to interoperability between heterogeneous systems. MOM objects encapsulate code and data that bundle service ands state into neat packages. A MOM object espouses

the virtues of modularity and re-use, and touts abstraction and concurrency as potential benefits. All these factors make MOM a very powerful utility for modeling of real-time distributed systems. Using the advantages offered, developers could use MOM as building blocks of their systems, which would ensure security and reliability.

6.3 Future Work

One of the limitations of ROC is that multicast of messages is not possible in ROC. Currently messages can be sent from one agent to another only and all messages have to be consumed for the agent to send the next message. The ability to model multicast messages would be necessary for modeling communication protocols in a cluster of distributed systems, where lots of multicast messages are often used. There is a need for work to be done to support both synchronous and asynchronous communication.

Also, as mentioned in the conclusion section above, the ability to model time in ROC is necessary to extend the effectiveness of ROC into the distributed real-time domain. Time and timing is the essence of real-time systems and any real-time system model would require the ability to model time. One approach could be to introduce a timer, which keeps sending *ticks* to all agents. Due to the lack of multicast, one cannot introduce use this timer concept. Also, the effectiveness of this approach needs to further explored. Another approach would be to introducing time into ROC. If time becomes a primitive of ROC, then the modeling of real-time systems would come naturally. Also, the HOL theorems have to be modified and re-verified. If one can achieve making time a primitive of ROC, higher layers can be built on it, which would have time as an in-built component of them.

Numbers play an important role in engineering problems. There is necessity to perform basic arithmetic such as *addition*, *subtraction*, *division*, *multiplication*, comparison operators like *less than*, *greater than*, *equal to*, and Boolean operations such as *AND*, *OR*, *NOT*, *NEGATIVE* besides the basic *TRUE* and *FALSE*. Numbers by themselves are not basic primitives in ROC. Thus, all the above need to be modeled. While almost all of these can be modeled by treating numbers as agents, the effectiveness of such models in real-time systems needs to be answered. The effects of keeping numbers out of ROC primitives on real-time system models need to be studied, and probably ROC primitives need to be enhanced to incorporate numbers.

BIBLIOGRAPHY

[ALV91] J. Alves-Foss, K. Levitt, Mechanical verification of secure distributed systems in higher order logic. *Proceedings of the 1991 International Workshop on the Higher Order Logic Theorem Proving Systems and its Application*, 1991, pp. 263-278.

[BAR84] H.P. Barendregt, *The Lambda Calculus – Its Syntax and Semantics*, Studies in Logic and Foundations of Mathematics, 103, North-Holland, 1984.

[BUR97] A. Burns, A. Wellings, *Real-time Systems and Programming Languages*, Addison-Wesley, Reading, Massachusetts, 1997.

[CAC91] R. Caceres, P. B. Danzig, S. Jamin, D. J. Mitzel, Characteristics of Wide-Area TCP/IP Conversations, *Computer Communications Review*, September 1991, vol.21, no.4, pp.101-112.

[CHO95] C.T. Chou, Mechanical Verification of Distributed Algorithms in Higher Order Logic, *The Computer Journal*, 1995, Vol. 38, No. 1.

[CLA96] E. Clarke, J. Wing, *Formal Methods: State of the art and future directions*, Technical Report CMU-CS-96-178, Carnegie Mellon University, August 1996.

[COU01] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems, Concept and Design*, Addison-Wesley, Reading, Massachusetts, 2001.

[DIL90] A. Diller, *Z: An Introduction to Formal Methods*, John-Wiley, New York, 1990.

[HOA85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, New York, 1985.

[GOR93] M. Gordon, T.F. Melham, editors, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, Cambridge, U.K., 1993.

[HAL97] J. C. Hale, *Seamless and Secure Interoperation of Heterogenous Distributed Objects*, Ph.D. Dissertation, Department of Computer Science, The University of Tulsa, Tulsa, Oklahoma, 1997.

[JAC81] V. Jacobson, R. Braden, D. Borman, *TCP Extensions for High Performance*, RFC 1323, May 1992, <http://www.cis.ohio-state.edu/htbin/rfc/rfc1323.html>.

