CHANNEL MARKET ANALYSIS OF APPLICATION-DRIVEN CONNECTION RECOVERY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Robert Surton

May 2014

This dissertation is
dedicated to Elizabeth
and my children, who are
vastly more important to me
than it is.

CHANNEL MARKET ANALYSIS OF APPLICATION-DRIVEN CONNECTION RECOVERY

Robert Surton, Ph.D.

Cornell University 2014

The channel market model is a tool for making communication systems dependable. It is a generalization of the network stack model: Where the network stack model uses network graphs as the fundamental abstraction and layering as the compositional structure, the channel market model starts smaller, using channels as the fundamental abstraction, and builds more freely, using a marketplace for composition. In a channel market, a communication system is a channel transformer, which uses some of the channels offered in the market to implement new channels and offer them in turn. The model developed out of work on connection recovery for the Transmission Control Protocol (TCP), as a tool for understanding the complex dynamics of the standard network stack. In this dissertation, I apply the lessons learned from the channel market model back to TCP, and in particular to application-driven connection recovery.

Application-driven connection recovery is a technique by which a fault-tolerant application can recover and migrate connections, leveraging middleware to avoid modifications to its TCP implementation. The middleware depends on very little state, making application-driven recovery a lightweight and fast technique. To demonstrate what is possible, I present recovery middleware using both formal and empirical methods.

Formally, I present specifications of TCP and recovery middleware. The specification of TCP also serves as an introduction to the details of the protocol; to serve that purpose, the it follows a novel decomposition I developed for my own understanding while working on application-driven connection recovery. Using both specifications, I prove that the simple middleware is sufficient for a failing and recovering TCP to refine non-failing TCP.

Empirically, I present TCPR, an implementation of recovery middleware. I describe the systems problems that arise from masking connection failure and migrating without modifying TCP or sockets, particularly where the common interfaces violate the TCP specification or unnecessarily restrict what state is available to an application. I also present the results of a study of the Border Gateway Protocol (BGP), highlighting the severity of the routing disruptions that can be avoided only with connection recovery.

The channel market model's role in the presentation displays its usefulness in both ways that a scientific model can be useful: For understanding existing complexity (as in the decomposition of TCP), and for simplifying the design of the new (as in TCPR). I wrap up by presenting two design principles that have emerged from using channel markets: The separation of justification and the haggling principle. The channel market model and its design principles are useful tools beyond making TCP more dependable, and they stand waiting for future work.

BIOGRAPHICAL SKETCH

Robert Surton is a philomath from Portland, Oregon. Throughout his childhood, he was inspired by the examples and personal attention of several excellent educators. He was deeply influenced by his time at NRST, a magnet high school created by a small team of caring teachers for the sake of the intelligent, interested students who stood out a little too much in their mainstream schools. It was there that he met his wife, Elizabeth. They were brought together in the Flying Hedgehogs, an advanced student-run theatre troupe that inspired Robert's love of theatre.

Robert has always had a love of his homeland, spending countless hours on walks through his neighborhood or long backpacking trips through the wilderness. Most of his wilder adventures were in the company of good friends he made in Boy Scout Troop 598. Robert is an Eagle Scout and a Brotherhood member of the Order of the Arrow.

When it came time to move on to undergraduate studies, Robert chose to fly to Troy, New York for Rensselaer Polytechnic Institute, on the basis that of his favorite choices, it was the farthest away from his support structure back home. He explored diverse interests from fencing to West-African percussion, and continued his theatre work with the RPI Players. He also developed a new love, for the Brazilian art of Capoeira, whose music and physical challenge energized him while he finished his studies.

All along, Robert had been inspired to teach, and suffered from an interest in pursuing a project to its basics and building better wheels, which Elizabeth noted might be better served in graduate school than a cubicle. Robert sent in only one application, to Cornell University, and his risk paid off not only with acceptance to the school, but by many years there surrounded by great people, resources, and encouragement.

Initially, Robert concentrated on computer science, but he grew increasingly unable to stay out of the theatre. So, when it came time to choose a minor, he marched right in, was told there was no graduate minor in the theatre department, and stayed anyway. Through the friendship and personal attention of great people there, he flourished as an actor and playwright, and created the minor he needed.

Finally, Robert was drawn back home. Although Ithaca reminded him strongly of a small, isolated Portland, he brought his adventure in New York full circle and returned to Oregon. He now seeks to continue his research by learning and teaching.

ACKNOWLEDGMENTS

I am grateful most for Elizabeth, my partner. Her greatest contribution to this work has been always asking the right questions; whenever I am lost, a conversation with her always steers me in the right direction.

Robbert van Renesse is both my advisor and my friend. He has shaped my work in the deepest way possible, by providing the environment in which I did it. Even more, he is one of the few people with whom I would look forward to sharing a four-hour car trip. We have great ideas together.

Nate Foster and Ken Birman have been inspirational colleagues, opened doors, and worked in my committee beyond my expectations, making this dissertation a reality.

Beth Milles and my minor advisor Bruce Levitt might not have been aware of their beneficial influence on this work, but by facilitating my escapes to the theatre, they have significantly affected all aspects of my life. I am particularly thankful for having Beth around when my child was in the NICU, and I needed to balance my life with the zaniness of Commedia early in the morning.

My inspiration to study dependable communication is due to the Ludd project, a collaboration around a highly-available router. I am grateful to Robert Broberg for bringing us all together, keeping us happy, and giving us hard problems to solve.

When I first came to Cornell, I worked with Emin Gün Sirer, who taught me a lot about technical speaking, lively discussion, and rigorous writing. Fred Schneider has also shaped my perspective on rigorous writing through his example and personal attention, and at a crucial moment he set me on the right path with his knowledge about composing specifications.

Many people have done me the favor of reading my work and giving me their thoughts on it, and I thank them all. Shoshanna Cole and Justine Sherry have been particularly helpful. Finally, I thank those I have taught and those I have learned from—I valued every moment.

CONTENTS

LIST OF FIGURES

MAE. I don't think they are cymbals.
MEG. No?
MAE. I think … they represent somethin' else.

A Stye of the Eye,
Christopher Durang

CHAPTER 1

INTRODUCTION

The channel market model is a tool for making communication systems more dependable. I discovered it as a side effect of work on connection recovery for the Transmission Control Protocol, and in this dissertation I will apply the model back to connection recovery and to TCP in general. The channel market model promises a lot of interesting avenues for future work as well, some of which I will hint at in conclusion.

TCP is an important protocol, because the Internet depends on it. The Internet Protocol (IP) offers global addressing and best-effort delivery of packets across the world. It is TCP that starts from those unreliable, message-oriented channels, and constructs reliable, connection-oriented channels that then carry the Web (HTTP), email (SMTP), and even the routing infrastructure of IP itself (BGP). But if the Internet depends on TCP, how do we know it is dependable?

1.1 Dependability

Dependability is 'the ability to deliver service that can justifiably be trusted' (Avižienis et al. 2004). Justification is essential, because in practice, a correct system people don't trust is no more useful to them than a faulty one. One must prove to them that the system will meet their expectations.

It is worth distinguishing expectations from specifications. Dijkstra (1982) distinguishes two separate concerns in the task of 'making a thing satisfying our needs'. The first is 'stating the properties of a thing, by virtue of which it would satisfy our needs'. The second is 'making a thing guaranteed to have the stated properties'. The formally stated properties are a specification, and if the thing does not satisfy its specification then it is incorrect. However, if the specification does not correspond to one's expectations, then even a correct system can be surprising, and a surprising system is not dependable. As Knuth once warned, 'Beware of bugs in the above code. I have only proved it correct, not tried it'.

'Proving it' and 'trying it'—that is to say, formal reasoning and experimentation—are the basis of a scientific justification of a system. Both are essentially *modus ponens*, in the form Dijkstra suggested: First, show that some particular standard is dependable, and second, show that the dependability of the standard implies the dependability of the system.

In formal reasoning, the standard is an expression of the system's specification in some logic or calculus, and its correctness is an axiom. In order to show the implication, the system itself must also be represented in the same language, and then the proof is a matter of symbolic manipulation. Once such a proof is shown, it guarantees that the system will satisfy every aspect of its specification in any conceivable environment, so it provides significant confidence. However, there is no way to manipulate symbols that proves a mathematical object corresponds to expectations held by a real person, or to a system in the real world.

In experimentation, the standard is the past behavior of the system in the real world, and its correctness is justified by its success at meeting the expectations held by real people. That the system's past correctness implies its total correctness is justified by the exhaustiveness of the experiments. For example, one might experiment with a wide range of inputs, loads, and environments. Unlike formal reasoning, experimentation justifies beliefs about the real world. However, predicting the future is not only tricky in practice, it is known to be weaker than formal reasoning; indeed, the class of liveness properties is defined by being impossible to verify experimentally (Alpern and Schneider 1984).

Combining the two approaches in order to mitigate their weaknesses is what is known as the scientific method. People use models to abstract real phenomena into something understandable. Formal manipulation within the model leads to new facts with absolute certainty, but only to the extent that the model faithfully corresponds to the real world. Experimentation either reveals surprises that falsify the model, or, through continued success in a variety of environments, builds confidence in the model's predictions.

1.2 TCP

Is TCP dependable? Experimentally, it certainly seems so. It has been in heavy and widely varied use for decades, and although it has problems, even those are fairly well understood. Now and then something surprising happens, such as when TCP began to be used over high-bandwidth, long-delay subchannels, or over wireless subchannels in which loss does not necessarily imply congestion. In the sense that TCP has survived, it could be said to be dependable.

Unfortunately, the guiding principle of 'rough consensus and running code', while essential for building the Internet to begin with, did not result in a system that is amenable to reasoning, and sometimes a simple mistake can have significant consequences. For example, the Border Gateway Protocol (BGP), which interconnects networks to form the Internet, assumes that TCP failure implies the failure of its subchannels. That is not always true, and in chapter 3 I present measurements of the severe routing disruptions that can result.

One way to make a system more dependable is to adapt it somehow so that it can be justified to satisfy a stronger specification. To solve the problem of the dependence of BGP and other protocols on TCP connections that never fail, I have developed the technique of application-driven connection recovery. In chapter 4, I present a system called TCPR which implements application-driven connection recovery using middleware, enhancing the specification of TCP using an unmodified TCP implementation.

Another way to make a system more dependable is to take it apart and understand it better, so that whatever its properties are they are not surprising. Although TCP's main strength is experimental justification, there has been some effort at formally verifying aspects of TCP. The TCP specification itself (RFC 793) is not formal, but a detailed explanation in English prose. However, some projects have proven that simplified variants of TCP satisfy desirable properties such as reliably delivering data; a brief survey appears in section 5.6.

One notable effort combines a formal specification with experimental validation (Bishop et al. 2005a; Bishop et al. 2005b; Ridge, Norrish, and Sewell 2009). The result is a very complete picture of common network implementations as they behave in real deployments, given in a formal language with unambiguous semantics.

When a specification is intended primarily to help people understand a system, one of its most important features is the way the whole system is broken down into components. Each component must be simple enough to be easily understood at once, and the structure of their composition must be simple enough that they can be understood mostly in isolation. In chapter 5 I will give a formal specification of TCP that features a novel decomposition I discovered to be helpful when I had to reason about TCPR.

The specification of TCP is useful beyond explaining the features of TCP, as I show in chapter 6 by using it to prove that simple abstract middleware is sufficient for application-driven recovery. Beyond demonstrating that recovery is possible, the proof is the simplest statement of the requirements imposed on the middleware and the assumptions required of the application.

The main lesson I have learned from working with TCP and networking implementations is that the usual model, the network stack, was unnecessarily limiting. I discovered a model that was more useful to me, which I call the channel market model. While the new model came at the end of the work in this dissertation, I see the entire line of work as the first use of the model.

Internet	OSI	
Application	Application	7
	Presentation	6
	Session	5
Transport	Transport	4
Internet	Network	3
Link	Data Link	2
	Physical	1

Figure 1: The Internet and OSI network models side by side. Both are layered, preventing networks from building up mutually. They are also fixed in scale, so protocols that build topologies using TCP or UDP subchannels are relegated to being application-layer overlays rather than true networks.

1.3 Network models

The channel market model is a generalization of the network stack model. The difference between the models lies in modularization. When a system is too complex for a person to understand it all at once, they naturally resort to abstraction, which is essentially metaphor. They replace the complex system with a simpler one that is the same for their purposes. Modularization is divide-and-conquer metaphor.

In the network stack model, each module is a network. A network is a graph, representing agents (vertices) and their ability to communicate (edges). Networks are composed into a whole system by stacking them up in a single column. Each network provides the communication abstractions that become edges in the network above it. There are two major camps within the network stack model, the OSI model and the Internet model. They are shown side-by-side in figure 1.

Despite differences in detail, the OSI and Internet models build up in the same pattern. At the lowest level, communication systems connect computers to each other directly, creating networks. Global addressing and routing replace all of the lower-level networks with a convenient logical clique. End-to-end protocols build stronger guarantees on top of the clique. Everything else is an application.

The network stack model arose naturally from the development of the Internet, but it is not well suited for dependability. The Internet began as an attempt to connect the original ARPANET with the ARPA packet radio network, without violating their independent administration (Clark 1988). In that context, the network stack is a very direct metaphor. However, strict layering between networks is both mythical and impractical.

Layering is mythical both in the small and in the large. An example of a small violation is the TCP header, which, despite occupying the transport layer, contains a checksum over parts of the IP header, in the network layer below it. In the big picture, the routing protocol BGP runs on top of TCP, which would put it with 'everything else' in the application layer. However, BGP routers create the paths through networks that form the clique below the transport layer.

Layering is impractical because it is inefficient to respect it in concrete implementations. The IETF, the standards body of the Internet, considers layering harmful (Bush and Meyer 2002), because layers tend to duplicate each other's functionality (Tennenhouse 1989) and hide information necessary for global optimization.

The channel market starts from simpler modules—channels—and uses a more flexible compositional structure—markets—to provide a richer language. I describe the model in detail in chapter 2. In my experience, starting from smaller pieces prompts better understanding. However, just making modules as small as possible is not the best strategy; chapter 7 is a survey of design principles that have been successful, along with two new principles I associate with the channel market model.

A scientific model for dependable communication should enable the student to understand existing systems, without learning about exceptions, and enable the architect to create new systems that perform well, without requiring exceptions. The channel market model is promising as such a tool; I will describe the origins of the model and its first uses.

1.4 Contributions

The main systems contribution of this work is the introduction of the application-driven technique for connection recovery, which enables TCP recovery to be used with fault-tolerant applications, and with application protocols that do not match the assumptions of prior work.

The novel TCP decomposition presented in the formal specification is a valuable contribution for those who want to understand what TCP offers, from students to network researchers and programmers. By starting from smaller components, and then explicitly composing them, the concepts appear in more manageable units.

Finally, the main contribution that should be taken from this line of work is the channel market model, its concepts, and its design principles.

CHAPTER 2

CHANNEL MARKETS

Claude Shannon introduced the concept of the channel in his 'Mathematical Theory of Communication' (1948). He defined a channel as the medium used to convey a symbol between sender and receiver agents. A channel might be 'a pair of wires, a coaxial cable, a band of radio frequencies, a beam of light, etc.'. The most important feature of a channel is that, when it is observed, it conveys one particular symbol out of a set of possible symbols. It is the uncertainty of observing that symbol, as opposed to all the other symbols in the set, that conveys information.

I use an equivalent definition of a channel as a phenomenon from which agents can infer meaning. That meaning is the symbol, and all of the meanings that could have been inferred are the set. For example, when gunslingers agree to duel at high noon, the angle of the sun is a channel, and it can convey two symbols: Either it is directly overhead or it is not. The movement and rotations of celestial bodies need not be astrologically related to the gunslingers; it is the common meaning they infer that creates the channel.

Dependability has long been known to be an important aspect of a channel. For example, the ship that carried young men and women to be sacrificed to the Minotaur traditionally flew a black sail; when Theseus volunteered to go, with the intent to kill the monster and return alive, he promised to return under a white sail instead. Although Theseus killed the Minotaur, his escape was rushed and he forgot to change the sail. Seeing the traditional black sail, King Aegeus inferred that his son had failed, threw himself from a cliff.

One technique computer scientists have discovered that might have saved the Athenian king's life is redundancy. A more elaborate protocol might have used a second channel to convey the same symbol as the sail in a different way (error coding), or the king could have delayed his suicide until his suspicion was confirmed (acknowledgment), or a redundant agent could have been responsible for setting the sail channel if Theseus failed (replication). Because it is the way agents interpret channels that makes them so, agents can use existing channels to construct new ones with different properties.

That construction is the basis of a market. One way to think of a channel market is simply as set of all the channels that have been constructed. Agents can agree to use channels from the set in order to communicate in new ways, and offer the channels they construct by adding them back to the set. I call this the 'shopping list' view. At a more detailed level, every good or service available in a market is offered by somebody, to somebody. To implement a channel market means giving agents a mechanism to describe, send, and receive channels, through other channels, to share them with other agents. I call this the 'market stall' view.

Consider two examples of agents—a router and a web browser—and how they interact with a channel market.

Routers are agents that are included in a channel market so that each of the other agents can make a simplifying assumption: that there is a channel connecting each pair of them directly. One technique is for each router to send, through each of the channels it can manipulate, every channel it is aware of. Thus, each router eventually receives all of the channels originally available in the market, and can find paths to every connected endpoint. The routers offer the new channels—implemented using paths across the network—into the market, and other agents, such as web browsers, can build further using direct connections.

(When I described the routers sending channels to each other explicitly, that was the 'market stall' view. When I described the paths being offered in the market, generally, for other agents to pick up and use, that was the 'market stall' view.)

A web browser is another kind of agent, which solves the problem of downloading a web resource. It might be thought of as observing an input channel to receive a URL, and emitting on an output channel the value of the resource named by that URL. To offer that high-level communication, the web browser must construct it using existing channels. The URL gives the name of a host, which the browser sends through a channel to a DNS agent, which looks up the appropriate IP channel and sends it back through a channel to the browser. The browser sends the IP channel to an agent that constructs a TCP channel through it, and then similarly requests an HTTP channel implemented through the TCP channel. The browser can communicate over the HTTP channel to retrieve the resource and forward the result to its high-level output channel.

Note that by assuming the existence of agents providing DNS, IP, TCP, and HTTP, the web browser agent just needs to communicate with them (at their market stalls, if you will) to construct the communication resources necessary to fulfill its task. And it need not worry about every channel in the market (in the shopping list view); note that the IP channels the DNS agent offers were likely constructed by router agents that the browser does not need to know of.

The task of justifying trust in a network is simplified by its construction from channels found in the market. Each agent can be analyzed independently to show just one thing—that given the items on its shopping list, it can offer channels that correctly implement some specification.

2.1 Layers considered harmful

The traditional network model, the network stack, is a special case of a channel market. There are two well-known variants of the network stack: the OSI model (OSI) and the Internet model (Braden 1989). They both divide protocols into layers based on where in the network they are implemented. For example, in the Internet model, the link layer provides connectivity within local networks, the Internet layer connects such networks into a global address space, the transport layer adds end-to-end functionality, and everything

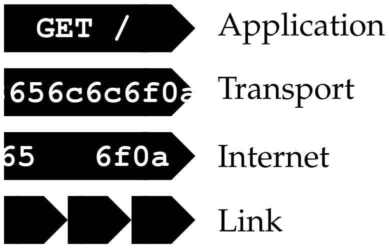

Figure 2: A network stack.

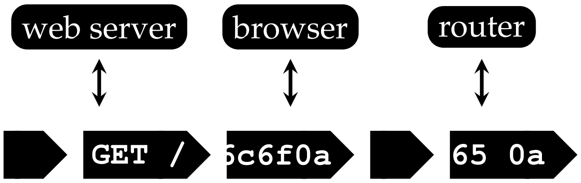

Figure 3: A channel market.

else is at the application layer. The two models correspond closely, as shown in figure 1, although the OSI breaks the application and link layers down into more detail. The OSI model also numbers each layer, which leads to terminology such as 'layer 7 switch' for a network device that inspects application layer data to make decisions.

Whereas the modular unit in a general channel market is a channel, the unit in a network stack is a network—a graph of agents and channels among them. And whereas in a general channel market an agent can in principle offer the channels it constructs to any other in the market, in a network stack the networks are arranged into layers and can only offer channels upward. Figures 2 and 3 show the two models for comparison.

There are two things wrong with network stacks being considered fundamental. First, layering puts unnecessary limits on how agents can compose their functionality. In fact, layering is considered harmful by the IETF (Bush and Meyer 2002), in defense of the less strict approach taken by their Internet model compared to the OSI model.

Second, and worse, the network stack is not an accurate model of existing networks—real-world protocols have dealt with the limits of layering by sidestepping it, so that although one of the advantages of layering should be that each layer can evolve separately, in practice they are very tightly coupled. For example, the TCP standard defines

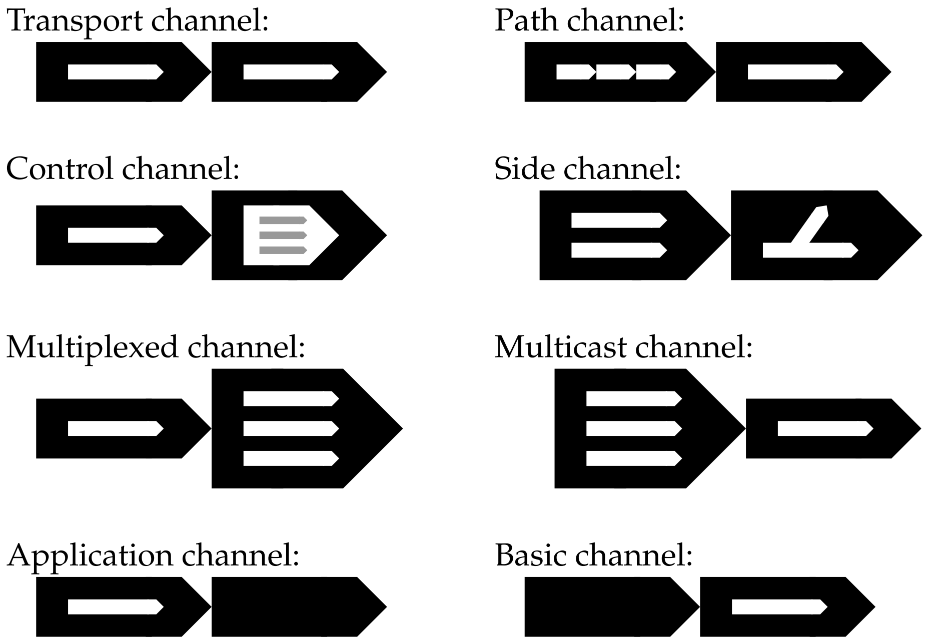

Figure 4: Common kinds of channels. Each icon represents subchannels being offered through a control channel on the left, followed by the constructed channels being offered through a control channel on the right.

the checksum for each segment to include header fields from the IP version 4 packet that carries it. A separate standard specifies how to compute the checksum for version 6. Using TCP over any other subchannel is undefined.

2.2 Keep what is useful

The network stack model does not persist because nobody is aware of its flaws, but because it is, in spite of them, a very useful tool for reasoning about communication. For example, having language for the concepts of layer 2 and layer 3 switches is so useful that when protocols like MPLS started to blur the lines between those layers, the language expanded to include a layer 2.5. However, the useful concepts are not inherent to composing networks in layers; they can be expressed in the more general channel market model as well. Figure 4 summarizes the concepts explained in this section.

Control channels

Control channels are the market stalls of a channel market. A control channel carries and describes other channels. Although it can be convenient to think of a channel market as a pool of resources to be dipped from and poured into, in fact there are only agents and channels. For example, a routing agent offers a path channel to an IP agent, which creates multiplexed channels from it and offers one to a TCP agent, and so forth. Or, an agent sends a domain name to a DNS agent, which makes an offer by sending back an appropriate IP channel. Control channels are the metachannels that enable agents to create and offer the resources that come to be seen, all together, as the contents of the market.

Path channels

A path channel makes a symbol observable to its destination through a sequence of subchannels, which in this context might be called link channels, with the cooperation of agents forwarding it from hop to hop. A path channel is usually offered by an agent participating in a routing algorithm, such as BGP or OSPF. If the agent discovers that the path is no longer connected, it can implement the same channel transparently using a different path of subchannels; thus, path channels are the abstraction that hides routing.

Multiplexed channels

When an agent presents a subchannel as multiple, separate channels, the offered channels can be called multiplexed channels. Usually, a multiplexed channel is implemented over a subchannel by adding a unique label that can be used by the destination to distinguish the different multiplexed channels based on observations of the subchannel. Based on the terminology of the abstraction being built, such labels can be called addresses, protocols, or ports. For example, IP provides 2^{64} channels by addresses, and it further multiplexes each of those channels into 2^8 channels by protocols, such as UDP or TCP. UDP and TCP, in turn, multiplex each one into 2^{32} channels by port.

The same technique is also invaluable for implementing control channels. For example, in the IMAP mail protocol, requests can be labeled with identifiers, and each response includes the identifier of the request it answers. The same pattern appears in the application interface in the TCP standard. When the application calls the receive function, it provides an identifier, which TCP includes with the response that carries the data. It is essentially a channel-based continuation passing style, such as might be commonly seen in the π-calculus, but both the TCP API and the IMAP protocol are implementing using single subchannels: The market is made possible by multiplexing.

Transport channels

A transport channel adapts a single subchannel to present a different abstraction. For example, TCP adapts a best-effort, packet-oriented subchannel into a reliable, connection-oriented channel. Ethernet adapts a constant-rate, symbol-oriented subchannel into a frame-oriented channel. TLS adapts a reliable, connection-oriented channel by adding authenticity and privacy.

A transport channel can also be useful when it offers the same abstraction as its subchannel, but hides its implementation so that the subchannel can be replaced.

The end-to-end principle (Saltzer, Reed, and Clark 1984) is one of the most successful principles already used in network design; restated for a channel market, it is the principle that whenever two agents require some property to hold for a channel between them, and that property can or must be provided by adapting it with a transport channel, then the same property should not be redundantly added to lower subchannels (unless doing so provides a convincing performance advantage). Thus, transport channels are one of the most common and important concepts in channel market design.

Side channels

A side channel provides information about another, otherwise independent channel. For example, a sniffer mimics the symbols sent on another channel. The statistics gathered by firewalls and network devices are crucial side channels for maintaining networks. Side channels carry checksums, timestamps, and acknowledgments. As control channels are metachannel channels, side channels are metadata channels.

Multicast channels

A broadcast channel replicates each symbol to all of its subchannels, usually with the purpose of delivering it to agents that would otherwise not be able to observe a common channel. Broadcast can also be useful when the subchannels all have the same destination, because it enables the channel to tolerate faulty subchannels.

An anycast channel replicates each symbol to any of its subchannels, usually with the intent that each destination is equivalent. Anycast can also be useful when the subchannels all have the same destination, because it enables the channel to exploit the combined capacity of the subchannels.

Application channels

An application channel communicates directly with the environment. The environment might be a different abstraction in different contexts, but it is always possible to recognize application channels by the fact that they are not subchannels of any other channel in the market.

Basic channels

Within any given channel market, normally there is a set of channels that are considered basic. Be it physical wire protocols, or Ethernet, or TCP, only a certain granularity is relevant to any given problem. The basic channels are the bottom layer in a layered

Channel
A phenomenon from which an agent can infer meaning.

Symbol
A particular meaning that can be inferred.

Agent
An entity that observes channels and reacts to symbols.

Market
Composition as a result of agents communicating channels.

Figure 5: The four fundamental concepts of the channel market.

model; whereas application channels are not subchannels of any other channel in the market, no channels in the market are subchannels of a basic channel.

The complement of 'basic' is 'overlay'. All of the non-basic channels in a market are composed of other channels, so they are all overlays. However, it is worth taking note any time a channel or network is described as being an overlay, or virtual. Both Resilient Overlay Networks (RON) and the Transmission Control Protocol (TCP) are overlays on IP, so why is RON an 'overlay' while TCP is a 'protocol'? I have often found it useful to hear 'overlay' as a signal that the system in question is being treated as second-class, and to search for the possibilities that are being implicitly ignored by not considering overlays normal.

2.3 Formalisms

Like the network stack model, the channel market model does not have a native formalism. It is a pattern that can be seen in a variety of places. There are four fundamental concepts of the model, shown with brief definitions in figure 5. The four concepts can be seen in any formalism commonly used to specify communication systems.

Generally, whatever changes is a channel. Often it is called state. By definition, the values the channels can take are the symbols. Agents are what can be seen as causing the changes to channels, based on observing other channels; sometimes agents are described explicitly, in which case they are often called processes, but often, only their behavior is specified, such as by next-state relations or actions or transitions. Each formalism represents a market by whatever mechanism it provides for composition, such as a parallel composition operator on processes, or a module system for composing specifications.

In π-calculus, which is a very close fit, a channel is a channel, a symbol is also channel, an agent is a process, and the market comprises all communication.

In I/O automata, a channel is a state variable, an agent is specified by its behavior through actions, and the market is represented by the hiding, renaming, and composition operators.

In TLA+, a channel is a state formula. An agent is not named explicitly, but its influence on channels is specified by a transition formula. Composition most often takes the form of logical conjunction and the module system, but TLA+ can also explicitly represent shopping in the market for subchannels with a particular specification, using the temporal implication operator.

Channels, symbols, agents, and markets can be found in other formalisms as well, in programming languages, in textbooks about network protocols, and in networking standards. I have found it an interesting endeavor to look for them.

CHAPTER 3

THE BORDER GATEWAY PROTOCOL

The channel market model is a valuable tool for *post hoc* analysis of complex systems. For example, the Border Gateway Protocol (BGP) is overlaid on TCP subchannels, and it suffers from a design flaw based on assuming more of those channels than TCP guarantees. Even when both BGP and TCP are correct, because BGP relies on an incorrect assumption about TCP, a correct TCP can seem faulty and the resulting BGP error can be catastrophic, which illustrates how important assumptions are to dependability. The flaw in BGP, which I characterize and measure in this chapter, was the motivation for the work in chapter 4 on connection recovery.

3.1 Setup

BGP agents are the routers at the edge of ISPs and other autonomous networks stitch the Internet together out of smaller autonomous networks. Each pair of neighboring routers maintains a TCP connection, which they use to synchronize their routing tables. If a router detects that one of its connections has failed, it assumes that its link to that neighbor has also failed, so it immediately withdraws all of its affected routes—even if the neighbor immediately reconnects. Once the connection is re-established, it cannot be used to carry network updates or inform routing decisions until each endpoint has transmitted its entire routing table. Meanwhile, as network routing re-converges to accommodate the supposed failure, which can take minutes, packet loss can increase 30-fold due to transient routing loops and black holes (Labovitz et al. 2000; Pei, Zhang, et al. 2006; Zhao et al. 2003; Sahoo, Kant, and Mohapatra 2006).

There is a standardized solution called Graceful Restart (GR). Because there are many reasons a connection might close other than link failure, a router can enable GR for a connection, informing its peer not to trigger re-convergence immediately when the connection fails. Instead, the peer will wait until a new connection is re-synchronized and the link is known to be down (Sangli et al. 2007). The time and update load to re-synchronize the connection is still necessary, but the routing disruption is avoided. Unfortunately, the assumption that the link survives can be just as wrong as the assumption that the link fails.

To demonstrate the problem, I evaluated connection failover in BGP. My experiments were inspired by Pei, Zhao, et al. (2004), who simulated networks of BGP routers to measure the effectiveness of various proposals at limiting the disruption of link failover. As in this prior work, I evaluate CLIQUE and B-CLIQUE topologies, but rather than studying link failure, I inject router failure followed by immediate recovery.

The BGP fault injection benchmarks were conducted using an experimental framework I implemented, which connects actual software routers and network stacks in a virtual network. All of the nodes run in network namespaces on a single host machine, which was not bottlenecked by CPU. Running on a single machine means that network latency is negligible, and all of the nodes share a precise wall clock. The network is simulated, but the routers run full BGP implementations.

In these experiments, most of the routers run the Quagga routing software and the Quagga implementation of BGP. However, to the best of my knowledge no open source BGP router supports recovering with GR. Accordingly, I added one node running `exabgp`, an easily-configured route injector that supports GR. This node also has a packet filtering ability that I exploit to artificially inflate the path length of some routes, enabling the exploration of scenarios with connected-but-undesirable backup paths.

I didn't modify the routers in any way, except for enabling network namespace virtualization. Each router thus had total control over its own routing table. I recorded the experiments using `rtmon`, a utility that is included with the standard Linux package `iproute2`. The result is a log of timestamped updates to each routing table. I also used `tcpdump` on the host bridge to record all of the BGP messages sent over the virtual network.

3.2 Control failover

Modern routers are often constructed from a collection of computing nodes. An internal node in a router that runs protocols such as BGP is called a control element, whereas a forwarding element runs the hardware engine and terminates physical links to peers. A common configuration for large routers is to have two redundant control elements, so that when one fails, the other can replace it.

In order to measure the disruptions caused by failover from one control element to the backup, I used a CLIQUE topology of 16 BGP routers, as shown in figure 6. The routers were all identical and each peered with all others. One router had an additional peering relation with the route injector. Once the initial convergence completed, a script killed and immediately restarted the route injector process, then observed the BGP network until it converged again. It is worth emphasizing that none of the links failed, only the BGP connection, so no routing changes were necessary. The results are based on data over more than 100 runs.

Figure 7 shows a CDF of convergence times in each experiment. On average, it took more than 15 seconds to re-converge to the original route, even though the underlying network topology was unchanged. During this period, I observed many events in which some of the routers believed some destination to be unreachable, or reachable through a path that is actually a routing loop. I report these periods in figure 8, showing a CDF of durations each router was unable to reach one or more destinations, as determined by analysis of the global collection of the routing tables at each instant. On average, every router in the system experienced more than 11 seconds of connectivity loss for each failover event, even though BGP itself recovered immediately.

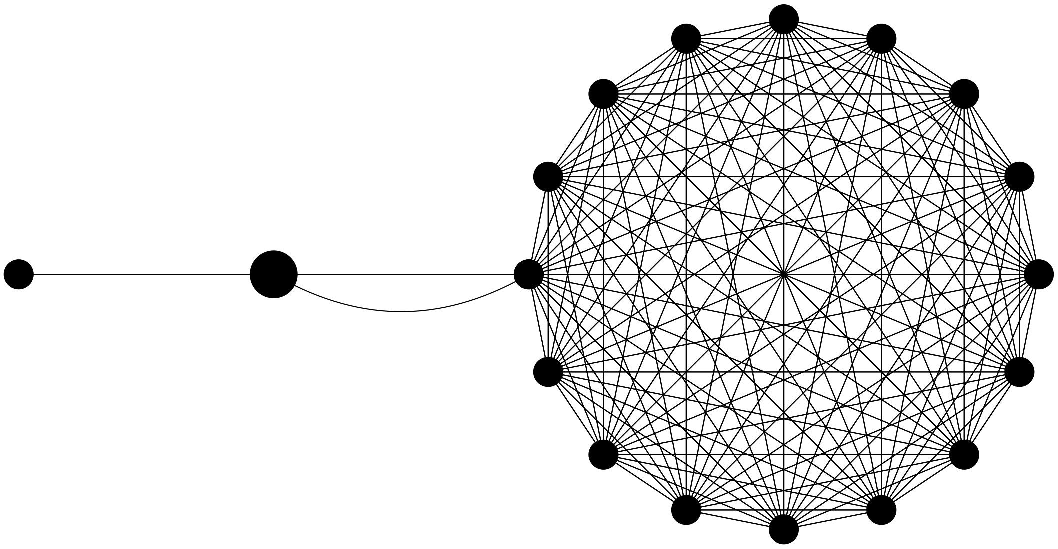

Figure 6: The CLIQUE topology.

In addition to disrupting the forwarding plane, re-convergence taxes the control plane by consuming bandwidth and CPU to process updates. Figure 9 shows the number of updates sent for each router for the single destination being advertised. In a core Internet deployment, where a routing table might include tens of thousands of destinations, the more than 45 updates per router would be multiplied by the number of destinations.

The vertical CDFs demonstrate the impact of using GR. With this feature enabled, the router announces that it will preserve its routing tables across restarts, and its peers continue to route traffic through it during restart. The routing flap seen with the basic BGP recovery is thus avoided and re-convergence is immediate. No network disconnections occur, and there is just a single redundant advertisement of each route. Thus, in these experiments, GR does nearly as well as possible, except for sending unnecessary route advertisements.

3.3 Forwarding failover

There are cases when GR can perform worse than totally unmasked restarts. GR is effective because it changes the way that peers interpret connectivity loss. In the default

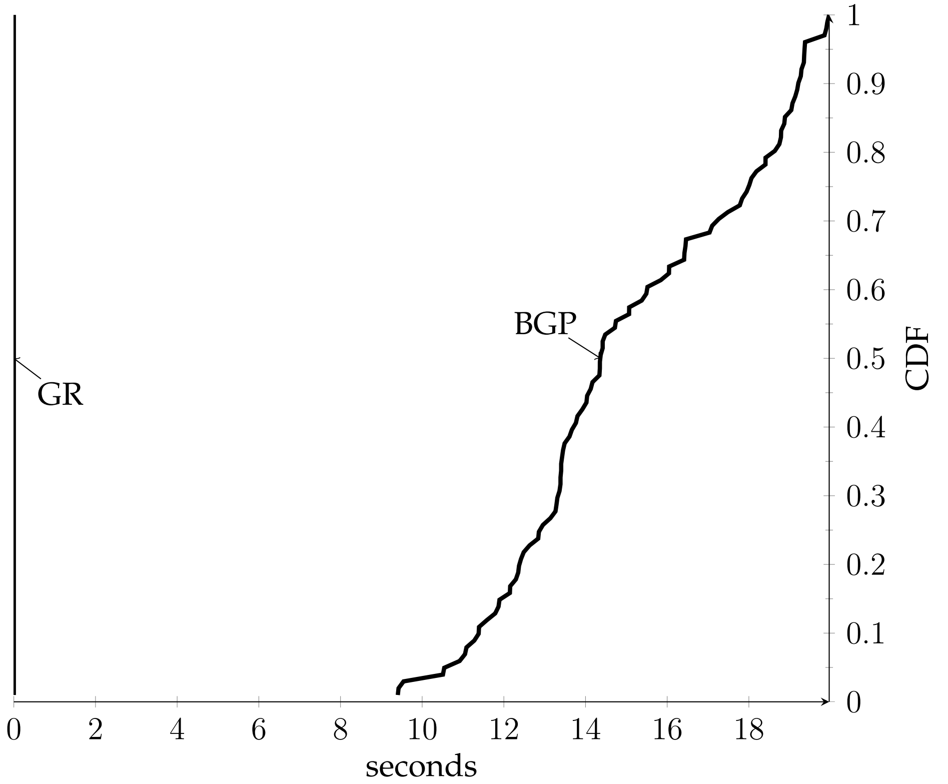

Figure 7: Duration of convergence due to control failure.

BGP behavior, connection loss is interpreted to indicate forwarding plane failure; with GR, the forwarding plane is assumed to continue functioning on 'autopilot', so that only the control plane requires re-synchronization. When this assumption is valid, GR almost solves the failover problem. However, when the network topology does change while the control plane is still recovering, GR can instead delay the needed routing adaptation. A consequence is that routers may use bad paths based on the assumption that the recovering router has applied a routing update that it has not had time to receive, process, and install in its hardware-mediated forwarding plane.