[JAC88] V. Jacobson, Congestion Avoidance and Control, *Computer Communications Review*, August 1988, Vol.18, No.4, pp.314-329.

[JAC90] V. Jacobson, *Modified TCP Congestion Avoidance Algorithm*, April 30, 1990, end2end-internet mailing list (Apr.).

[MIL89] R. Milner, *Communication and Concurrency*, Prentice Hall, New York, 1989.

[MIL92] R. Milner, The Polyadic π-calculus: A Tutorial, In M. Broy, editor, *Logic and Algebra of Specification*, Springer-Verlag, Amstredam, The Netherlands, 1992.

[MPW89] R. Milner, J. Parrow, D. Walker, *A Calculus of Mobile Process, pars i and ii*, Technical Report ECS-LFCS-89-85 and 86, University of Edinburgh, March 1989.

[NAG84] J. Nagle, *Congestion Control in IP/TCP Internetworks*, RFC 896, January 1984, <http://www.cis.ohio-state.edu/htbin/rfc/rfc896.html>.

[NIE91] O. Nierstrasz, Towards an Object Calculus, In M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, Amstredam, The Netherlands, 1991.

[OPE92] Open Systems Foundation, *Security in a distributed environment*, Technical Report OSF-O-WP11-1090-3, Open Systems Foundation, Cambridge, Massachussetts, 1992.

[PIE95] B. Pierce, D. Turner, Concurrent Objects in a Process Calculus, In Takaysau Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming* (*TPPP*), Springer-Verlag, April 1995, Vol. 907 of *Lecture Notes in Computer Science*, pp. 187-215.

[POS81] J. B. Postel, *Transmission Control Protocol*, RFC 793, September 1981, <http://www.cis.ohio-state.edu/htbin/rfc/rfc793.html>.

[SAI96] H. Saiedian, An Invitation to Formal Methods, *IEEE Computer*, 1996, Vol. 29, No. 4, pp. 16-30.

[STE99] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Reading, Massachusetts, 1999.

[TAN95] A. Tannenbaum, *Distributed Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[THR96] J. R. Threet, J. Hale, S. Shenoi, *A Process Calculus for Distributed Objects*, Technical Report UTULSA-MCS-96-7, The University of Tulsa, Tulsa, Oklahoma, April 1996.

[THR97] J. R. Threet, *A Formal Process Calculus and Execution Model for Distributed Agent Computing*, Ph.D. Dissertation, Department of Computer Science, The University of Tulsa, Tulsa, Oklahoma, 1997.

[WAL91] D. Walker, π-Calculus Semantics for Object Oriented Programming Languages, In M. Gaudel and J. Jouannaud, editors, *TACS'91: Proceedings of*

International Conference on Theoretical Aspects of Computer Science, Springer-Verlag, Amstredam, The Netherlands, 1991, Vol. 526 of *Lecture Notes in Computer Science*, pp. 532-547.

[WIN90] J. Wing, A Specifier's Introduction to Formal Methods, *Computer*, September 1990, Vol. 23, No. 9, pp. 8-21.

[ZHA00] J. Zhang, *Visualization of Robust Object Calculus Virtual Machine*, Masters thesis, Department of Computer Science, Washington State University, Pullman, Washington, December 2000.

APPENDIX A

ROC MODELS OF TCP ALGORITHMS

A.1 TCP Header

TCPMessage (srcPort, dstPort, seqNum, ackNum, hdrLen, rsvd, FLAGS,

wndSize, chksum, urgPtr, OPTIONS, data) :=

#[srcPort#, dstPort#, seqNum#, ackNum#, hdrLen#, rsvd#, FLAGS#, wndSize#,

chksum#, urgPtr#, OPTIONS#, data#] ^ nil;

FLAGS := [URG# | null#, ACK# | null#, PSH# | null#, RST# | null#,

SYN# | null#, FIN# | null#]#;

OPTIONS := [EOPL#, NOP#, MSS#, WSF#, TS#]#;

EOPL := [kind#]#;

NOP := [kind#]#;

MSS := [kind#, len#, mss#]#;

WSF := [kind#, len#, shiftcount#]#;

TS := [kind#, len#, tsValue#, tsEchoReply#]#;

A.2 TCP Connection Establishment Protocol

System := (Client & Server);

Client := (#[cPort#, sPort, cSeqNum#, null#, [null#, SYN#]] ^ cSynSent);

cSynSent := (?[sPort, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb);

Server := (?[cPort?, sPort, cSeqNum?, cAckNum?, [null, SYN]] ->

#[sPort#, cPort#, sSeqNum#, cSeqNum1#, [ACK#, SYN#]] ^ sSynRcvd);

sSynRcvd := (?[cPort?, sPort, cSeqNum1?, sSeqNum1, [ACK, null]] -> sEstb);

A.3 TCP Connection Termination Protocol

System := (cEstb & sEstb);

cEstb := (#[cPort#, sPort, cSeqNum#, null#, [null#, FIN#]] ^ cFinWait1);

cFinWait1 := (?[sPort, cPort, sSeqNum?, cSeqNum1, [ACK, null]] ->

?[sPort, cPort, sSeqNum1?, cSeqNum1, [null, FIN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum2, [ACK, null#]] ^

cTimeWait);

cTimeWait := (nil);

sEstb := (?[cPort?, sPort, cSeqNum?, cAckNum?, [null, FIN?]] ->

#[sPort#, cPort, sSeqNum#, cSeqNum1, [ACK#, null#]] ^

sCloseWait);

sCloseWait := (#[sPort#, cPort, sSeqNum1#, cSeqNum1, [null#, FIN#]] ^ sLastAck);

sLastAck := (?[cPort, sPort, cSeqNum?, sSeqNum2, [ACK, null]] -> nil);

A.4 Timeout of Connection Establishment

System := (Client & Timer & Channel);

Client := (#[cPort#, sPort, cSeqNum#, null#, [null#, SYN#]]# ^ cSynSent);

cSynSent := ((?[sPort?, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->

```
                    #[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb)
            | (?Timeout -> ?Timeout -> ?Timeout -> nil) );
Timer := (#Timeout ^ #Timeout ^ #Timeout ^ Timer);
Channel := (?PKT? -> Channel);
```

A.5 TCP State Transition Diagram

```
System := (Client & Server);
Client := (#[cPort#,sPort,[null#,SYN#,null#]] ^ cSynSent & cMsgHdlr);
cMsgHdlr := (?[sP?,cPort,[A?,S?,F?]] -> (
    cSynSent @ #[sP#,cPort,[A#,S#,F#]] |
    cEstb @ #[sP#,cPort,[A#,S#,F#]] |
    cFinWait1 @ #[sP#,cPort,[A#,S#,F#]] |
    cFinWait2 @ #[sP#,cPort,[A#,S#,F#]] |
    cTimeWait @ #[sP#,cPort,[A#,S#,F#]] |
    cClosing @ #[sP#,cPort,[A#,S#,F#]]
    ) & cMsgHdlr);
cSynSent := (?[sPort?,cPort,[ACK,SYN,null]] ->
            #[cPort#,sPort,[ACK#,null#,null#]] ^ cEstb);
cEstb := (#[cPort#,sPort,[null#,null#,FIN#]] ^ cFinWait1);
cFinWait1 := ( (?[sPort?,cPort,[ACK,null?,null?]] -> cFinWait2) +
    (?[sPort?,cPort,[null?,null?,FIN]] ->
            [cPort#,sPort,[ACK#,null#,null#]] ^ cClosing) +
    (?[sPort?,cPort,[ACK,null?,FIN]] ->
```

```
                #[cPort#,sPort,[ACK#,null#,null#]] ^ cTimeWait) );
cFinWait2 := (?[sPort?,cPort,[null?,null?,FIN]] ->
                #[cPort#,sPort,[ACK#,null#,null#]] ^ cTimeWait);
cClosing := (?[sPort?,cPort,[ACK,null?,null?]] -> cTimeWait);
cTimeWait := cClosed;
cClosed := (nil);
Server := (sListen & sMsgHdlr);
sMsgHdlr := (?[cP?,sPort,[A?,S?,F?]] -> (
    sSynRcvd @ #[cP#,sPort,[A#,S#,F#]] |
    sListen @ #[cP#,sPort,[A#,S#,F#]] |
    sEstb @ #[cP#,sPort,[A#,S#,F#]] |
    sCloseWait @ #[cP#,sPort,[A#,S#,F#]] |
    sLastAck @ #[cP#,sPort,[A#,S#,F#]]
    ) & sMsgHdlr);

sListen := (?[cPort?,sPort,[null?,SYN,null?]] ->
                #[sPort#,cPort,[ACK,SYN,null#]] ^ sSynRcvd);
sSynRcvd := (?[cPort?,sPort,[ACK,null?,null?]] -> sEstb);
sEstb := (?[cPort?,sPort,[null?,null?,FIN]] ->
                #[sPort#,cPort,[ACK#,null#,null#]] ^ sCloseWait);
sCloseWait := (#[sPort#,cPort,[null#,null#,FIN#]] ^ sLastAck);
sLastAck := (?[cPort?,sPort,[ACK,null?,null?]] -> sClosed);
sClosed := (nil);
```