To experiment with forwarding failover, I modified the control experiment to use a B-CLIQUE topology, as shown in figure 10. A B-CLIQUE is the same as a CLIQUE, but with an additional backup path. A second router in the clique peers with the route injector, and receives a route to the destination that has an inflated path length, making it less desirable

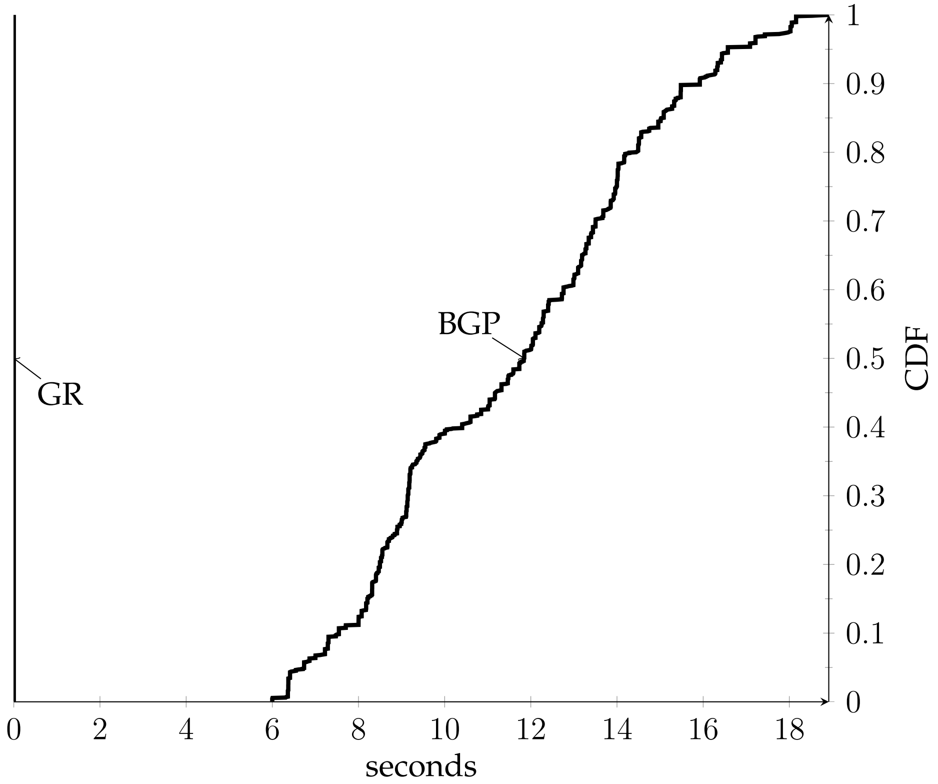

Figure 8: Average duration a router is disconnected from the destination due to control failure.

than any path through the primary link.

Although the control element fails over immediately, just as in the previous experiment, in this experiment its forwarding-plane link to the destination also fails, and does not recover. Network reconfiguration is necessary to switch over to the backup path.

As shown in figure 11, the convergence times for actual link loss are worse than for a pure control failover; on average, it takes more than 111 seconds for the network to completely switch over to the backup. The disconnectivity, shown in figure 12, is similarly inflated, with an average of 86 seconds outage. Finally, as shown in figure 13, an average of nearly 190 updates per router, per destination are required to reach convergence.

More important than the behavior of BGP under link failure, however, is the behavior of GR. In this experiment, the recovering connection with the primary link advertises many new routes, which get filtered out before reaching the main network but do serve

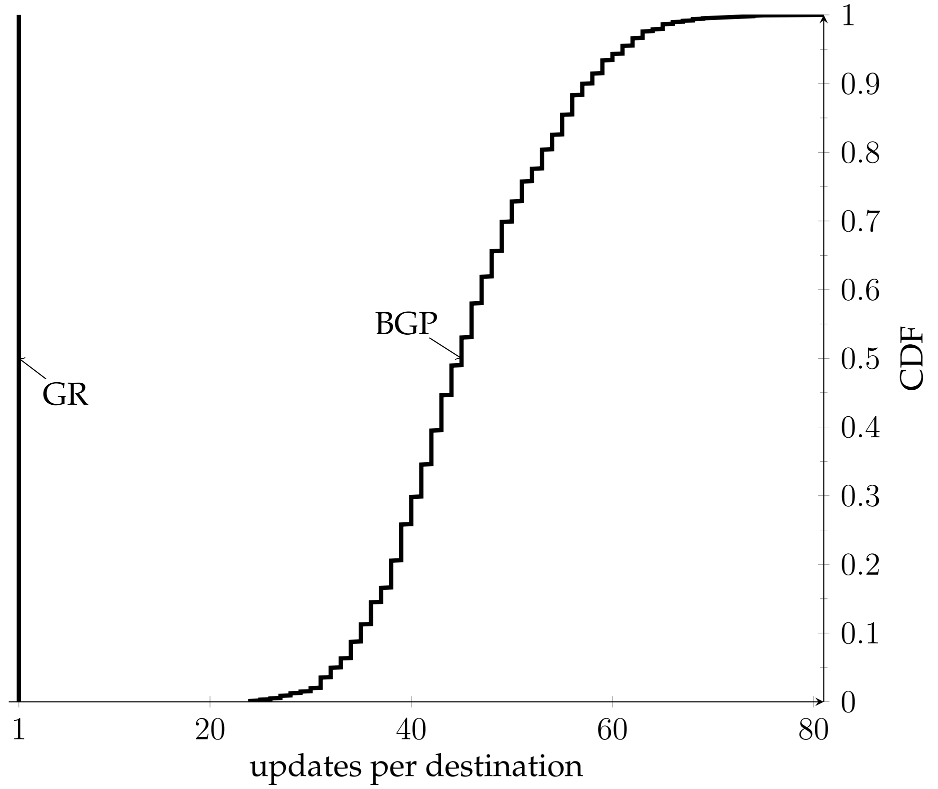

Figure 9: Update load due to control failure.

to prolong re-synchronization. During that time, GR prevents convergence from even beginning, even though an underlying network link has failed. Thus, the GR data can be seen to be similar to the raw case, but with a convergence delay increased by more than 100 seconds. The delay is not a function of the topology; it is the duration required to transmit the routing table. By doubling the number of additional routes to be advertised, the experiment could induce 200 seconds delay instead. Core Internet routing tables are far larger than those used in these experiments, and growing.

3.4 Conclusions

An important parameter in BGP convergence experiments is the Minimum Route Advertisement Interval (MRAI), which is a rate-limiting knob in BGP. It has been shown that up to a certain value, the MRAI improves convergence times, but past that, conver-

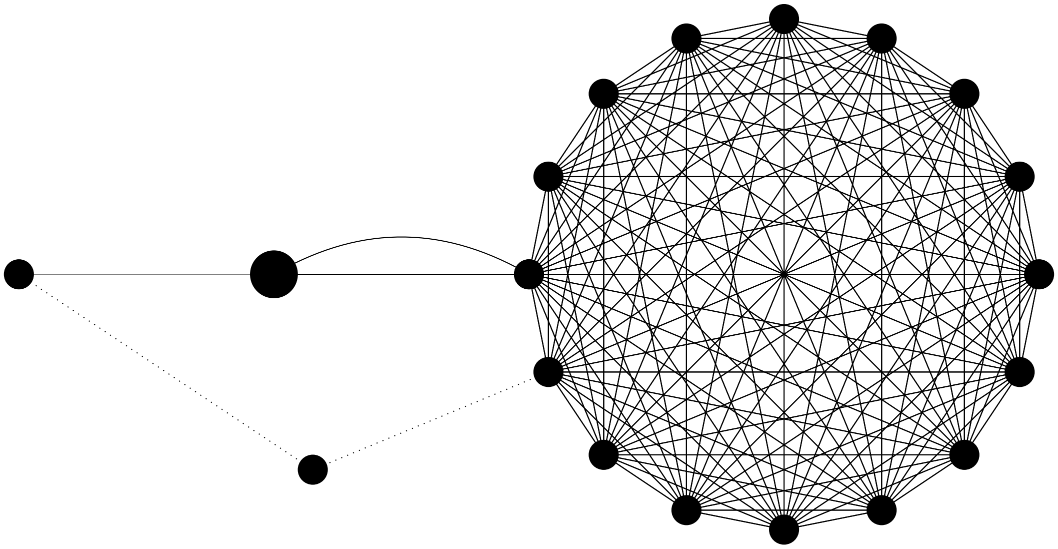

Figure 10: The B-CLIQUE topology.

gence times degrade. It is hard to determine the optimal MRAI for a particular topology, and unfeasible for the entire Internet; the original recommendation for MRAI in peerings between providers was 30 seconds, but newer experiments have indicated 5 seconds is better (Jakama 2008). My data was collected with an MRAI of 5 seconds. I also experimented with an MRAI of 30 seconds, and observed even longer convergence times and greater disruptions to network connectivity.

Pei, Zhao, et al. (2004) have studied the disruption caused during BGP convergence, explaining how topology and configuration affect the result, and evaluated a number of proposed strategies for making convergence faster and less disruptive. There has also been work that, rather than making convergence itself faster, ensures that routers can rapidly fail over to temporary routes until convergence is complete (Bonaventure, Filsfils, and Francois 2007). Both branches of work are valuable when convergence is unavoidable, such as when the topology genuinely changes.

My evaluation of BGP uses a novel measurement framework to demonstrate that unmasked failover can result in tens or hundreds of seconds of outages, even when it is not necessary to re-converge at all. Graceful Restart, although beneficial in some cases, can

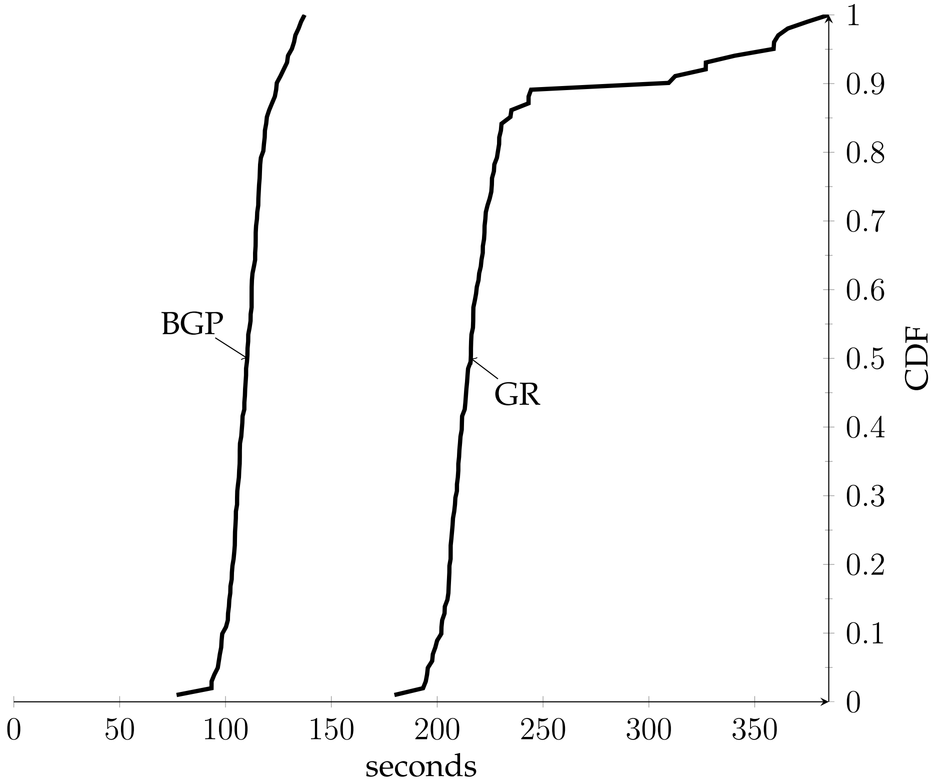

Figure 11: Duration of convergence due to correlated control and forwarding failure.

arbitrarily delay convergence when forwarding and control failures are correlated.

Ironically, a high-availability router might benefit from a design that increases the correlation between such failures. Forwarding elements in large routers usually have general-purpose CPUs and memory, in addition to the specialized forwarding hardware. The general-purpose hardware exists to handle slow-path forwarding decisions and configuration tasks, but is usually over-provisioned. Offloading control element responsibilities onto parts of forwarding hardware can dramatically increase the replication possibilities, from one or two dedicated control nodes to every component in the router. With more replicas come more frequent faults, but also additional fault-tolerance and greater parallelism for distributing workloads (Agapi et al. 2011). Nonetheless, restructuring a router to exploit the extra resources would be unwise with Graceful Restart, because any forwarding element failure would be correlated with the failure of some control element

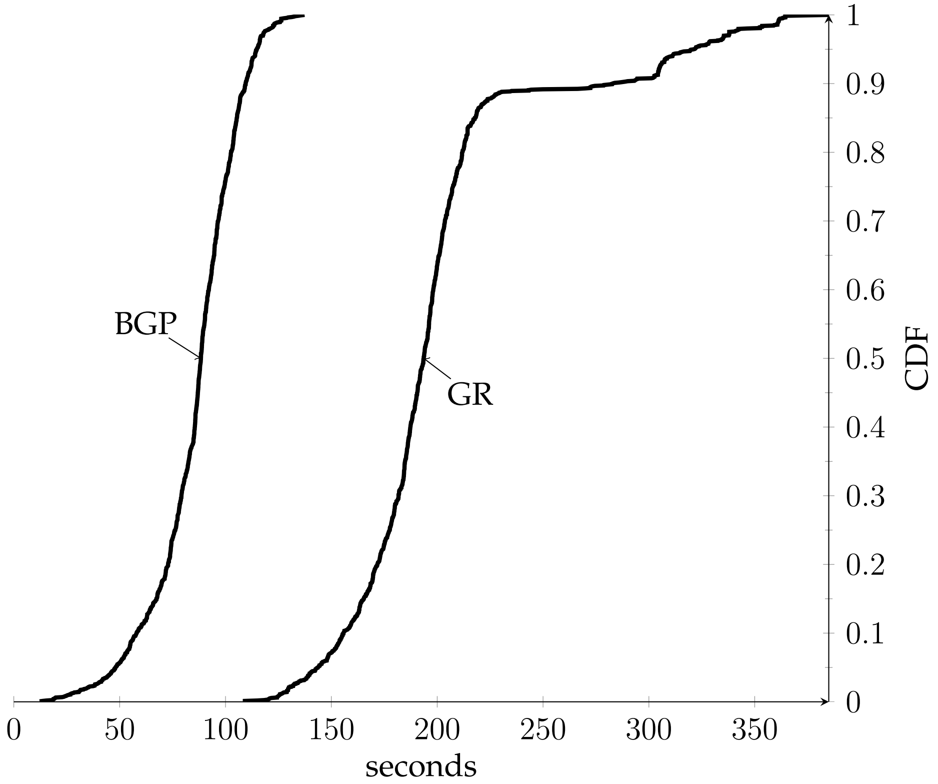

Figure 12: Average duration a router is disconnected from the destination due to correlated control and forwarding failure.

functionality.

The problem is in the assumptions BGP makes about TCP. The state of a peering is tied to the fate of the TCP connection, even though a TCP connection is not guaranteed to fail if and only if its subchannel fails. Nor, as GR assumes, is a TCP connection guaranteed to fail if and only if the subchannel does not fail. Rather, TCP connections in a distributed system can sometimes fail over internally, and the only way to avoid routing disruption without correcting the way the BGP protocol depends on TCP is to recover connections transparently.

BGP is an important application for TCP recovery, because broken connections trigger disproportionate amounts of disruption for core Internet forwarding. Furthermore, BGP stretches or breaks some crucial assumptions made by prior work on connection recovery, including determinism, whether the application can be a TCP client, and limitations on

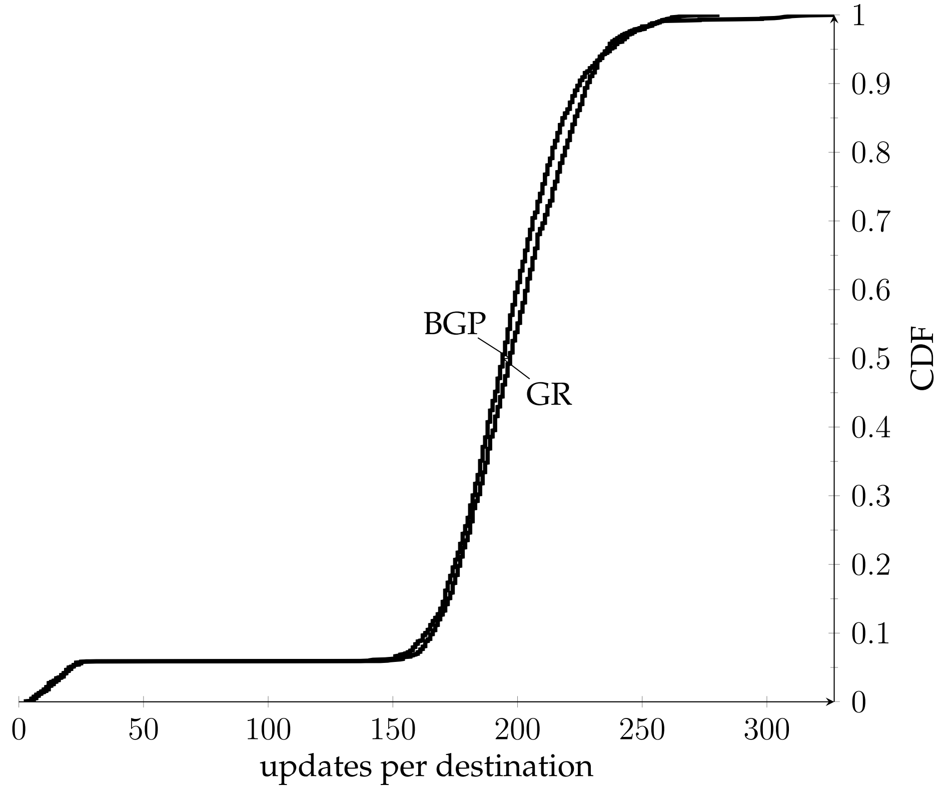

Figure 13: Update load due to correlated control and forwarding failure.

the length of its input. In the next chapter, I introduce an approach that addresses these challenges: application-driven recovery.

CHAPTER 4

APPLICATION-DRIVEN RECOVERY

The state that defines a TCP connection is generally encapsulated within a local implementation of TCP, while the usual socket interface provides no mechanism for an application to checkpoint, recover, or migrate that state. Unfortunately for an application such as HTTP or BGP, written standards and legacy implementations prevent using a session layer to decouple application state from connection state, so TCP is a point of vulnerability for the entire system.

In HTTP, for example, a client might submit a request to a distributed web application in the cloud. If the request modifies application state, but the connection is reset, the client can neither assume that the request was processed nor safely resubmit it. Chapter 3 shows how significantly connection loss in BGP can destabilize the Internet.

Application-driven connection recovery is a technique in which middleware adapts the subchannel used by TCP, adding a side channel so that a fault-tolerant application has access to sufficient connection state, enabling the application itself to mask failure and put an existing connection into a new local TCP.

4.1 INTRODUCTION

This work on connection recovery originated in a collaboration with Cisco. The project had the overall goal of developing a highly-available prototype based on the CRS-1 router (Agapi et al. 2011). Given a cluster manager and a BGP implementation that could migrate within the router and survive hardware failures, it became necessary to find a way for the TCP connections to migrate with it, and do so transparently to avoid unnecessarily disrupting the control plane.

This chapter describes a new technique for tolerating connection failure, in the face of a local TCP with no interface to enable it, and legacy remote applications that preclude graceful re-connection. BGP will serve as the example, both because it motivated the work and because it challenges many assumptions of prior work in the area, but the technique is applicable to any use of TCP.

To understand how prior work could be effective for HTTP but not for BGP, it is worth distinguishing between recovery and fault masking (Avižienis et al. 2004). Consider a technique in which a backup TCP peer is kept synchronized with the one used by the application. When the primary peer fails, the backup replaces it. The fault has been masked, because the remote peer need not be aware of the failover from primary to backup. However, recovery has not taken place, because there is no longer a backup, and a future fault will cause a failure.

In order to recover, prior work starts a fresh copy of the application and a fresh TCP, then replays all of the input received from the remote peer since the connection was established. Under the assumption that the application is deterministic, the result is a valid backup replica.