A.6 Reset Segments

```
System := (Client & Server);

Client := (#[cPort#, somePort#, cSeqNum#, null#, [null#, SYN#]]# ^ cSynSent);

cSynSent := (  ( ?[sPort?, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->
                #[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb ) |
                ( ?[sPort?, cPort, anySeqNum?, anyAckNum?, [RST]] -> nil )  );

cEstb := (#OUT# ^ nil);

Server := (( ?[cP?, sPort, cSeqNum?, cAckNum?, [null, SYN]] ->
                #[sPort#, cP#, sSeqNum#, cSeqNum1#, [ACK#, SYN#]] ^ sSynRcvd) |
                ( ?[cP?, nonExistentPort?, anySeq?, anyAck?, [A?, SYN?]] ->
                #[nonExistentPort#, cPort, null#, null#, [RST#]] ^ Server)       );

sSynRcvd := (?[cPort?, sPort, cSeqNum1?, sSeqNum1, [ACK, null]] -> sEstb);

sEstb := (#OUT# ^ nil);
```

A.7 TCP Simultaneous Open

```
System := (Client & Server);

Client := (#[cPort#, sPort, cSeqNum#, null#, [null#, SYN#]]# ^ cSynSent);

cSynSent := (  (?[sPort?, cPort, sSeqNum?, cSeqNum1, [ACK, SYN]] ->
                        #[cPort#, sPort, cSeqNum1#, sSeqNum1, [ACK, null#]] ^ cEstb)
            | (?Timeout -> ?Timeout -> ?Timeout -> nil)     );

Server := Client;
```

A.8 TCP Simultaneous Close

System := (cEstb & sEstb);

cEstb := (#[cPort#, sPort, cSeqNum#, null#, [null#, FIN#]] ^ cFinWait1);

cFinWait1 := (?[sPort, cPort, sSeqNum?, cSeqNum1, [ACK, null]] ->

?[sPort, cPort, sSeqNum1?, cSeqNum1, [null, FIN]] ->

#[cPort#, sPort, cSeqNum1#, sSeqNum2, [ACK, null#]] ^

cTimeWait);

cTimeWait := (nil);

sEstb := (cEstb);

A.9 Interactive Input

System := (Client & Server);

Client := (#databyte# ^ ?ackdatabyte? -> ?echodatabyte? -> Client);

Server := (?databyte? -> #ackdatabyte# ^ #echodatabyte# ^ Server);

A.10 Delayed Acknowledgements

System := (Client & Server & SAppln);

SAppln := (#timeout# ^ #sDataByte# ^ SAppln);