The benefit of that replay-driven approach to recovery is that the application can be treated as a black box, requiring no modifications. However, not only does the approach assume a deterministic application, it also assumes that replay is practical. The input to an HTTP server is usually a short request. On the other hand, BGP connections persist for the lifetime of a peering between two routers, and often carry thousands of update messages per minute. Replaying such a connection quickly becomes more burdensome than failure.

By contrast, I have developed an application-driven approach. The prototype, TCPR, is network middleware not unlike a NAT box. It sends state gleaned from packets to the application, and obeys the application to manipulate packets for recovery. By accepting minor changes to the application, application-driven recovery avoids the much greater burden of accepting responsibility for the application's state. Furthermore, by assuming the application is checkpointing its connection state along with its other state, TCPR is left with little to do but enable the application's operations, resulting in a lightweight, simple implementation.

4.2 Design

Consider a fault-tolerant application that can recover its own state, for example using checkpoints, but depends on TCP connections that have inaccessible state within a TCP implementation. I will refer to the fault-tolerant application as 'the application', to its TCP implementation as 'the local TCP' or just 'the TCP' when it is clear from context, and to the remote endpoint of a connection as 'the peer'. I assume nothing of the local TCP, the application's interface to it, or the remote peers, other that what is guaranteed by the TCP standard (RFC 793).

A TCP connection consists of two independent streams of bytes, one from the peer to the remote peer and one in the opposite direction. A TCP stream is reliable, in the sense that each byte is delivered exactly once, in order. To that end, each packet bears both a sequence number, indicating the position of its first data byte within its stream, and an acknowledgment, indicating the next sequence number expected in the opposite direction. A TCP buffers each byte it sends, retransmitting it as necessary, until it receives an acknowledgment with a later sequence number.

When a connection is established, each TCP chooses an initial sequence number at random, in order to avoid confusion with packets that might still be in the network from a previous connection between the same addresses. A packet with the SYN flag establishes a new stream and sets the initial sequence number; for example, after the handshake in figure 14, the local TCP's first data byte will have sequence number 401 (a SYN counts as a byte sent), and the peer's first data byte will have sequence number 301. A packet too far in advance of the expected sequence, or which bears a SYN flag even though the

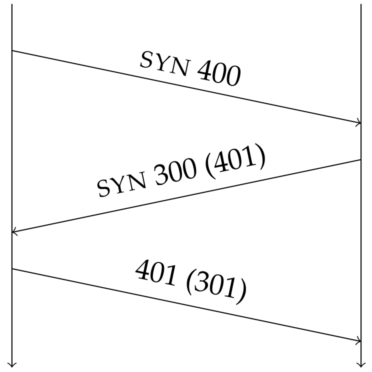

Figure 14: A TCP connection begins with a SYN, SYN–ACK, ACK handshake that establishes each endpoint's initial sequence number.

connection has already been established, is considered unacceptable.

TCP uses unacceptable packets to drive a form of connection recovery without fault masking. If a TCP with an established connection receives an unacceptable packet, it replies with a control packet, indicating the sequence number and acknowledgment it believes are current. However, if the connection is not established, the peer replies by acknowledging the unacceptable sequence number with the RST flag, notifying the remote peer and causing it to abort its connection. If a recovering client attempts to reconnect, the connection will recover as shown in figure 15. The outcome is that the old incarnation of the connection is aborted, and the two peers establish a new one.

The design of TCPR builds on TCP's notion of recovery, and makes it transparent by interposing middleware between the peer and the network peers, as shown in figure 16. The middlebox, TCPR, communicates with the application both implicitly, through the behavior of the TCP peer, and directly, through a side channel. TCPR maintains state for each connection, of which the application also maintains a copy; the side channel exists to synchronize those copies of the connection state.

The goal for TCPR is to enable recovery with the simplest possible middlebox and the least burden on the application, in terms of code modification, per-connection state, communication on the side channel, and overhead versus unprotected TCP.

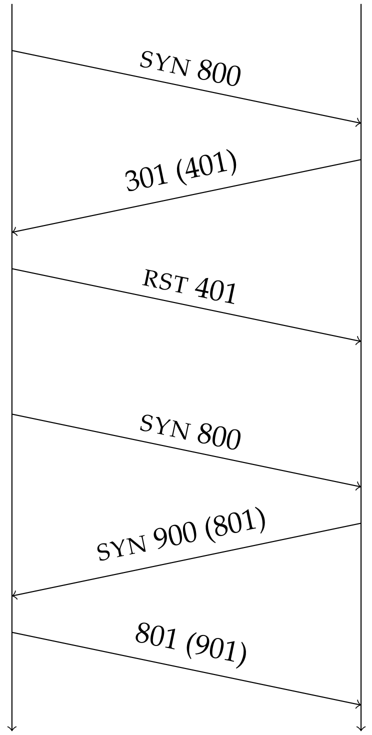

Figure 15: Continuing after figure 14, the client fails and reconnects. Its SYN is unacceptable, so the server replies with an empty packet. The reply is in turn unacceptable to the client, which does not yet have a connection, so the client sends a RST and the server deletes the state of the old connection. When the client retransmits its SYN, they establish a new connection.

Resynchronizing

As in standard TCP, the application signals its desire to recover by reopening the failed connection. when its network stack sends a SYN in the middle of an established connection. When TCPR receives the SYN packet in the middle of an established connection, it infers that the application is recovering, and rather than revealing it to the peer, TCPR intercepts the SYN and establishes a new incarnation of the connection locally, as shown in figure 17.

As with any new connection, TCP chooses an initial sequence number in a manner deliberately designed to be unacceptable to the peer. In order to splice the new and old connections back together, TCPR's per-connection state includes the acknowledgments

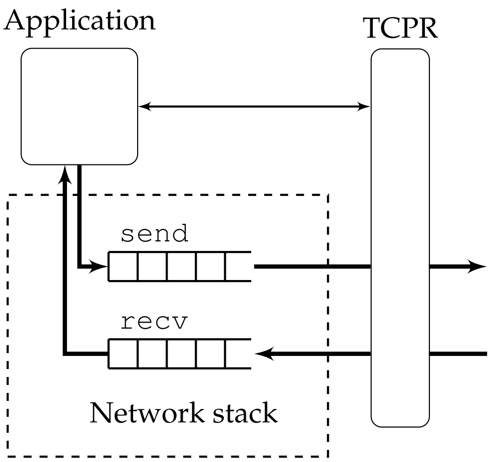

Figure 16: TCPR is a packet filter interposed between the application's network stack and peers, which allows the application to initiate connection recovery in a manner transparent to the remote end-point.

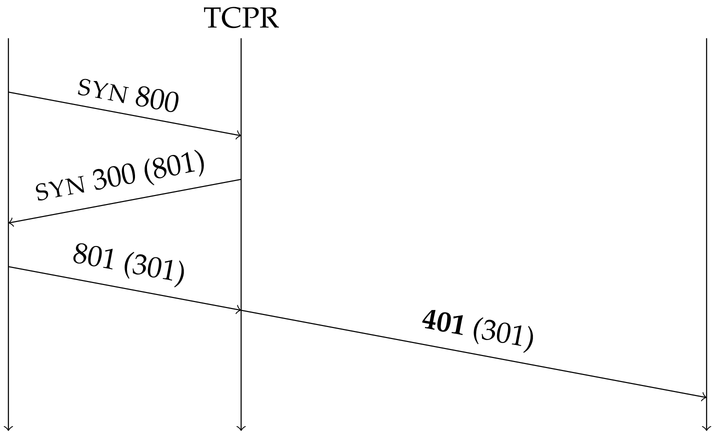

Figure 17: TCPR tracks acknowledgments, so that it can immediately answer a recovery SYN. If the client from figure 15 were an application using TCPR, TCPR would establish a new connection locally and splice it back to the original, so neither network stack is aware of the recovery.

sent by the local TCP and the peer, *ack* and *peer_ack* respectively.

By definition, the peer expects that *peer_ack* will be the next sequence number it receives. Suppose the new connection begins from some other value, *seq*. TCPR computes $\Delta = seq - peer\_ack$, and for the life of the new connection, subtracts Δ from the local TCP's sequence numbers, and adds Δ to the peer's acknowledgments. Thus, translation occurs through small header modifications on packets in flight, much as a network address translator remaps addresses and ports.

The opposite stream is simpler to resynchronize. TCPR chooses its initial sequence number as $ack - 1$, so that the local TCP's new incarnation expects the peer to continue from *ack* just as before (the decrement is due to the SYN flag implicitly occupying a position in the sequence).

However, TCP's reliability depends not only on sequence numbers, but on the send and receive buffers at each endpoint, which enable data to be retransmitted until it is safe at the remote endpoint. If the application migrates to a new machine or its network stack loses its state, the lost buffers must be recovered.

Recovering the send buffer

When an application calls `send` in the sockets interface, success only indicates that the argument has been copied into the send buffer in the local TCP. Should the send buffer be lost, some of its contents might not yet have been sent, or might be dropped in the network. Outside the TCP, only the application itself knows what it intended to send, so TCPR depends on it to replay what might be lost.

The application can use TCPR to learn how much of its output is safe at any time, and this information is necessary after resynchronization in order to know what to re-`send`. To do so, the application requests the latest state from TCPR, which includes *peer_ack*. To translate from sequence numbers to total bytes sent since an arbitrary checkpoint, the application can simply subtract the value *peer_ack* held in that checkpoint.

Note that unacknowledged data is not necessarily lost in a failure. Some might have been delivered to the peer, although no acknowledgment had arrived by the moment at which the local TCP lost its state. Thus, the application must repeat whatever it sent the first time. Generating all output deterministically is sufficient, but isn't necessary. For example, it is also sufficient for the application to checkpoint any data it is preparing to send, so that it recovers not only the old value of *peer_ack*, but at least enough buffered data to recover. Beyond that point, the application's output is unconstrained. On the other hand, deterministic output does not need a buffer. TCPR enables the most efficient choice based on specialized knowledge about each connection.

Recovering the receive buffer

When data arrives from the peer, the local TCP buffers it until the application consumes it with `recv`. The TCP standard considers such data safe to acknowledge immediately; once acknowledged, the peer will remove the data from its send buffer. However, the application might not yet have invoked `recv` and obtained the data, let alone checkpointed it. Accordingly, TCPR intercepts and modifies acknowledgments to ensure that unsafe data will not be acknowledged, relying on the application to tell TCPR when received data is safely checkpointed. Delayed acknowledgments were introduced with FT-TCP (Zagorodnov et al. 2009), which acknowledges packets only after it has checkpointed them into a 'stable buffer', all hidden from the application. Putting the application in charge enables more flexibility; for example, the application could choose to checkpoint raw input immediately, or to process whole application-layer messages and checkpoint the resulting state changes—if some input causes no significant state changes, the application could acknowledge it without waiting for a checkpoint at all.

Delaying an acknowledgment can inflate the peer's estimate of the round-trip time of the connection. However, most TCP implementations already delay acknowledgments by up to 500 ms to conserve bandwidth and prevent 'silly window syndrome' (Braden 1989). Zagorodnov et al. (2009) evaluated a variety of strategies for generating delayed acknowledgments from an advancing checkpoint; TCPR uses the strategy they call 'Delayed', which provides the best throughput for the fewest packets.

By putting the application itself in charge of its acknowledgments, TCPR lifts the end-to-end argument (Saltzer, Reed, and Clark 1984) for TCP's reliability from the host level to the application level.

HANDLING OPTIONS

TCPR supports the most common TCP options. The TCP standard leaves up to 20 bytes in the TCP header for such options, and specifies three that all implementations must support: No-Operation, End of Option List, and Maximum Segment Size. The first two are used to manipulate padding, so they have no impact on a connection's state. An endpoint may advertise Maximum Segment Size with its SYN packet to negotiate a packet size that avoids IP fragmentation. TCPR simply passes the value through, and records it to ensure that the same negotiation takes place during recovery.

RFC 1323 (Jacobson, Braden, and Borman 1992) described the first additional options to be defined, Window Scaling and Timestamps, which support high-latency, high-bandwidth networks. To support Window Scaling and Timestamps, TCPR passes the parameters through and saves them for recovery, just as it does with Maximum Segment Size. The authors of RFC 1323 noted that the only previously non-padding option, Maximum Segment Size, was only sent on SYN packets, so they worried that buggy TCP implementations might erroneously fail to handle unknown options on normal traffic. To address that concern, they established the convention that TCP options are negotiated in the handshake or else disabled. The result for TCPR is that suppressing unknown options on a handshake will generally avoid the need to suppress them any further.

TCPR also supports the Selective Acknowledgments (Mathis et al. 1996) option, which enables the TCP to acknowledge data that it receives out-of-order or with gaps. Advancing the actual acknowledgment would erroneously cover the gaps, but failing to acknowledge the received data might force the peer to wastefully retransmit data that wasn't really lost. At first glance, Selective Acknowledgments might seem incompatible with TCPR's delayed acknowledgments. However, the standard specifies that Selective Acknowledgments are purely advisory: although they serve as notification, the sender is still responsible for eventually retransmitting that data if the cumulative acknowledgment never catches up. Thus, it suffices for TCPR to apply Δ to the peer's selective acknowledgments as well as to its ACKs.

Masking failure

TCPR cannot depend on the local TCP to accurately distinguish between failure and the application closing the connection. During failover or migration, the application has (conceptually if not in fact) two unsynchronized local TCPs—the new one, in which the connection is not yet established, and the old one, in which the connection is no longer established.

Until the new local TCP is synchronized, any packet it receives will be unacceptable, and it will send a RST as discussed in figure 15. An endpoint with an established connection never sends a RST, so TCPR drops it to prevent a 'Romeo and Juliet' scenario: if the peer received the notice that the application is dead, it would abort the connection just as the application came back to life.

Failure can also be revealed to the peer if the old local TCP tries to clean up by closing the connection, which often happens when only the application process fails.

TCP endpoint closes its output by sending a packet with the FIN flag, which occupies a byte at the end of the stream and must be acknowledged by the remote endpoint, like the SYN at the beginning. At the packet level, there is no way to distinguish whether a FIN indicates failure or a deliberate call to `close`. Prior approaches have interposed on the network stack's interface to learn when the application closes a connection deliberately. TCPR uses a more explicit approach: the application sets a flag, $done\_writing$, just before it closes. TCPR treats a FIN as spurious if and only if that flag is clear.

TCPR responds to a spurious FIN with a RST, in order to enable the application to recover quickly in the case where the old and new local TCPs are the same. The RST causes TCP to abort the connection and become ready to recover it immediately.

Closing input

To deliberately close the stream in the other direction, the application sets another flag, *done_reading*. When *done_reading* is set, TCPR does not delay acknowledgments. The local TCP is free to acknowledge any remaining data, notably including the peer's FIN, so the peer can close gracefully.

If the FIN is the only byte remaining to be acknowledged, the application could instead advance *ack* by one byte. TCPR provides *done_reading* to echo the behavior of `shutdown`, and it also enables experimental setups without delayed acknowledgments.

Recovering after closing

As with `send`, a successful `close` indicates only that the FIN is in the send buffer. It might take some time for all of the data to be sent and acknowledged. Even once the final acknowledgment is sent, TCP implementations wait, usually 120 seconds, to be sure that the acknowledgment arrives and to handle any straggling packets. What if the application crashes after abdicating its socket?

TCPR sets a flag, *done*, when it thinks both flows are closed, and another, *failed*, when it has detected that the local TCP has failed. The application can check at any time whether the connection is really closed on the wire. If necessary, the application can recover as normal. Upon recovery, TCPR always unsets *done_writing* to give the application the chance to call `close` again.

Once *close* has been set for an appropriate duration (such as 120 seconds) the application explicitly instructs TCPR to delete its state. Of course, there is no reason the application has to wait, and the ability to control when TCPR deletes its state also makes it easy to experimentally inject failures.

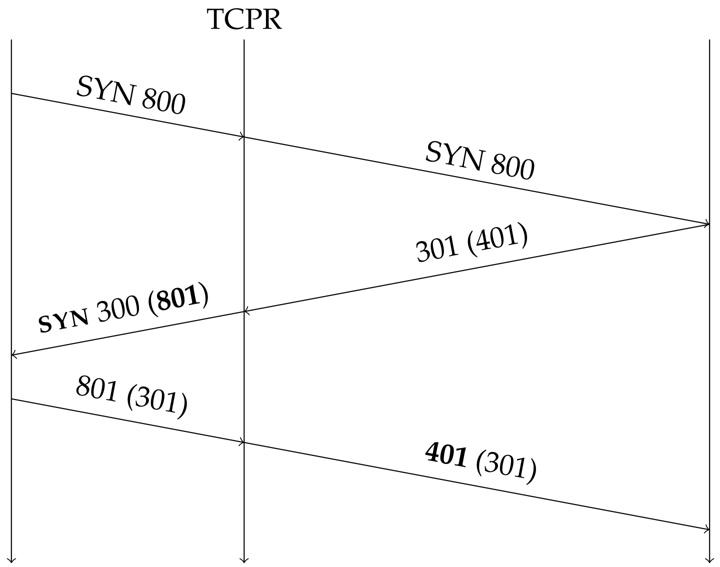

Figure 18: When TCPR fails over, it depends on the application for some of its state, and can recover the rest from the peer itself. For example, if TCPR cannot immediately answer a recovery SYN as in figure 17, sending the unacceptable packet to the peer prompts it to fill in the missing state.

Recovering TCPR

Should TCPR itself fail, it can recover some of its state, such as $peer\_ack$, by observing packets on the wire. The remaining state, such as the latest *ack* for delaying acknowledgments, is provided by the application. TCPR avoids the need to replicate any of its own state because recovery is driven by a fault-tolerant application.

For example, if the application is trying to recover but the latest $peer\_ack$ is missing, TCPR delivers the recovery SYN to the peer uncorrected; the peer's answer reveals the desired value, as in figure 18. Resynchronization takes place as in figure 17, but with an additional round-trip to recover soft state from the peer.

That default behavior also avoids the need for a special case to detect whether a connection is new or recovering. If $peer\_ack$ is missing, either TCPR crashed and lost it, or the connection is new and it doesn't exist yet; in the former case, the peer provides the missing value, and in the latter, it sends its own valid answer to the handshake.

On the other hand, fields such as ack and $done\_writing$ cannot be inferred from packet-level observation, because they are inherently controlled by the application. Neither can fields such as the saved values of options, which are advertised only once. Thus the application is expected to reset these values through the side channel it has with TCPR.

The state that depends on the application changes much more slowly than the soft state recoverable from packets. Whereas *peer_ack* is updated with every packet from the peer, all of the application-dependent state is either fixed at connection establishment (such as the peer's TCP options), set occasionally by the application itself (*ack*), or set by the application once when the connection closes (*done_reading* and *done_writing*). Thus, both the extent of the modifications to the application code and the communication overhead of maintaining TCPR's copy of the state are minimal.

4.3 Implementation

The TCP-manipulating core of TCPR is implemented as a portable C library. The simplicity of application-driven recovery is reflected in the fact that the library's single file contains only about 150 lines of code (measured by semicolons). TCPR is free software under the BSD license. Source code and documentation are available from:

`http://github.com/rahpaere/tcpr/`

I have experimented with TCPR using a variety of techniques to interpose on packets; the current prototype is a loadable module for the Linux kernel firewall, iptables. The system administrator writes rules to match packets to and from the application, delivering them to TCPR rather than dropping or forwarding them.

TCPR's per-connection state appears in figure 19. Notably, there is no buffered data. Both TCPR and the application keep exactly one `struct tcpr_ip4` per connection. The network-independent state is in `struct tcpr`, while only the subset of the state in `struct tcpr_hard` is crucial for recovery—if any field outside `struct tcpr_hard` is missing, TCPR will recover it on the fly.

```
struct tcpr_hard {
   uint16_t port;
   struct {
      uint16_t port;
      uint16_t mss;
      uint8_t ws;
      uint8_t sack_permitted;
   } peer;
   uint32_t ack;
   uint8_t done_reading;
   uint8_t done_writing;
};

struct tcpr {
   struct tcpr_hard hard;
   uint32_t delta;
   uint32_t ack;
   uint32_t fin;
   uint32_t seq;
   uint16_t win;
   uint16_t port;
   struct {
      uint32_t ack;
      uint32_t fin;
      uint16_t win;
      uint8_t have_fin;
      uint8_t have_ack;
   } peer;
   uint8_t have_fin;
   uint8_t done;
   uint8_t failed;
   uint8_t syn_sent;
};

struct tcpr_ip4 {
   uint32_t address;
   uint32_t peer_address;
   struct tcpr tcpr;
};
```

Figure 19: TCPR state structures. The application keeps one `struct tcpr_ip4` for each TCP/IP connection, but only the 14-byte portion defined by `struct tcpr_hard` is necessary for recovery.

The side channel is a UDP connection, and the protocol consists only of entire states sent back and forth. The state is small enough to avoid being a burden to send, and its constant size and the atomic delivery of UDP combine to make the update protocol easy to implement at both ends. For example, to acknowledge some data, the application locally sets `tcpr.hard.ack` and sends the entire state to TCPR.

Using UDP makes it possible for TCPR to be situated on a middlebox physically distinct from the one on which the application is running, but other options are also possible. In my experiments, the highest efficiency was achieved when running TCPR in a separate network namespace but on the same machine as the recoverable application. Network namespaces are a recent Linux kernel feature that enables an individual process to have an isolated routing table, network stack, and set of network interfaces, while sharing the host's memory, filesystem, processors, and kernel. With the application in its own network namespace, TCPR can run on the host as a middlebox while enjoying loopback-interface throughput and latency.

The TCPR distribution includes a `netcat`-like program and a TCP proxy that provides TCPR-support for unmodified applications, along with a utility to craft UDP TCPR updates on the command line to query TCPR state.

In order to be able to measure recovery time, I implemented a utility that opens hundreds of connections in parallel, injects failure on each of them, and then times its recovery using Linux's high-resolution real-time clock. To measure throughput, we I have also modified the venerable `ttcp` to support TCPR; including error handling and new command-line options for configuring TCPR, all that it required was the addition of 28 lines.

Modifying an application to use TCPR does not require any changes to existing socket system calls. Instead, one simply adds code to interact with TCPR during connection setup and teardown, and when input is to be checkpointed. A trivial example is shown in figure 20.

```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(s, &addr, addrlen);
listen(s, backlog);
c = accept(s, &peeraddr, &peeraddrlen);
getsockname(c, &addr, &addrlen);

// request connection state
state.address = addr.sin_addr.s_addr;
state.peer_address = peeraddr.sin_addr.s_addr;
state.tcpr.hard.port = addr.sin_port;
state.tcpr.hard.peer.port = peeraddr.sin_port;
write(tcpr, &state, sizeof(state));

// receive TCPR's copy
read(tcpr, &state, sizeof(state));

bytes = read(c, readbuf, readbuflen);

// update delayed acknowledgment
state.tcpr.hard.ack =
   htonl(ntohl(state.tcpr.hard.ack) + bytes);
write(tcpr, &state, sizeof(state));

write(c, writebuf, writebuflen);

// fail
close(c);

// recover
write(tcpr, &state, sizeof(state));
c = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(c, &addr, addrlen);
connect(c, &peeraddr, peeraddrlen);

// close gracefully
state.tcpr.hard.done_reading = 1;
state.tcpr.hard.done_writing = 1;
write(tcpr, &state, sizeof(state));
close(c);
```

Figure 20: A simple C server that uses TCPR. Closing without notifying TCPR is a convenient way to inject failure. The four lines that recover the connection include both application and TCPR recovery, and could be executed after migrating to a new host.

	Mbps	% Raw
Unprotected	896.354 ± 0.331	100
TCPR	896.385 ± 0.280	100

Figure 21: TCP throughput from the application to the peer.

A fault-tolerant application that uses TCPR makes the usual socket calls. After calling `connect` or `accept`—that is, once the connection has been established—the application retrieves the connection's state from TCPR. As the application consumes its input, it updates ack locally, then updates TCPR. Similarly, when there is no more input, it sets $done\_reading$, and when it is finished writing output, it sets $done\_writing$.

If TCPR itself fails, the application need only send another update message. If the application fails, it need only `bind` and `connect` again to establish a new connection with the same endpoints. The recovery snippet in figure 20 handles both cases.

A practical problem on Linux is that the local TCP will not allow an application to bind to a port that is in use in any way, even if it is only bound by listening sockets. Thus, a server that is still listening for incoming connections on a well-known port can not recover its connections with the same port numbers. To work around this problem, TCPR can remap the source port to avoid collisions; the application just specifies a `tcpr.port` that is different from `tcpr.hard.port`.

4.4 Evaluation

I conducted microbenchmarks of TCPR using two commodity Linux machines, each with two cores, connected by a 1 Gbps Ethernet link. TCP was implemented by an unmodified Linux kernel network stack.

Overhead

I evaluated throughput overhead using a version of `ttcp` modified to support TCPR. First, I measured send goodput from the application to the peer both with unprotected

	Mbps	% Raw
Unprotected	851.206 ± 4.652	100
Unsafe TCPR	837.275 ± 5.355	98
TCPR	838.699 ± 1.934	98

Figure 22: TCP throughput from the peer to the application.

	Microseconds
Unprotected	318 ± 27
Unsafe TCPR	326 ± 16
TCPR	334 ± 24

Figure 23: Latency from the application to the peer.

	Microseconds
Unprotected	550 ± 23
Unsafe TCPR	547 ± 94
TCPR	594 ± 85

Figure 24: Latency from the peer to the application.

TCP and with TCPR, reporting the average and standard deviation of 10 runs in figure 21. TCPR does not cause any measurable overhead.

Next, I measured the opposite direction: receive goodput from the peer to the application. This is the flow that is subject to delayed acknowledgments, so in addition to the previous two cases, I measured ‘Unsafe TCPR’, in which delayed acknowledgments were disabled.

As shown in figure 22, while there is slight overhead on the incoming packets, there is no measurable impact on throughput from delayed acknowledgments. That is reasonable, because TCP keeps a window of packets in flight in order to mask latency.

The latency impact of delayed acknowledgments can be seen in TCP round-trip times from packet traces. I used traces of throughput experiments like those described above. For each acknowledgment visible to the sender that covered new data, I computed the elapsed time since it had sent that data, reporting the average and standard deviation over all such packets (about 50 in each trace) in figures 23 and 24.

	Microseconds
Application	39 ± 8
TCPR + Application	167 ± 25

Figure 25: Recovery time.

There is no significant difference in latency from the application to the peer. However, for input from the peer to the application, both setups of TCPR exhibit much higher variance than unprotected TCPR; delayed acknowledgments seem to add nearly 50 microseconds of latency, but the overhead is small and within the standard deviation.

Notice that since the recoverable application is responsible for the delays to its own acknowledgments, these numbers could be arbitrarily large. My experiments are thus something of a best-case scenario, because I measured an application that always acknowledges its input as soon as possible after the `recv` operation. The delay in a real deployment will depend on the needs of the application.

Recovery

The delay associated with application recovery also depends strongly on the needs of the application, and would generally include the latency to detect failure and the latency associated with launching a replacement. To isolate the costs specifically associated with TCPR, I microbenchmarked its ability to recover a connection, measuring the time from when recovery is initiated until the application is able to `send` and `recv` again.

I measured two cases, shown in figure 25. In the 'Application' case, TCPR retained its state and only the application failed and recovered, so that TCPR could establish a new connection immediately. In the 'TCPR + Application' case, they failed and recovered together, so that soft state had to be restored from the peer's packets during recovery. (If the application had saved the soft state as well, recovery would proceed exactly as in the 'Application' case.)

To set these numbers into context, consider the BGP example: in both cases, TCPRmediated connection recovery is faster than usual inter-arrival times of BGP updates even on a heavily-loaded core Internet router. Thus, if the BGP failure itself is handled quickly enough by the router, is is possible to completely mask the event from BGP peers.

4.5 Related work

Pei, Zhao, et al. (2004) have studied the disruption caused during BGP convergence, explaining how topology and configuration affect the result, and evaluated a number of proposed strategies for making convergence faster and less disruptive. There has also been work that, rather than making convergence itself faster, ensures that routers can rapidly fail over to temporary routes until convergence is complete (Bonaventure, Filsfils, and Francois 2007). Both branches of work are valuable when convergence is unavoidable, such as when the topology genuinely changes. However, since they do not eliminate the disruption, it is preferable to mask failure and recover transparently when possible.

Prior work on transparent TCP masking and recovery has taken the approach of replicating the entire TCP stack and everything above it, the application included. For example, HydraNet-FT (Shenoy, Satapati, and Bettati 2000), CoRAL (Aghdaie and Tamir 2001; Aghdaie and Tamir 2009), HotSwap (Burton-Krahn 2002), ST-TCP (Marwah, Mishra, and Fetzer 2003; Marwah, Mishra, and Fetzer 2005), AR-TCP (Shao, Jin, and Wu 2006), TRODS (Lloyd and Freedman 2011), and others (Luo and Yang 2001; Zhang, Abdelzaher, and Stankovic 2004) do so using primary-backup replication.

FATPETS (Paris, Valderruten, and Gulias 2005) is a network stack written in Erlang, designed specifically for failover and migration. It can operate in an 'active' mode, which essentially consists of primary–backup replication of the entire state including buffers, or a 'passive' mode, which protects input with delayed acknowledgments like TCPR.

In order to avoid changing the network stack, new software can be interposed on its interfaces with the application and the network, recording and possibly modifying its incoming and outgoing events. Failover-TCP (Koch et al. 2003) and FT-TCP (Alvisi et al. 2001; Zagorodnov et al. 2003; Zagorodnov et al. 2009) use such a technique. Similar to the TCPR approach, acknowledgments are delayed until the data is safely handled and sequence numbers in the packets to and from a restarted network stack have to be rewritten.

All of these approaches suffer from being replay-driven. Often they maintain multiple active replicas by duplicating input to each of them, and using only the primary replica's output. Failover between active replicas is straightforward and fast. However, to avoid running out of replicas, new ones must be brought up to speed; the replay-driven approach assumes that the application is deterministic, and warms up new replicas by replaying all of a connection's input, at either the packet or the socket level. For an HTTP request, that can be quite effective, but it quickly becomes impractical for connections that receive a lot of input, such as a BGP session.

Another approach is to use a proxy server between a server and its clients (Marwah, Mishra, and Fetzer 2006). The client sets up a connection to the proxy server, and the proxy server handles failing over from a primary server to a backup server as necessary. However, without an approach for replicating the proxy (Marwah, Mishra, and Fetzer 2008), it also introduces a new single point of failure.

Virtualization offers an even lower-level approach to transparent migration. For example, Remus (Cully et al. 2008) replicates an entire virtual machine, including the application, the network stack, and the operating system. Virtual machine migration can also be exploited to protect particular applications (Keller, Rexford, and van der Merwe 2010; Clark, Fraser, et al. 2005). Using such technology, however, the failure or reconfiguration of the application itself within the virtual machine image still causes its TCP connections to break, thus providing only limited benefit for the heavyweight replication demands.

Backdoors (Sultan, Bohra, et al. 2005) takes a unique approach based on replicating state after the application has already failed, by assuming that the network interfaces survive the failure and modifying them to support remote DMA.

Finally, when it is possible to modify all of the application's peers, a good option is to introduce a session library or modified socket library (Huang and Kintala 1993; Orgiyan and Fetzer 2001; Snoeren, Andersen, and Balakrishnan 2001; Sultan, Srinivasan, et al. 2002; Zandy and Miller 2002). Such an implementation could automatically set up a new connection to a new server in case the connection to the current one fails, or even maintain redundant connections.

4.6 Conclusions

Application-driven TCP recovery is a novel approach that enables a fault-tolerant application to protect connections in cases that were impossible with replay-driven recovery. The current prototype, TCPR, confirms the simplicity of application-driven recovery. Rather than replicating the TCP state and everything above it, TCPR is not even replicated itself, because the application assumes responsibility. TCPR is middleware, outside the local TCP, enabling any application to use it with an unmodified, unwrapped TCP and whatever interface it provides. By controlling the acknowledgment of data, along with occasional input from the application layer, TCPR needs only a small, constant amount of state per connection, enabling low overhead in normal operation and sub-millisecond recovery.

The ability to easily migrate connections between replicas can also be useful when running multiple versions or configurations of an application. For example, a router could be made tolerant to implementation bugs by exploiting software diversity (Keller, Yu, et al. 2009). Using TCPR to protect the output of a master version or voting module can enable such an approach to tolerate hardware failure as well.

TCPR can offer benefit to applications other than BGP. For example, TCPR enables a lightweight approach to migration that can also tolerate unexpected failures, and can therefore be beneficial for load balancing long-running connections for streaming media within a CDN point-of-presence.

High-availability, fault-tolerant applications already do a lot of work to maintain their own state. Application-driven TCP recovery can be seen as merely a way to give such an application access to its own state, which would otherwise be encapsulated in a local TCP, without interfering with the implementation hidden behind that encapsulation.

CHAPTER 5

THE TRANSMISSION CONTROL PROTOCOL

In the Internet suite, the Transmission Control Protocol (TCP) adapts the best-effort, packet-oriented channels offered by the Internet Protocol (IP) into reliable, ordered, octet-stream channels (RFC 793). This chapter describes the structure of a formal specification, using a novel decomposition based on channel markets. The specification itself is given in TLA+, and appears in appendix A. Details of how the specification uses TLA+ appear in this chapter, but for a good introduction to TLA+ itself and to specification in general, read the book *Specifying Systems* (Lamport 2002).

5.1 Overview

A TCP connection is between two peers. The connection is initially closed, and when both peers open it, they establish a new incarnation of the connection, which consists of two streams of octets—each peer is the sender for one stream and the receiver for the other. The two streams open together, but each sender can close its stream independently. After both streams are closed, the connection itself becomes closed and can be opened again.

To mask duplication and reordering in the network subchannels, a sender sequentially numbers each octet in its stream, and a receiver delivers each octet to its local application exactly once and in order per sequence number. To mask loss, a sender retransmits each octet in its stream until its sequence number is acknowledged, and a receiver acknowledges a prefix of the sequence for which it guarantees delivery.

TCP does not mask peer failures, but it does recover from them. Although it is not implemented, each stream of each incarnation of a connection conceptually bears a unique identifier. When a peer loses its state and attempts to open a connection that has already been established, the other peer rejects the unexpected identifier and replies with the identifier it did expect. The recovering peer resets the connection, leading both peers to agree that the connection is closed, so they can reopen it as normal. Uniquely identifying the streams also provides some security, in that the peers will reject data injected by an adversary unless the adversary can guess or sniff the identifiers of the streams of the current incarnation.

TCP makes several concessions to the bounded resources of real computers and networks. One of the most notable is that, although there is no bound on the number of incarnations that may be opened of a connection, and no bound on the length of the streams in an incarnation, TCP headers are a fixed size. Ideally, every octet ever sent in a connection would be uniquely identified by the identifier of the stream (call it id) and the octet's sequence number within that stream (call it n). However, in transmission, the pair (id, n) is shoehorned into a single 32-bit field, simply called the sequence number. The sequence number in the header is computed as $id + n \pmod{2^{32}}$.

Even though 2^{32} octets is finite, TCP does not assume that a receiver can buffer that much at once. Instead, the receiver tells the sender what window of sequence numbers it is prepared for, and the stream proceeds through the 32-bit sequence number space as the receiver updates its window.

Because of such concessions, TCP cannot make the same guarantees that would be possible with distinct, unbounded stream identifiers and sequence numbers. Because the two concerns of distinguishing incarnations of a stream and distinguishing octets within a stream are combined in one field, it is possible for different incarnations of a connection to conflict, and the window of acceptable octets also presents a window of acceptable streams for a spoofing adversary. Because the field is bounded, it is possible for a stream that sends data too quickly to wrap around and conflict with itself. TCP overcomes those limitations with timing assumptions, such as assuming a bound on how long a segment can be in the network before it is dropped, and requiring a peer to wait in some cases until the network is empty of relevant segments.

Scope of the specification

I will present a specification of one incarnation of a connection. Thus, the specification includes the proper negotiation of opening and closing the incarnation, but not the exceptional conditions that can result from receiving segments from old incarnations of the connection. In the unbounded case, a simple agent can detect and address such cases outside the specification, and in the modular arithmetic case, TCP's timing assumptions exist to ensure that such cases are not exercised. In the specification, peers do not fail; in chapter 6 I will add the possibility of peer failure and use application-driven recovery to mask it.

The specification uses unbounded sequence numbers, rather than modular arithmetic. The two are not equivalent, but with appropriate additional timing assumptions, the bounded case can be shown to achieve the same property as the unbounded case (Smith 1997).

The specification does not include aspects of TCP specified outside of the core TCP standard (RFC 793), which notably excludes window scaling and congestion control. Congestion control is one of TCP's most important features in practice, but it is not related to the safety or liveness of TCP. Rather, congestion control is important for the use of network resources, which the specification does not model.

By focusing on a single incarnation of a single connection, the specification also excludes some aspects of the standard, including checksums and port numbers. The specification also abstracts away the Maximum Segment Size option, for simplicity; the security mechanism, which RFC 4614 notes is 'no longer implemented or used'; and the precedence mechanism, which RFC 2873 removed.

The specification includes aspects of TCP that are often ignored in formalizations, specifically, the urgent and push protocols. It is not surprising that the urgent protocol

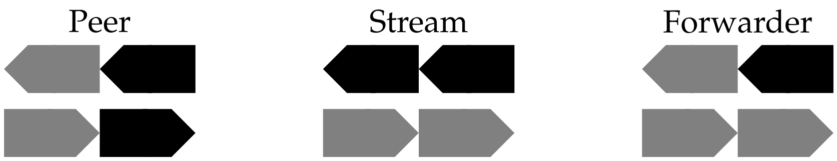

Figure 26: Three decompositions of TCP. From an implementation perspective, the peer-based decomposition makes the most sense, and it is adopted by the standard. From a verification perspective, the stream-based decomposition makes the most sense, and it is often adopted by formal specifications. Either decomposition can easily be recovered from a forwarder-based decomposition.

is often ignored, since it is rarely used, but I want to be able to show in chapter 6 that TCPR does not violate the safety of the mechanism. It is surprising that the push protocol is often ignored, because it is crucial to the only liveness property mentioned in the TCP standard. This specification covers both.

5.2 Taking TCP apart

Recall that a TCP connection consists of two streams between two peers, with each peer the sender for one stream and the receiver for the other. That description suggests two obvious decompositions: by peer and by stream. The TCP standard (RFC 793) takes an implementation-driven approach and describes the transitions of a peer, where a connection is composed of two such peers. Smith (1997) takes a verification-driven approach and describes the transitions of a stream, where a connection is the composition of two such streams.

I will adopt a third decomposition, starting from functionality shared by both the sender and receiver aspects of a peer: forwarding. A sender forwards from its local application to the network, and a receiver forwards from the network to its local application. A connection is the composition of four such forwarders. The three approaches to decomposing TCP are shown in figure 26.

A stream consists of three subchannels that convey information chosen by the application—

the data channel, urgent channel, and the push channel—and two subchannels that convey feedback from the remote peer—the acknowledgment channel and the window channel. Each subchannel's symbols have a natural ordering, and the function of a forwarder is to relay non-decreasing updates. I will begin the specification by describing, for each subchannel, its symbols, its ordering, and the role it plays in TCP.

THE DATA CHANNEL

The data channel communicates the data produced by an application. The payload of the data is the sequence of octets that represent the application protocol. The octets are delimited by two control flags: A SYN marks the beginning of the stream, and a FIN marks the end. Every octet and control is numbered sequentially from zero.

Some definitions for the data channel appear in the *Data* module in appendix A. An element of *SegmentData* is a contiguous subsequence, such as might be carried in a segment on the wire.

The symbols for the data channel, elements of *Data*, represent prefixes of a complete sequence. Thus, they include the empty set (the initial value before the connection opens) and all subsequences that begin with SYN. They are ordered by set inclusion, so the data grows by appending new octets and controls. Once the data includes FIN, it is not a subset of any other valid data; thus, the sequence cannot grow after it is closed.