Client := (#[cPort,sPort,cSeqNum#,null#,[null#,PSH#],cDataByte#] ^ cClientSent);

cClientSent := (?[sPort, cPort, sSeqN?, sAckN?, [A?, P?], sData?] -> (

cClientAckPshRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#]

cClientAckRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#]));

cClientAckRcvd := (?[sPort, cPort, sSeqNum?, sAckNum?, [ACK, null], sDataByte?] ->

```
        cClientAckPshRcvd);
cClientAckPshRcvd := (?[sPort, cPort, sSeqNum?, sAckNum?, [ACK, PSH],
                sDataByte?] -> Client);
Server := (?[cPort, sPort, cSeqN?, cAckN?, [A?, P?], cData?] -> (
    sTimeoutHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#] +
    sDataByteHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#]
    ) );
sTimeoutHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->
        ?timeout ->
        #[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, null#], null#] ^
        sDataByteHndl  @ #[cPort, sPort, cSeqNum#, cAckNum#, [null, PSH#],
        cData#]);
sDataByteHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->
        ?sDataByte ->
        #[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, PSH#], sDataByte#] ^
        Server);
```

A.11 Nagle Algorithm

```
System := (Client & Server & ClientAppln & ServerAppln);
ClientAppln := (#cDataByte# ^ ClientAppln);
ServerAppln := (#timeout ^ #sDataByte ^ ServerAppln );
Client := (?cDataByte? -> #[cPort, sPort, cSeqNum#, null#, [null#,PSH#], cDataByte#]
        ^ nil & cMsgHndlr);
```

```
cMsgHndlr := (  (?cDataByte? ->
            cClientSent @ #cDataByte# |
            cClientAckRcvd @ #cDataByte# |
            cClientAckPshRcvd @ #cDataByte#  ) |
        (?[sPort, cPort, sSeqN?, sAckN?, [A?, P?], sData?] ->
            cClientSent @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#]] |
            cClientAckRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#]] |
            cClientAckPshRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#]]
                )    );
cClientSent := ( (?[sPort, cPort, sSeqN?, sAckN?, [A?, P?], sData?] -> (
    cClientAckPshRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#] |
    cClientAckRcvd @ #[sPort, cPort, sSeqN#, sAckN#, [A#, P#], sData#] )   )
        | (?cDataByte? -> cClientSent)   );
cClientAckRcvd := (
    (?[sPort, cPort, sSeqNum?, cAckNum?, [ACK, null], sDataByte?] ->
    cClientAckPshRcvd)
    | (?cDataByte? -> cClientSent)    );
cClientAckPshRcvd := (
    (?[sPort, cPort, sSeqNum?, cAckNum?, [ACK, PSH], sDataByte?] ->
    Client) | (?cDataByte? -> cClientSent)    );
Server := (?[cPort, sPort, cSeqN?, cAckN?, [A?, P?], cData?] -> (
        sTimeoutHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#] +
        sDataByteHndl @ #[cPort, sPort, cSeqN#, cAckN#, [A#, P#], cData#]
```

```
    ) );
sTimeoutHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->
        ?timeout ->
        #[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, null#], null#] ^
        sDataByteHndl  @ #[cPort, sPort, cSeqNum#, cAckNum#, [null, PSH#],
                                        cData#]);
sDataByteHndl := (?[cPort, sPort, cSeqNum?, cAckNum?, [null, PSH], cDataByte?] ->
        ?sDataByte ->
        #[sPort, cPort, sSeqNum#, cSeqNum1#, [ACK#, PSH#], sDataByte#] ^
                Server);
```

A.12 Normal Data Flow

```
System := (host1Sender & host1Receiver & host1app & host2Sender & host2Receiver);
host1Sender := (localhost2winsize @ iszero -> ( (?false -> #MSG# ^ HostSender &
                localhost2winsize @ # [-, host1datasize#]) +
                (?true -> ?wakeup? -> #MSG# ^ HostSender &
                localhost2winsize @ # [-, host1datasize#]) ) );
MSG := ( #[host1Port#, host2Port, host1SeqNum#, host2SeqNum @ #[+, 1#],
            #host1winsize#, #host1datasize#, #host1data] );
host1Receiver :=( ?[host2Port, host1Port, host2seqNum?, host1SeqNum @ #[+, 1#],
        localhost2winsize?, host2datasize?, host2data?) ->
            (buffer @#[host2datasize#, host2data#] &
            host1winsize @ (-, host2datasize#) )  );
```

host1app := (?host2data? -> host1winsize @ (+, host2datasize#));

host2sender := (host1sender);

host2receiver := (host1receiver);