THE URGENT CHANNEL

TCP allows the application sending a sequence to notify the remote application that some prefix of the data is urgent. The remote application is presumed to use that information somehow, but 'urgent' means nothing to TCP itself.

(This is the protocol related to the out-of-band mechanism, and the `MSG_OOB` flag, in the sockets interface. Note that the data is still in-band. It is only the notification that is out-of-band.)

The urgent channel communicates the length of the prefix of the sequence that is urgent. As a length, its symbols are just the natural numbers, initially zero and increasing according to the usual order.

The push channel

The push mechanism is irrelevant to the safety of TCP; apart from liveness, there would be no need for it. A push communicates the length of the prefix of the sequence that must be forwarded. A forwarder may, but need not necessarily, forward parts of the sequence that have not been pushed.

Unlike the urgent channel, which only communicates the length of the longest urgent prefix, the push channel communicates the lengths of all of the pushed prefixes. That is necessary because the liveness requirements a push implies for a forwarder do not depend on network fairness.

Normally, one push with a length of 9 has the same meaning as two pushes, one with a length of 5 and one with a length of 9. In either case, eventually the first nine octets of the sequence must be delivered to the remote application.

However, the two cases are different if the network stops delivering any segments after the first five octets have been delivered. With only the single push of length 9, the push itself would not yet have been forwarded, and the receiver would not be obligated to forward the data to the remote application. In the latter case, the push of length 5 would have been forwarded, and the receiver would forward the data.

Thus, the symbols of the push channel are subsets of the natural numbers, initially empty and growing according to set inclusion.

The acknowledgment channel

The acknowledgment channel is feedback from the receiver of the sequence, communicating the length of the prefix of the sequence that it guarantees to deliver to the remote application. As a length, the channel's symbols are just the natural numbers, initially zero and increasing according to the usual order.

Note that the local receiver's acknowledgment, which guarantees to the remote sender how much of the sequence will be delivered to the local application, is the channel affected by TCPR's checkpointed acknowledgments. Normally, acknowledgment channels are not visible to either application.

The window channel

The window channel is feedback from the receiver of the sequence, communicating the length of the longest sequence that it is able to buffer. Whether or not the application has sent data past the window, the sender should only forward data that fits in the window.

The sender is always allowed to forward at least one octet of data, even when the window does not include any space for new data. This exception is necessary for the window to reopen reliably, since data is retransmitted upon loss but window updates are not. Furthermore, the standard requires a receiver to accept a segment as long as any part of it fits in the window, by ignoring the part that does not fit; thus, in a sense, windows are not important for safety, but, like congestion control, they are important for resource usage.

Unlike acknowledgments, windows are not ordered by length—a receiver is allowed, although discouraged, to decrease the window. Rather, window updates should be accepted in the order in which they are sent. However, due to network reordering, it is not always possible for a sender to know the correct order of the feedback it gets, so window updates (defined in the $\mathit{WindowUpdate}$ module) include not only the window length, but the sequence number and acknowledgment of the segment that carried the update. The order in which the updates were sent is approximated by comparing sequence numbers and using acknowledgments to break ties. When two updates are equivalent under that ordering, either length is acceptable. The standard does not specify an initial window

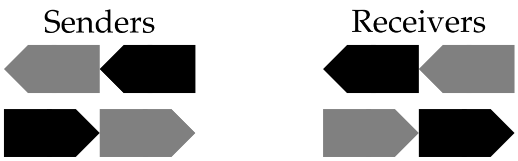

Figure 27: A sender forwards from its local application to the network. A receiver forwards from the network to its local application.

during connection establishment.

5.3 Putting TCP back together

A TCP peer is the sender for one stream and the receiver for another stream; thus, it is composed of two kinds of forwarders. A sender forwards stream, urgent, and push data from the local application, while its acknowledgment and window are based on feedback from the other endpoint through the network. Conversely, a receiver forwards stream, urgent, and push data from the network, while its acknowledgment and window are chosen locally. Both kinds of forwarder are shown in figure 27.

The channels

What kinds of channels enable these agents to communicate? The concept of 'forwarding' implies that an agent observes a channel it regards as input, then manipulates accordingly another channel it regards as output. In TLA+, the simplest channel is a variable. It has a value in every state, which is undesirable in this context because it is impossible to have a transition in which a value is not sent, and because it is impossible to distinguish sending the same message twice and a stuttering transition in which nothing happens.

A higher-level channel with message-passing semantics is defined in the $MsgChannel$ module. The predicate $Send(m)$ holds exactly when a transition takes place that corresponds to sending the message m. Sending a message twice is well-defined, as is not sending any message. No message is sent in a stuttering transition.

The application interface

The standard does not require any particular interface between TCP and an application. This specification adopts a simple interface, in which the application's entire state, as represented in the *ApplicationUpdate* module, is transmitted across a single *MsgChannel*.

As with TCP, an application has both a sending and receiving role. The *ApplicationSender* module specifies the source of a stream. The *Hidden* inner module is given in terms of the *dat*, *psh*, and *urg* variables, representing the data, push, and urgent channels, respectively, and the *chan* channel representing the interface with TCP. The inner module is then used in a TLA+ idiom to hide the internal variables, ultimately specifying the application sender only by the behavior it exhibits on the interface with TCP.

The *ApplicationReceiver* module specifies the other end of the stream, also in terms of the interface with TCP. The receiver just records the data, push, and urgent values sent through the protocol.

The senders and receivers

A sender, specified in the *Sender* module, receives application updates in one channel *chan*, forwards segmented pieces of it through the *seq* (sequence number), *dat* (data), *psh* (push), and *urg* (urgent) channels, and receives feedback through the *ack* and *wnd* channels.

The *Update*, *Send*, and *Receive* transitions correspond to receiving an update from the application, forwarding to the network, and receiving feedback, respectively. There is an additional *Liveness* condition that describes the push mechanism.

A receiver, specified in the *Receiver* module, is similar except it forwards in the other direction; for example, the *Update* transition corresponds to forwarding the received data to the application.

5.4 Modeling the network

A network can also be seen as essentially a forwarder. The specification appears in the $Network$ module, and describes an agent that forwards messages from a snd channel to a rcv channel, although it might drop, duplicate, or reorder the messages in transit. The uncertainty is modeled using a hidden net variable containing the segments currently in transit; a segment enters the network in a $Send$ transition, and might be $Deliver$ed zero or more times before it is finally $Drop$ped.

TCP assumes that a network is fair, that is, that a message TCP retransmits infinitely often will be delivered infinitely often (Gouda and Chang 1986). However, the messages TCP delivers reliably are data octets, although it sends them bundled up into segments at the network interface. Because TCP can divide data into segments arbitrarily, it is not obvious that fairness for segments implies fairness for octets. Fortunately, there are only a finite number of segments that can carry each octet—note, for example, that the complete sequence of data in a stream is finite. Therefore, if TCP retransmits an octet infinitely often, there exists some segment that carries that octet and is transmitted infinitely often. It is sufficient for a network to deliver segments fairly, without regard for their contents.

The fairness condition is formally stated as $Fairness$, which requires strong fairness (SF) of the $Deliver$ transition for every possible segment. Strong fairness states that if the transition is enabled infinitely often, it takes place infinitely often. Thus, even if the segment is dropped (disabling the transition), if it is sent again (re-enabling it) and again until it is delivered, the delivery must occur eventually.

The structure of a segment is specified in the $Segment$ module. A segment can comprise updates for each of the five subchannels I described for a forwarder, although only the data update is required.

The data in a segment seg is a subsequence of the entire stream's data, starting from offset $seg.seq$ and conveying the octets and controls in $seg.dat$.

The segment conveys a push when $seg.ctl.psh$ is set, in which case the push covers the stream up through the end of the segment's data. The segment conveys urgent information when $seg.ctl.urg$ is set, in which case the length of the urgent prefix is given by $seg.urg + seg.seq$.

When $seg.ctl.ack$ is set, the segment also conveys feedback for the other stream. The acknowledgment is carried in $seg.ack$ and the window length is given by $seg.wnd + seg.ack$.

The way a segment is formed from each of the five subchannels is specified in the *SegmentMux* module. The specification is notable because it is written as a forwarder between a channel that carries segments and five channels carrying the absolute values of the components, and because it is reversible. The specification of a forwarder that combines five inputs into an output segment is the same as the specification of a forwarder that unwraps an input segment into five outputs.

5.5 THE CONNECTION SPECIFICATION

The *Peer* module wraps up a sender, a receiver, and two segment multiplexers to produce an agent that forwards back and forth between the network interface and the application interface. The *Endpoint* module further composes a peer with its local application, resulting in a very simple specification (hiding a lot of complexity) about the behavior of TCP in terms of its input and output network subchannels.

A connection is two communicating peers. The complete specification appears in the *Connection* module, which uses temporal implication to state that as long as the networks behave according to the *Network* specification, the TCP endpoints will behave according to the *Endpoint* specification. Because the temporal implication operator describes what TCP offers (the endpoint specification) given two subchannels conforming to specific assumptions (the network specification), it corresponds to the shopping list view of the channel market.

5.6 Related work

Subsets of TCP have been specified in a variety of formalisms. When reading such a specification, I find it most interesting to note not just what formalism was chosen for the task—and they range from higher-order logic to I/O automata to Petri nets—but also what details of TCP have been ignored. Abstraction is valuable precisely because it leaves out the irrelevant details of a thing, and one of the best ways to see the differences between various authors' goals is to notice what they consider irrelevant.

Guttman and Johnson (1994) summarized using Communicating Sequential Processes (CSP) to give a high-level specification of TCP, focusing on the lessons they learned from the attempt. They discovered errors and unnecessary complexity in the TCP standard, and verified the high-level safety property that 'the sequence of octets of data received by one entity is an initial part of the sequence of octets sent by the other entity'. They described CSP as elegant at a high-level, but mentioned trying and failing to use it for a detailed description of an endpoint. They concluded that a state-machine-based formalization would be more appropriate.

Smith (1997) used I/O automata to specify TCP, first with unbounded sequence numbers and then identifying timing assumptions necessary to introduce bounded sequence numbers. Smith then proved that the Transactional TCP (T/TCP) protocol (Braden 1994) does not refine TCP even under unbounded counters, and gave a formal specification of the properties it does provide. Smith and Ramakrishnan (2002) used I/O automata again to verify the safety of the TCP SACK mechanism (Mathis et al. 1996) and explore some of its performance implications. In both the T/TCP and SACK projects, the specifications were limited to a single stream with a fixed server and client.

Vigna (2003) presented an *ad hoc* graphical model of a network in order to explore sniffing and spoofing attacks on UDP and TCP. Because the focus was on using the internal topology of the network in the attack, the TCP protocol was extremely simplified, leaving out, for example, windows and retransmissions.

Han (2004) used Colored Petri Nets for the purpose of verifying the connection establishment mechanisms of TCP. Han broke TCP down into low-level services such as opening and handling aborts, and was able to observe that, for example, including an abort service increased the complexity of the state space much more than including a simultaneous open service. The focus on connection establishment does not include closing the connection, and the specification explicitly leaves out features such as urgent data and push, which are only relevant to data transfer.

Zaghal and Khan (2005) used the Specification and Description Language (SDL) to model TCP and the TCP Reno congestion control algorithm. The model of TCP follows RFC 793 very closely.

The Network Semantics project (Bishop et al. 2005a; Bishop et al. 2005b; Ridge, Norrish, and Sewell 2009) takes a completely different approach from all of the previous, because it is not a formalization of the TCP standard. Rather, it is a formalization of the sockets interface, including TCP, as it is actually implemented. The authors note that, despite a very different and hopefully more readable structure, their higher-order logic (HOL) specification is about the same textual length as an actual TCP implementation. Furthermore, although the specification itself is formal, due to its complexity is has not been formally proven equivalent to anything—rather, it was validated using numerous packet traces, emphasizing errors and corner conditions. The result is a novel combination of formal and empirical methods, and the best starting point for a study of how something would interact with TCP in a real network.

CHAPTER 6

RECOVERING IS AS GOOD AS NOT FAILING

It is possible to recover connections using middleware, without modifying the original TCP implementation. Here is a simple proof: if the middleware itself implements a complete TCP peer, with an appropriate interface enabling the application to save and restore its state, then the system need not depend on the original TCP at all.

It does not prove much that such an approach is sufficient. The challenge is to use simpler middleware that leverages the fallible original TCP as much as possible. This chapter describes the structure of a formal specification of such middleware, building on the TCP connection specification from chapter 5 and appendix A. The recovering connection specification, in TLA+, appears in appendix B. This chapter concludes with a refinement proof, which shows that recovering is as good as not failing.

6.1 The fallible peer

The most obvious difference between this specification and the previous one is failure. The $\mathit{Replicas}$ module provides a fail-stop abstraction wrapped around some other channel. The f variable records the number of failures that have ever occurred, while the r variables contains an infinite set of replicas indexed by the natural numbers. When a failure occurs, the current replica halts and a new replica becomes current; a failure is observable through non-stuttering transitions of f. Between failures, the current replica can transition according to some specification, outside the scope of the $\mathit{Replicas}$ module. The $\mathit{Replicas}$ module does specify that no other replica (failed or not yet used) has any non-stuttering transitions at all.

Building from this simple abstraction, the $\mathit{RecPeers}$ module specifies a fallible TCP peer. There are indexed sets of replicas of *Peer*s, as defined by the original TCP specification, and the usual channels for communicating with each peer.

6.2 The fault-tolerant application

The application must change in order to use the recovery middleware; this specification captures those changes. One way in which it does not change is by failing, because one assumption is that the application is fault-tolerant. If the application can fail and recover, it is not modeled in this specification.

The *RecApplicationSender* module specifies the part of the application corresponding to the original *ApplicationSender*. The changes are additions; notably, the *dat*, *psh*, and *urg* variables have the same values and transition through the same *Internal* transition in either specification.

The recovery version has the additional f variable for detecting failure and the indexed set of peer replicas. Additionally, whenever the peer fails, the application receives (through the *recoff* channel) the offset within the data where it should start with the new peer. The corresponding change in the *Update* transition edits the data, push, and urgent updates to be consistent with the current peer's view that the connection only started at the given offset.

The application's receiving role is specified in the *RecApplicationReceiver* module, corresponding to the original *ApplicationReceiver* module. When it receives new data from its current peer, it interprets the update through an offset, collecting the entire stream. It also sends the length of the prefix of the data that has actually arrived as a checkpointed acknowledgment (through the *recack* channel). During failure, the receiver can compute its new offset directly from its checkpointed acknowledgment.

6.3 The middleware

The middleware, specified in the *RecMiddleware* module, communicates with the application through the *recoff* and *recack* channels, as I just described from the application's perspective. It also performs its main function by translating between segment channels.

The external channels, *extsnd* and *extrcv*, correspond to the input and output networks of an original TCP peer. They communicate with the environment. The internal channels, *intsnd* and *intrcv*, are replicated and connect to each of the replicated TCP peers associated with f.

For example, when the current peer follows its specification to send a segment, rather than going to an externally visible network, the segment appears in the middleware's current *intsnd* channel. The middleware might modify the segment with the appropriate offset and checkpointed acknowledgment (the *SndSeg* transition) and forward it to the *extsnd* channel. When it detects that recovery is necessary (the *Recovery* transition), it instead drops that segment and replies back through the *intrcv* channel for the current peer. The *RcvSeg* transition specifies forwarding a segment from the network, *extrcv*, to the current peer's *intrcv*.

6.4 The recovering connection specification

The *RecEndpoint* and *RecConnection* modules serve the same roles as the *Endpoint* and *Connection* modules served in the original TCP specification. A *RecEndpoint* wraps up a fault-tolerant application, its TCP replicas, and the middleware, hiding everything but the input and output networks. At that point, a *RecEndpoint* has the same interface as an original *Endpoint*.

A *RecConnection* is a complete TCP connection, with one legacy *Endpoint* and one recoverable *RecEndpoint*. The *RecConnection* module also formally states the refinement theorem, that a connection with a recovering endpoint is as good as a connection without failure.

6.5 The refinement

To prove the refinement, it is first necessary to state a refinement mapping, a function from the states of the recovering connection specification to the states of the original connection specification. For such a function to be a refinement mapping, the images of initial states must be initial states of the original specification, and for every transition in the recovering specification, there must exist a sequence of transitions in the original specification between the images of the states. Good descriptions of the refinement mapping technique have been given by Lynch and Vaandrager (1995) and Abadi and Lamport (1991).

The refinement mapping

In this refinement, the legacy $\mathit{Endpoint}$ and the two $\mathit{Network}$s (a and b) play themselves. Note that the composition in the $\mathit{RecConnection}$ module does not give the two endpoints any channels in common except for the networks, and each endpoint synchronizes on opposite ends of each network (one transitions with Send, the other with $\mathit{Deliver}$), so there are no transitions in which both endpoints simultaneously change. Similarly, the network Drop transitions occur independently of both endpoints. Thus, it suffices to show that the $\mathit{RecEndpoint}$ refines the original $\mathit{Endpoint}$ specification.

An $\mathit{Endpoint}$ is the composition of an $\mathit{ApplicationSender}$, an $\mathit{ApplicationReceiver}$, and a Peer. The application roles are played by the corresponding $\mathit{RecApplicationSender}$ and $\mathit{RecApplicationReceiver}$, respectively. The original and recovering application specifications share the dat, psh, and urg channels, and there the refinement is direct.

The Peer is played by the middleware and replicated internal peers. Within the simulated peer, the Sender and $\mathit{Receiver}$ are simulated as follows: The maxdat, maxpsh, and maxurg channels, which store the information from the application, comprise the merged values from all of the replicated peers, with the appropriate offsets (as recorded in the middleware) applied. That is, $maxdat$ is the $Merge$ of all of the offset $maxdat$s from all of the replicated peers; $maxurg$ is the maximum of all of the offset $maxurg$s from all of the replicated peers, *et cetera*. For the $Sender$, the feedback channels $maxack$ and $maxwnd$ are also simulated by merging the values from the replicas. For the $Receiver$, $maxack$ is taken from the middleware's $rcvack$ (the checkpointed acknowledgment), and $maxwnd$ is taken from just the current peer replica's receiver.

A NOTE ON LIVENESS

This refinement does not preserve the liveness requirements associated with the recovering endpoint's receiver's push. It is possible that a replica receives data and a push for the data, but fails before forwarding it to the application. Thus neither the data nor the push will be in the current replica, but it will be in the simulated receiver due to the merge. There are two ways I could make the refinement more sophisticated to handle liveness.

The first way is to add a fairness assumption. Specifically, if the network is fair, any data or push in the merged values of all the replicas but missing from the current peer will be retransmitted by the remote endpoint, due to the checkpointed acknowledgment. Thus, the network fairness assumption is sufficient to guarantee the receiver's liveness with respect to push, because the current receiver will catch up to the simulation.

The second way is to leverage the network. Rather than letting it play itself, the simulated network would never drop any segments, thereby serving the role of a history variable recording all of the segments ever sent. And, rather than naïvely merging all of the replicas, the merged result could be trimmed not to exceed the checkpointed acknowledgment. The simulated peer would not deliver segments until the actual peer had advanced its checkpointed acknowledgment to cover those segments. Parts of segments could be handled by manipulating the simulated window to trim them.

I have chosen to describe both possibilities here, but to focus on safety for the purpose of this simulation.

INITIAL STATES

The initial states are easy to compare, because in both cases, all of the streams are empty (no data, no urgent prefix, no pushes, no acknowledgments, arbitrary windows) and all of the channels are empty (no segments in the network, no messages sent between application and TCP).

APPLICATION SENDER INTERNAL TRANSITION

The $\mathit{RecApplicationSender}$'s $\mathit{Internal}$ transition is identical to the $\mathit{Internal}$ transition in the simulated $\mathit{ApplicationSender}$.

SENDER UPDATE TRANSITION

The transition in which the application's sending role updates its local TCP sender (the Update transition in $\mathit{RecApplicationSender}$ and Sender) consists of sending the application state, offset appropriately for the point in the stream where the current peer became current. Because all of the data before that offset must have already been in a previous replica, the peer simulated by merging the replicas transitions appropriately as though the entire application state were sent. Thus, the update transition is simulated by the original Update transition of a non-replicated Sender and original $\mathit{ApplicationSender}$.