A.13 TCP Slow Start

HSender & HReceiver & segmentsToSend & acksToRecv & cwnd;

cwnd := cwnd @ #[<, advertisedWnd#];

HSender := (segmentsToSend @ iszero) & (false? -> (TCPSegment# ^ HSender &

segmentsToSend @ #[-, 1#]));

HReceiver := [acknum?, advertisedWnd?, data?] -> segmentsToSend @

#[+, (seqNum @ #[-, acknum#]) @ #[÷, MSS#]];

A.14 TCP Congestion Avoidance Algorithm

receiver := (?timeout -> (ssthresh :=cwnd @ (÷, ½) & cwnd := 1))

| (?duplicateACK -> (ssthresh := cwnd @ (÷, ½)))

| (?properACK? ->

(cwnd @ (<=, ssthresh) -> ?true -> cwnd @ (×, 2))

+(cwnd @ (+, 1/cwnd)));

A.15 Fast Retransmit and Fast Recovery Algorithm

receiver := (?timeout -> (ssthresh :=cwnd @ (÷, ½) & cwnd := 1))

| ((?duplicateACK ->numberofDupAcks @ (==, 3) ->

(?true -> (ssthresh := cwnd @ (÷, ½)) &

```
                transmitter @ #[missingSegment] &
                cwnd := ssthresh @ (+, segmentsize @ (×, 3) )  )
                | (?false -> cwnd @ (+, segmentsize) )    )
    | (?properACK? ->
        ( cwnd @ (<=, ssthresh) -> ?true -> cwnd @ (×, 2) )
        +(  cwnd @ (+, 1/cwnd) )  );
```

A.16 TCP Retransmission Timer

```
HostSender := (  (?Retransmit -> TCPSegment# ^ HostSender) +
                (TCPSegment# ^ GetDataState & RetransmissionTimer ) );
RetransmissionTimer := (?RESET? -> nil | Timeout);
HostReceiver := ( (?TCPSegmentACK? -> StoreData &
                        RetransmissionTimer @ #[RESET#] ) +
                        (?Timeout -> HostSender @ #[Retransmit]) );
```

A.17 TCP Persist Timer

```
System := (Sender & Receiver & Channel);
Receiver := (#[winsize0#, [ACK]] ^ #[winsize1024#, [ACK]] ^ Receiver);
Channel := (?[winsize1024, [ACK]] -> nil);
Sender := (?[winsize0, [ACK]] -> SenderPause);
SenderPause := (?[winsize1024, [ACK]] -> #[data#] ^ Sender);
```

A.18 TCP Keepalive Timer

```
2hourTimer := (#2hourTimeout ^ 2hourTimer);

75secondTimer := (#75secondTimeout ^ 75secondTimer);

Channel := (?Any? -> Channel);


Server := (?2hourTimeout ->
                #[sPort, cPort, sSeqNumMinus1#, cAckNum#, [null#, null#, null#]] ^
                sProbeSent & 75secondTimer
                );
sProbeSent := ( (?[cPort, sPort, cSeqN?, sSeqNum, [ACK, null, null]] -> Server) |
        (?75secondTimeout ->
                #[cPort, sPort, sSeqNumMinus1#, cAckNum#, [null#, null#, null#]] ^
        sProbeSent) );


Client := ( (?[sPort, cPort, sSeqNum, cAckNum, [A?, P?, F?]] ->  clientNextState) |
    (?[sPort, cPort, sSomeSeq?, cAckNum, [A?, P?, F?]] ->
     #[cPort, sPort, cSeqNum#, sSeqNum#, [ACK#, null#, null#]] ^ Client)  );
```