RECEIVER UPDATE TRANSITION

Nearly the same argument applies to the receiver update transition as for the sender update transition—the merge in the refinement mapping inverts the offset made necessary by the replication. The primary difference between sender and receiver update steps is that the receiver also passes on the new checkpointed acknowledgment to the middleware. This corresponds to an $\mathit{Internal}$ transition of the simulated peer's $\mathit{Receiver}$. Thus, the simulation of a receiver update is the original update, followed by an internal transition to set the acknowledgment.

Receiver internal transition

Because the simulated receiver acknowledgment is taken from the middleware, an actual internal transition of the current receiver that changes the acknowledgment is a stutter. When the internal transition also changes the window, that does affect the simulated receiver—in the same way as the same internal transition, but with no change to the acknowledgment.

Send transitions

The Send transition of the current peer's Sender does not change any of the state in the peer relevant to the refinement mapping, and it also does not impact the network, because the segment is only sent to the internal network between peer and middleware. Thus, the simulation of the internal send transition is a stutter.

When the middleware forwards the segment with a SndSeg transition, the simulated network must change accordingly. The segment produced corresponds to a Send transition of the simulated Sender; in fact, it can be seen as the *raison d'être* of the middleware to map segments from internal to external so that they could have been produced by the simulated, un-failed peer.

Recovery transition

When the middleware instead handles a segment on the internal network with a $\mathit{Recover}$ transition, it is because the current peer has not yet established the connection. The recovery step is a stutter, because it only influences the internal networks which are not part of the mapping, but it produces a segment that can later trigger an internal receive. That internal receive will also be a stutter, because it is simply the current peer establishing the connection—it will not receive any data, urgent, or push updates, so it will not change the merged peer being simulated.

Receive transitions

As with an internal send, an external receive is a stutter because although the middleware updates some of its state and forwards the segment to the internal network, none of the state relevant to the mapping is changed. An internal receive might change the current peer, and thus might change the merged values in the simulated peer, thus corresponding to the relevant *Receive* transition.

Failure transition

Failure impacts every module of the recovering endpoint—note the *Fail* transitions of the *RecApplicationSender*, *RecApplicationReceiver*, and *RecMiddleware* modules, and implied by the *Replicas* instances in *RecPeers* and throughout. However, although failure results in the computation and communication of the offsets that will be necessary to use a new peer, none of the module's failure transitions changes the data, push, urgent, acknowledgment, or windows of any of the middleware, application, or replicated peers. Thus, it is a stuttering step in the simulated endpoint.

That failure is simulated by stuttering should be seen as an important feature. It guarantees that failure is truly masked.

CHAPTER 7

DESIGN PRINCIPLES

The channel market model is not only a tool for reasoning about existing systems, it is also a tool for guiding the design of novel systems. A model influences what systems get built, because some systems are easier to explain with it than others. One way a model can be beneficial is by making the best systems the easiest to explain. Because people want those benefits to be reusable, when they notice a pattern, they often state it in a pithy way and call it a design principle. In a sense, a design principle is merely an aspect of a model—people have found a way to name and discuss one facet of the influence from thinking in a particular way.

7.1 MODULARIZATION

The most fundamental choices in design concern decomposing a system into parts. One might discover the decomposition top-down, by successive refinement of the system's goal until it is a practical implementation, or bottom-up, starting from achievable components and seeking a composition that adds up to a system that meets the goal. Usually, real designs arise from a mixture of the two. In any case, breaking the whole system into parts that are easily comprehended by a person is essential, if people are to build the system and justify its dependability.

A model often provides guidance by providing the structure of the components. Such guidance often takes the form of 'everything is an x', where x might be file, object, function, array, list, and so on. In the network stack model, everything is a network; in the channel market model, everything is a channel. Such broad statements are usually oversimplifications, but at least the designer knows how to recognize a module. The problem remains, however, to decompose a particular system into the right ones—where should the boundaries between modules be drawn?

The problem was solved independently by Parnas (1972), who introduced the principle of information hiding, and Dijkstra (1982), who introduced the separation of concerns. Both are derived from the observation that not only must the modules themselves be manageable, but the structure of their composition into the whole system must be as well.

Parnas (1972) explained information hiding by developing an example program in two ways, first based on control flow, and second based on design decisions. The two designs are about as easy to write, but the first is vastly inferior to the second when design decisions change. Encapsulating them within modules is a form of design-time fault-tolerance.

Dijkstra (1982) explained the separation of concerns as a basis for science, and compared it to the division of science into fields and specialties: 'When the layman asks the computing scientist, what is meant by 'Modularization', a reference to the way in which the knowledge in the world has been arranged, is probably the best concise answer.' He described two requirements for such an arrangement to be successful. Internally, a module requires coherence, that is, useful work must be possible with the parts within a module. Externally, a module requires a 'thin interface', that is, the least possible dependence on parts of other modules.

Most other design principles can be seen as additional requirements that might help one judge whether a module has been drawn at the right boundary.

7.2 The Internet model

The design principles that follow from the Internet model are well expressed in the end-to-end principle, the simplicity principle, the robustness principle, and the tussle principle. Of them, the most influential is the end-to-end principle.

The end-to-end principle was first expressed by Saltzer, Reed, and Clark (1984) using the following argument: If the implementation of some particular functionality, such as reliable delivery of a file, requires knowledge only available at the endpoints of the communication, then its implementation at the intermediaries would be redundant and should be avoided. In some cases, the redundant implementation is justified because it yields significant performance benefits, but the end-to-end argument pushes a design toward smart endpoints connected by a dumb network.

The end-to-end argument soon evolved into a more general principle, because of two main benefits of such designs even when the functionality in question does not depend on the endpoints for correctness (Kempf and Austein 2004). First, it eases innovation, because new end-to-end functionality requires only new endpoints, not a new network. Second, it improves survivability, because if state is placed at intermediaries only for performance enhancement, then the loss of that state can at worst degrade performance. Thus, such state is called 'soft state' (Clark 1988).

The hard state necessary for correctness, on the other hand, can be lost if and only if one of the communicating endpoints crashes. If one further assumes that the communication is meaningless without the endpoint, then the loss of that state is no worse than the crash itself. In such a case of 'fate-sharing' (Clark 1988), there is no benefit from replicating the hard state in multiple locations, and, in the words of William of Ockham, *pluralitas non est ponenda sine necessitate* ('plurality should not be posited without necessity').

Such a sentiment is explicitly supported by the simplicity principle: that complexity is the primary cause of inefficient scaling (Bush and Meyer 2002). RFC 3439 quotes Doyle: 'Complexity in most systems is driven by the need for robustness to uncertainty in their environments and component parts far more than by basic functionality.' Ironically, 'complexity added for robustness also adds new fragilities, which in turn leads to new and thus spiraling complexities.' (Bush and Meyer 2002) The simplicity principle is directly related not only to Ockham's razor, but to other popular exhortations for simplicity, including the KISS principle and the principle of least astonishment.

Another principle, almost as famous as the end-to-end principle, is the robustness principle: 'be conservative in what you do, be liberal in what you accept from others.' (RFC 793; Braden 1989). Whereas the end-to-end principle pushes toward designs that support innovation and survivability, the robustness principle pushes to support an even more important goal: connectivity (Clark 1988). Even in the face of buggy, malicious, non-compliant, or misconfigured network entities, the robustness principle requires one to proceed as well as possible. (A similarly pithy expression of that priority is the goal of 'rough consensus and running code'.) Robustness in that sense is not always desirable, because it makes it much more difficult to trace down what caused a particular strange behavior, harming survivability, and, by allowing non-compliant implementations to become legacy, harms long-term innovation. Thus it is a clear example of the way the Internet model has prioritized its goals, which, although it does not match every user's needs, contributed to the Internet's success.

That success led the Internet to a very different environment than the one in which it was created. No longer do all of the participants even agree that connectivity is the highest priority—for example, companies expect to be paid. Governments expect accountability, while users expect performance. The technological mechanism of the network has become a battlefield for all those entities to fight for their preferred policies. Ideally, it would be possible to separate policy from mechanism, enabling designs that stay out of the battle entirely. The opposite viewpoint might be to use technology to settle the battle permanently somehow. Clark, Wroclawski, et al. (2005) suggests that neither is realistic, and suggests designing with the 'tussle' in mind. The resulting design principle is reminiscent of Parnas's information hiding principle: wherever tussle can be identified or predicted, the modularization of the design should encapsulate each tussle in its own component. Just as information hiding protects the overall system from design decisions that change for technical reasons, the tussle principle encourages designs that are similarly protected from design decisions that change for political reasons.

Those who design networks in the new environment are fortunate that the designers of the most successful network ever, the Internet, have recorded a rough consensus of the design principles that follow from their model (Carpenter 1996; Kempf and Austein 2004; Bush and Meyer 2002), along with a detailed account of the priorities that motivated such a model in the first place (Clark 1988; Saltzer, Reed, and Clark 1984). Different goals will naturally require different priorities, and it is becoming increasingly clear that the Internet is not the best model for the demands of its users today. However, changing priorities can in no way diminish the value of so much distilled experience.

7.3 The OSI model

The designers of the OSI model, the other major incarnation of the network stack, enumerated their design principles with letters, and explicitly invoked them to justify each division between the famous seven layers (OSI).

(a) Do not create so many layers as to make the system engineering task of describing and integrating the layers more difficult than necessary.

(b) Create a boundary at a point where the description of services can be small and the number of interactions across the boundary are minimized.

(c) Create separate layers to handle functions that are manifestly different in the process performed or the technology involved.

(d) Collect similar functions into the same layer.

(e) Select boundaries at a point which past experience has demonstrated to be successful.

(f) Create a layer of easily localized functions so that the layer could be totally redesigned and its protocols changed in a major way to take advantage of new advances in architectural, hardware or software technology without changing the services expected from and provided to the adjacent layers.

(g) Create a boundary where it may be useful at some point in time to have the corresponding interface standardized.

(h) Create a layer where there is a need for a different level of abstraction in the handling of data, for example morphology, syntax, semantics.

(j) Allow changes of functions or protocols to be made within a layer without affecting other layers.

(k) Create for each layer, boundaries with its upper and lower layer only.

One lesson to draw is that good ideas are more universal than any one group's goals. For example, (a) is essentially the simplicity principle, (b) the separation of concerns, (f) information hiding, and (j) tussle. Since priorities change over time, perhaps the greatest impact possible from any new model is to distill and name some of those good ideas to keep them alive.

7.4 Other principles

The Network Semantics project, which developed the HOL specification of actual TCP and sockets implementations, also produced some suggestions for designing new protocols that would make similar verification easier (Bishop, Fairbairn, Norrish, Ridge, et al. Draft).

- Clearly identify the part of the overall system that the specification is intended to cover.
- Specify both the service that the protocol is intended to achieve and the protocol internals, and the relationship between the two.
- Arrange so that an efficient test oracle can be built directly from the specification.
  - In some cases, one could arrange for the specification to be completely deterministic between observable events, and there one could write those parts of the specification in an executable pure functional language, and then use that directly for testing and as an executable prototype.

  - In other cases, where one really does want to leave implementation freedom that should be factored out, one either needs a more expressive specification language and a constraint-solving checker, or one should write a test oracle directly.

- Either test (or ideally prove) that the protocol-level specification does provide the intended service.
- Set up random test generation infrastructure, tied to the test oracle, to use for implementations.

These principles tie together formal and experimental methods, suggesting that protocol designers not only specify their systems carefully, but make it as easy as possible for implementers to know whether they have faithfully reproduced the specification.

Anderson et al. (2003) produced six design principles to address network protocols that are robust to crash failures but not to semantically corrupted messages, such as might be produced by corruption or misconfiguration.

- Value conceptual simplicity.
- Minimize your dependencies.
- Verify when possible.
- Protect your resources.
- Limit the scope of vulnerability.
- Expose errors.

An additional recommendation, not one of the six but mentioned in the conclusion, is that RFCs should contain a 'Robustness Considerations' section, not unlike the 'Security Considerations' section that is already ubiquitous. I think it is a valuable suggestion, particularly taken as the design principle that the designers should put explicit, written effort into thinking about exceptional conditions.

7.5 The channel market model

The channel market model has so far inspired two new design principles: the separation of justification and the haggling principle.

Separation of justification

The principle of separation of justification is that modules encapsulate justification. Here is the pithy statement: If you would be tempted to simplify it out of a proof of correctness, make it a module.

The separation of justification might well be seen as merely a facet of Dijkstra's separation of concerns. In 'The effective arrangement of logical systems' (Dijkstra 1976), a lesser-known but worthwhile article, he explains the separation of concerns further, with practical examples, and—notable in this context—he compares such separation to the division of a logical proof into lemmas. Once the lemmas are stated and proved, they are useful elsewhere without the slightest care for how they were proved. Thus justification is clearly a kind of concern.

The reason separation of justification is worth its own name is that it is so often forgotten. Large systems are often seen as impossible to formally verify, or at least not worth the effort. But progress can be made by modularization, because any complexity that is too daunting right now can be encapsulated and proved later or at least left explicit as an assumption, an unproved lemma.

Separating the justifications of modules is also useful beyond putting off work or tackling it in manageable pieces. Veríssimo (2006) argues that it is beneficial, sometimes even necessary to divide a system into parts with different theories. For example, the classic FLP impossibility result (Fischer, Lynch, and Patterson 1985) can be overcome by designing an asynchronous system side-by-side with a synchronous system such as a failure detector (Chandra and Toueg 1996). Separate justifications, in this case, enables totally different theorems to apply to different parts of the system, and thereby new kinds of hybrid systems.

The haggling principle

The haggling principle is to make the boundaries between modules assume–guarantee specifications. Assume–guarantee specifications are a classic approach for writing formal specifications that can easily be composed (Jones 1983). However, the haggling principle is not about logic or even about formal methods. Rather, it is a reminder: think about assumptions and guarantees.

Do not assume too little or too much. When a system is too timid to assume the environment will provide a channel with the right properties, it takes on the complexity of adapting what it gets into what it needs; for example, many of the layers in the Internet model apply redundant checksums to protect against corruption that cannot happen in practice, rather than explicitly assuming an underlying channel that delivers only messages that are sent. When a system is too specific about its assumptions, those unfounded preconceptions might prevent it from being composed in ways that might otherwise be acceptable; for example, many services that build on top of the Internet protocols use the sockets interface directly, which prevents them from being composed with each other.

Do not guarantee too little or too much. Guaranteeing too little is like assuming too much: It limits composition unnecessarily. Guaranteeing too much is like assuming too little: It forces the system to do incorporate unnecessary complexity.

The assumption–guarantee dichotomy is also a valuable reminder of the structure of a system in the channel market model. Any system builds on the existing resources available from the market, then offers what it builds back to the market. Assumptions are a shopping list on a visit to the market, and guarantees are what a hawker shouts from a market stall.

CHAPTER 8

CONCLUSION

If I learned one thing from the work in this dissertation, it is certainly to be explicit about the channels. When I got stuck on the TCP specification and wrote it out using circles to represent what I knew should be agents, and big heavy lines representing some communication I knew must happen between them, it was the big heavy lines that needed to be stated more clearly. Sometimes I felt a temptation not to specify channels, and just to write more in the transitions (agents) to describe their synchronization without naming the channel they used, and that was always a mistake. Name the channels.

The primary technical contribution is the TCPR middleware, and the introduction of the application-driven recovery technique. On the other hand, the channel market model is a lesson I learned while I tried to justify for myself that the technical work made sense. Throughout the process, the model I was developing was useful again and again, sometimes just by reminding me of mistakes I knew better than to make. I believe the model itself will be the contribution of the most lasting value, because it was a lesson in dependability.

Dependability is an increasingly important concern for communication systems. As more and more services come to rely on the cloud, in particular, it is crucial to be able to reason about what people should be able to expect of them. The need for clear models that enable people to describe such services compositionally can be seen clearly. It can be seen in the software-defined network community, for example, in the search for the elusive Northbound API. It can be seen in the cloud computing community, in the efforts to pin down an abstraction for service chaining. It is crucial to know what the pieces are and how it is possible to fit them together, to know what assumptions are being made, and what promises are being made.

I believe the most influential future work to be the development of language. That includes the language of the researchers and engineers who talk about systems, as I have tried to support in the new model. It also includes formalisms. The greatest benefit from any formalism is really the careful thinking required to write down a specification. Beyond that, it is desirable to express channel markets in a language that features both first-class control channels, as in π-calculus, and logical agent specifications, as in TLA+. The temporal logic explicit in TLA+, and implicit in others such as I/O automata, is powerful; adding an epistemic or doxastic modality might make both agents and channels first-class.

Formalism is not necessary for every task, however. Remember that there are two kinds of justification. And TCP, despite its many problems, is widely regarded as very dependable—rightly so, because of its substantial experimental support. Even a formally verified protocol that could be incrementally deployed would be unlikely to supplant TCP at this point, because it is hard to compete with that justification. It is worth keeping the human need for confidence in mind when designing a system. That is the role of dependability in design.

Also remember that there are two ways to make a system more dependable. One way is to change it to meet a more useful specification, with the same confidence. The other way is to change nothing about the system, but make it easier to understand, easier to predict, better justified. The latter is the role of pedagogy. Therefore, the final future work that I will mention is teaching. Existing network protocols, experimental designs, working code, formalisms, and scientific models are all valuable ways to access the possibilities of dependable communication. A good way to test any such approach is to teach it to someone else.

APPENDIX A

THE CONNECTION SPECIFICATION

The following pages carry the complete TLA+ specification of a TCP connection from chapter 5.

MODULE *ApplicationReceiver*

MODULE *Hidden*

LOCAL INSTANCE *Data*
LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *ApplicationUpdate*

VARIABLES $chan,\ dat,\ psh,\ urg$

$Chan \triangleq$ INSTANCE $MsgChannel$ WITH $Msg \leftarrow ApplicationUpdate$

$$\begin{aligned} Init \;\triangleq\; & \land\ dat = \{\} \\ & \land\ psh = \{\} \\ & \land\ urg = 0 \end{aligned}$$

$$\begin{aligned} Update(u) \;\triangleq\; & \land\ Chan!Send(u) \\ & \land\ dat \subseteq dat' \\ & \land\ psh \subseteq psh' \\ & \land\ urg \leq urg' \\ & \land\ dat' = u.dat \\ & \land\ psh' = u.psh \\ & \land\ urg' = u.urg \end{aligned}$$

$Next \triangleq \exists\, u \in ApplicationUpdate : Update(u)$

$Spec \triangleq Init \land \Box[Next]_{\langle chan,\, dat,\, psh,\, urg \rangle} \land Chan!Spec$

VARIABLE $chan$

$Hidden(dat,\ urg,\ psh) \triangleq$ INSTANCE $Hidden$

$Spec \triangleq \boldsymbol{\exists}\, dat,\ urg,\ psh : Hidden(dat,\ urg,\ psh)!Spec$

MODULE *ApplicationSender*

MODULE *Hidden*

LOCAL INSTANCE *Data*
LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *ApplicationUpdate*

VARIABLES *chan*, *dat*, *psh*, *urg*

$Chan \triangleq \textsc{instance}\ MsgChannel\ \textsc{with}\ Msg \leftarrow ApplicationUpdate$

$$\begin{aligned}Init \triangleq\ & \land\ dat = \{\}\\ & \land\ psh = \{\}\\ & \land\ urg = 0\end{aligned}$$

$$\begin{aligned}Internal \triangleq\ & \land\ dat' \in Data\\ & \land\ dat \subseteq dat' \land dat \neq dat'\\ & \land\ [urg' = Length(dat')]_{urg}\\ & \land\ [psh' = psh \cup \{Length(dat')\}]_{psh}\\ & \land\ \textsc{unchanged}\ chan\end{aligned}$$

$$\begin{aligned}Update \triangleq\ & \land\ Chan!Send([dat \mapsto dat,\ psh \mapsto psh,\ urg \mapsto urg])\\ & \land\ \textsc{unchanged}\ \langle dat,\ psh,\ urg\rangle\end{aligned}$$

$Next \triangleq Internal \lor Update$

$Spec \triangleq Init \land \Box[Next]_{\langle chan,\ dat,\ psh,\ urg\rangle} \land Chan!Spec$

VARIABLE *chan*

$Hidden(dat,\ urg,\ psh) \triangleq \textsc{instance}\ Hidden$

$Spec \triangleq \exists\, dat,\ urg,\ psh : Hidden(dat,\ urg,\ psh)!Spec$

```
MODULE ApplicationUpdate

LOCAL INSTANCE Data
LOCAL INSTANCE Naturals

ApplicationUpdate ≜ [dat : Data, urg : Nat, Psh : SUBSET Nat]

Offset(u, n) ≜
  IF n ≤ 1 THEN u
           ELSE [dat ↦ Shift(Suffix(u.dat, n), 1 − n)
                         ∪ Prefix(u.dat, 1),
                 psh ↦ {p ∈ Nat : p + n − 1 ∈ u.psh ∧ p > 0},
                 urg ↦ IF u.urg ≥ n THEN u.urg + 1 − n ELSE 0]
```

MODULE *Connection*

VARIABLES $a,\ b$

$$
\begin{array}{lcl}
\mathit{EndA} & \triangleq & \text{INSTANCE } \mathit{Endpoint} \text{ WITH } \mathit{sndnet} \leftarrow a,\ \mathit{rcvnet} \leftarrow b \\
\mathit{EndB} & \triangleq & \text{INSTANCE } \mathit{Endpoint} \text{ WITH } \mathit{sndnet} \leftarrow b,\ \mathit{rcvnet} \leftarrow a \\
 & & \\
\mathit{NetA} & \triangleq & \text{INSTANCE } \mathit{Network} \text{ WITH } \mathit{snd} \leftarrow a,\ \mathit{rcv} \leftarrow b \\
\mathit{NetB} & \triangleq & \text{INSTANCE } \mathit{Network} \text{ WITH } \mathit{snd} \leftarrow b,\ \mathit{rcv} \leftarrow a \\
 & & \\
\mathit{Spec} & \triangleq & (\mathit{NetA}!\mathit{Spec} \land \mathit{NetB}!\mathit{Spec}) \overset{+}{-\!\!\triangleright} (\mathit{EndA}!\mathit{Spec} \land \mathit{EndB}!\mathit{Spec})
\end{array}
$$

```
──────────────────────────── MODULE Data ────────────────────────────
LOCAL INSTANCE Naturals
LOCAL INSTANCE FiniteSets

$Octets \triangleq \{n \in Nat : n < 2^8\}$

$Syn \triangleq$ CHOOSE $x : x \notin Octets$
$Fin \triangleq$ CHOOSE $x : x \notin Octets \cup \{Syn\}$

$Message \triangleq \{Syn, Fin\} \cup Octets$

$NumberedData \triangleq \{d \in$ SUBSET $(Nat \times Message) : IsFiniteSet(d)\}$

$Length(d) \triangleq Cardinality(d)$

$SegmentData \triangleq \{d \in NumberedData :$
                     $\forall \langle n, x \rangle \in d, \langle m, y \rangle \in d :$
                        $\land\ n < Length(d)$
                        $\land\ (n = m) \quad \Rightarrow (x = y)$
                        $\land\ (x = Syn) \Rightarrow (n \leq m)$
                        $\land\ (x = Fin) \Rightarrow (n \geq m)\}$

$Data \triangleq \{d \in SegmentData : d = \{\} \lor \exists \langle n, x \rangle \in d : x = Syn\}$
─────────────────────────────────────────────────────────────────────
$Prefix(s, n) \triangleq \{\langle a, x \rangle \in s : a < n\}$
$Suffix(s, n) \triangleq \{\langle a, x \rangle \in s : a \geq n\}$
$Shift(s, n) \triangleq \{\langle a + n, x \rangle : \langle a, x \rangle \in s\}$

$Merge(ds) \triangleq$
  LET $m \triangleq \{d \in Data : d \subseteq$ UNION $ds\}$
  IN   IF $m = \{\}$ THEN CHOOSE $d : d \notin Data$
                    ELSE  CHOOSE $d \in m : \forall e \in m : Length(d) \geq Length(e)$
─────────────────────────────────────────────────────────────────────
```

```
┌──────────────────────── MODULE Endpoint ────────────────────────┐
┌───────────────────────── MODULE Hidden ─────────────────────────┐
 VARIABLES sndapp, rcvapp
 VARIABLES sndnet, rcvnet

 SndApp ≜ INSTANCE ApplicationSender WITH chan ← sndapp
 RcvApp ≜ INSTANCE ApplicationReceiver WITH chan ← rcvapp

 Peer ≜ INSTANCE Peer

 Spec ≜ ∧ SndApp!Spec
        ∧ RcvApp!Spec
        ∧ Peer!Spec
└─────────────────────────────────────────────────────────────────┘

 VARIABLES sndnet, rcvnet

 Hidden(sndapp, rcvapp) ≜ INSTANCE Hidden

 Spec ≜ ∃ sndapp, rcvapp : Hidden(sndapp, rcvapp)!Spec
└─────────────────────────────────────────────────────────────────┘
```

MODULE *MsgChannel*

LOCAL INSTANCE *Sequences*

CONSTANT *Msg*

VARIABLE *chan*

$Init \triangleq chan = \langle\rangle$

$Send(m) \triangleq chan' = Append(chan, m)$

$Next \triangleq \exists\, m \in Msg : Send(m)$

$Spec \triangleq Init \land \Box[Next]_{chan}$

$Sent \triangleq chan[Len(chan)]$

MODULE *Network*

MODULE *Hidden*

LOCAL INSTANCE *Segment*

VARIABLES *snd*, *rcv*, *net*

$$\begin{array}{l} \mathit{Snd} \triangleq \textsc{instance}\ \mathit{MsgChannel}\ \textsc{with}\ \mathit{chan} \leftarrow \mathit{snd},\ \mathit{Msg} \leftarrow \mathit{Segment} \\ \mathit{Rcv} \triangleq \textsc{instance}\ \mathit{MsgChannel}\ \textsc{with}\ \mathit{chan} \leftarrow \mathit{rcv},\ \mathit{Msg} \leftarrow \mathit{Segment} \end{array}$$

$$\mathit{Init} \triangleq \mathit{net} = \{\}$$

$$\mathit{Send}(s) \triangleq \begin{array}{l} \land\ \mathit{net}' = \mathit{net} \cup \{s\} \\ \land\ \mathit{Snd}!\mathit{Send}(s) \\ \land\ \textsc{unchanged}\ \mathit{rcv} \end{array}$$

$$\mathit{Deliver}(s) \triangleq \begin{array}{l} \land\ s \in \mathit{net} \\ \land\ \textsc{unchanged}\ \mathit{snd} \\ \land\ \mathit{Rcv}!\mathit{Send}(s) \end{array}$$

$$\mathit{Drop} \triangleq \begin{array}{l} \land\ \mathit{net}' \subseteq \mathit{net} \\ \land\ \textsc{unchanged}\ \langle \mathit{snd},\ \mathit{rcv} \rangle \end{array}$$

$$\mathit{Next} \triangleq \begin{array}{l} \lor\ \exists\, s \in \mathit{Segment} : \mathit{Send}(s) \\ \lor\ \exists\, s \in \mathit{Segment} : \mathit{Deliver}(s) \\ \lor\ \mathit{Drop} \end{array}$$

$$\mathit{Fairness} \triangleq \forall\, s \in \mathit{Segment} : \mathrm{SF}_{\langle \mathit{snd},\ \mathit{rcv},\ \mathit{net} \rangle}(\mathit{Deliver}(s))$$

$$\mathit{Spec} \triangleq \begin{array}{l} \land\ \mathit{Init} \\ \land\ \Box[\mathit{Next}]_{\langle \mathit{snd},\ \mathit{rcv},\ \mathit{net} \rangle} \\ \land\ \mathit{Fairness} \\ \land\ \mathit{Snd}!\mathit{Spec} \\ \land\ \mathit{Rcv}!\mathit{Spec} \end{array}$$

VARIABLES *snd*, *rcv*

$$\mathit{Hidden}(\mathit{net}) \triangleq \textsc{instance}\ \mathit{Hidden}$$

$$\mathit{Spec} \triangleq \boldsymbol{\exists}\, \mathit{net} : \mathit{Hidden}(\mathit{net})!\mathit{Spec}$$

```
┌──────────────────────────── MODULE Peer ────────────────────────────┐
┌─────────────────────────── MODULE Hidden ───────────────────────────┐

LOCAL INSTANCE Data
LOCAL INSTANCE Segment
LOCAL INSTANCE Naturals
LOCAL INSTANCE ApplicationUpdate

VARIABLES sndapp, sndseq, snddat, sndpsh, sndurg, sndack, sndwnd, sndnet
VARIABLES rcvapp, rcvseq, rcvdat, rcvpsh, rcvurg, rcvack, rcvwnd, rcvnet

SndApp ≜ INSTANCE MsgChannel WITH chan ← sndapp, Msg ← ApplicationUpdate
RcvApp ≜ INSTANCE MsgChannel WITH chan ← rcvapp, Msg ← ApplicationUpdate

SndTcp ≜ INSTANCE Sender WITH seq ← sndseq,
                              dat ← snddat,
                              psh ← sndpsh,
                              urg ← sndurg,
                              ack ← sndack,
                              wnd ← sndwnd,
                              app ← sndapp

RcvTcp ≜ INSTANCE Receiver WITH seq ← rcvseq,
                                dat ← rcvdat,
                                psh ← rcvpsh,
                                urg ← rcvurg,
                                ack ← rcvack,
                                wnd ← rcvwnd,
                                app ← rcvapp

SndMux ≜ INSTANCE SegmentMux WITH seg ← sndnet,
                                  seq ← sndseq,
                                  dat ← snddat,
                                  psh ← sndpsh,
                                  urg ← sndurg,
                                  ack ← rcvack,
                                  wnd ← rcvwnd

RcvMux ≜ INSTANCE SegmentMux WITH seg ← rcvnet,
                                  seq ← rcvseq,
                                  dat ← rcvdat,
                                  psh ← rcvpsh,
                                  urg ← rcvurg,
                                  ack ← sndack,
                                  wnd ← sndwnd

SndNet ≜ INSTANCE MsgChannel WITH chan ← sndnet, Msg ← Segment
RcvNet ≜ INSTANCE MsgChannel WITH chan ← rcvnet, Msg ← Segment

Spec ≜ ∧ SndApp!Spec
       ∧ SndTcp!Spec
       ∧ SndMux!Spec
       ∧ SndNet!Spec
```

$$
\begin{aligned}
&\qquad \land\ RcvApp\,!\,Spec\\
&\qquad \land\ RcvTcp\,!\,Spec\\
&\qquad \land\ RcvMux\,!\,Spec\\
&\qquad \land\ RcvNet\,!\,Spec
\end{aligned}
$$

VARIABLES $sndapp,\ rcvapp$
VARIABLES $sndnet,\ rcvnet$

$$
\begin{aligned}
&Hidden(sndseq,\ snddat,\ sndpsh,\ sndurg,\ sndack,\ sndwnd,\\
&\qquad rcvseq,\ rcvdat,\ rcvpsh,\ rcvurg,\ rcvack,\ rcvwnd)\ \triangleq\ \textsc{instance}\ Hidden
\end{aligned}
$$

$$
\begin{aligned}
&Spec\ \triangleq\ \exists\, sndseq,\ snddat,\ sndpsh,\ sndurg,\ sndack,\ sndwnd,\\
&\qquad\qquad rcvseq,\ rcvdat,\ rcvpsh,\ rcvurg,\ rcvack,\ rcvwnd :\\
&\qquad\qquad\quad Hidden(sndseq,\ snddat,\ sndpsh,\ sndurg,\ sndack,\ sndwnd,\\
&\qquad\qquad\qquad\quad rcvseq,\ rcvdat,\ rcvpsh,\ rcvurg,\ rcvack,\ rcvwnd)\,!\,Spec
\end{aligned}
$$

```
──────────────────────── MODULE Receiver ────────────────────────
──────────────────────── MODULE Hidden ─────────────────────────

LOCAL INSTANCE Data
LOCAL INSTANCE Naturals
LOCAL INSTANCE ApplicationUpdate

VARIABLES seq, dat, psh, urg, ack, wnd, app
VARIABLES maxdat, maxpsh, maxurg, maxack, maxwnd

Seq  ≜ INSTANCE MsgChannel WITH chan ← seq, Msg  ← Nat
Dat  ≜ INSTANCE MsgChannel WITH chan ← dat, Msg  ← SegmentData
Psh  ≜ INSTANCE MsgChannel WITH chan ← psh, Msg  ← Nat
Urg  ≜ INSTANCE MsgChannel WITH chan ← urg, Msg  ← Nat
Ack  ≜ INSTANCE MsgChannel WITH chan ← ack, Msg  ← Nat
Wnd ≜ INSTANCE MsgChannel WITH chan ← wnd, Msg ← Nat
App  ≜ INSTANCE MsgChannel WITH chan ← app, Msg  ← ApplicationUpdate

Init ≜  ∧ maxdat  = {}
        ∧ maxpsh  = {}
        ∧ maxurg  = 0
        ∧ maxack  = 0
        ∧ maxwnd = 0

Internal ≜  ∧ maxack′  ∈ Nat
            ∧ maxwnd′ ∈ Nat
            ∧ maxack′  ≤ Length(maxdat)
            ∧ maxack′  ≥ maxack
            ∧ maxwnd′ ≥ maxack′
            ∧ UNCHANGED ⟨seq, dat, psh, urg, ack, wnd, app⟩
            ∧ UNCHANGED ⟨maxdat, maxpsh, maxurg⟩

Update ≜  ∧ App!Send([dat ↦ maxdat, psh ↦ maxpsh, urg ↦ maxurg])
          ∧ UNCHANGED ⟨seq, dat, psh, urg, ack, wnd⟩
          ∧ UNCHANGED ⟨maxdat, maxpsh, maxurg, maxack, maxwnd⟩

Send ≜  ∧ ∃ n : Seq!Send(n)
        ∧ IF maxack > 0 THEN Ack!Send(maxack) ∧ Wnd!Send(maxwnd)
                        ELSE UNCHANGED ⟨ack, wnd⟩
        ∧ UNCHANGED ⟨dat, psh, urg, app⟩
        ∧ UNCHANGED ⟨maxdat, maxpsh, maxurg, maxack, maxwnd⟩

Receive(n, d, p, u) ≜  ∧ maxdat′ ∈ Data
                       ∧ maxdat′ = Merge({maxdat, Prefix(Shift(d, n), maxwnd)})
                       ∧ Seq!Send(n)
                       ∧ Dat!Send(d)
                       ∧ IF Psh!Send(p) THEN maxpsh′ = maxpsh ∪ {p}
                                        ELSE UNCHANGED ⟨psh, maxpsh⟩
                       ∧ IF Urg!Send(u) THEN IF u > maxurg THEN maxurg′ = u
                                                           ELSE UNCHANGED maxurg
                                        ELSE UNCHANGED ⟨urg, maxurg⟩
                       ∧ UNCHANGED ⟨ack, wnd, app⟩
                       ∧ UNCHANGED ⟨maxack, maxwnd⟩
```

$$\begin{aligned} \mathit{Next} \triangleq\ & \lor \mathit{Internal} \\ & \lor \mathit{Update} \\ & \lor \mathit{Send} \\ & \lor \exists\, n,\ p,\ u \in \mathit{Nat},\ d \in \mathit{SegmentData} : \mathit{Receive}(n,\ d,\ p,\ u) \end{aligned}$$

$$\mathit{Liveness} \triangleq \forall\, n \in \mathit{Nat} : \mathrm{SF}_{\mathit{app}}(\mathit{Update} \land \exists\, p \in \mathit{maxpsh} : \mathit{App}!\mathit{Init} \lor p > \mathit{Length}(\mathit{App}!\mathit{Sent.dat}))$$

$$\begin{aligned} \mathit{Spec} \triangleq\ & \land \mathit{Init} \\ & \land \Box[\mathit{Next}]_{\langle \mathit{seq},\, \mathit{dat},\, \mathit{psh},\, \mathit{urg},\, \mathit{ack},\, \mathit{wnd},\, \mathit{app},\, \mathit{maxdat},\, \mathit{maxpsh},\, \mathit{maxurg},\, \mathit{maxack},\, \mathit{maxwnd} \rangle} \\ & \land \mathit{Liveness} \\ & \land \mathit{Seq}!\mathit{Spec} \\ & \land \mathit{Dat}!\mathit{Spec} \\ & \land \mathit{Psh}!\mathit{Spec} \\ & \land \mathit{Urg}!\mathit{Spec} \\ & \land \mathit{Ack}!\mathit{Spec} \\ & \land \mathit{Wnd}!\mathit{Spec} \\ & \land \mathit{App}!\mathit{Spec} \end{aligned}$$

$\textsc{variables}\ \mathit{seq},\ \mathit{dat},\ \mathit{psh},\ \mathit{urg},\ \mathit{ack},\ \mathit{wnd},\ \mathit{app}$

$$\mathit{Hidden}(\mathit{maxdat},\ \mathit{maxpsh},\ \mathit{maxurg},\ \mathit{maxack},\ \mathit{maxwnd}) \triangleq \textsc{instance}\ \mathit{Hidden}$$

$$\begin{aligned} \mathit{Spec} \triangleq\ & \boldsymbol{\exists}\, \mathit{maxdat},\ \mathit{maxpsh},\ \mathit{maxurg},\ \mathit{maxack},\ \mathit{maxwnd} : \\ & \quad \mathit{Hidden}(\mathit{maxdat},\ \mathit{maxpsh},\ \mathit{maxurg},\ \mathit{maxack},\ \mathit{maxwnd})!\mathit{Spec} \end{aligned}$$

```
──────────────────────── MODULE SegmentMux ────────────────────────
LOCAL INSTANCE Data
LOCAL INSTANCE Segment
LOCAL INSTANCE Naturals

VARIABLES seg, seq, dat, psh, urg, ack, wnd

Seg  ≜ INSTANCE MsgChannel WITH chan ← seg, Msg  ← Segment
Seq  ≜ INSTANCE MsgChannel WITH chan ← seq, Msg  ← Nat
Dat  ≜ INSTANCE MsgChannel WITH chan ← dat, Msg  ← SegmentData
Psh  ≜ INSTANCE MsgChannel WITH chan ← psh, Msg  ← Nat
Urg  ≜ INSTANCE MsgChannel WITH chan ← urg, Msg  ← Nat
Ack  ≜ INSTANCE MsgChannel WITH chan ← ack, Msg  ← Nat
Wnd  ≜ INSTANCE MsgChannel WITH chan ← wnd, Msg ← Nat

Demux(s) ≜ ∧ Seg!Send(s)
           ∧ Seq!Send(s.seq)
           ∧ Dat!Send(s.dat)
           ∧ IF s.ctl.psh THEN Psh!Send(s.seq + Length(s.data))
                          ELSE UNCHANGED psh
           ∧ IF s.ctl.urg THEN Urg!Send(s.urg + s.seq)
                          ELSE UNCHANGED urg
           ∧ IF s.ctl.ack THEN ∧ Ack!Send(s.ack)
                               ∧ Wnd!Send(s.wnd + s.ack)
                          ELSE UNCHANGED ⟨ack, wnd⟩

Next ≜ ∃ s ∈ Segment : Demux(s)

Spec ≜ ∧ □[Next]_⟨seg, seq, dat, psh, urg, ack, wnd⟩
       ∧ Seg!Spec
       ∧ Seq!Spec
       ∧ Dat!Spec
       ∧ Psh!Spec
       ∧ Urg!Spec
       ∧ Ack!Spec
       ∧ Wnd!Spec
────────────────────────────────────────────────────────────────────
```

```
──────────────────────── MODULE Segment ────────────────────────
LOCAL INSTANCE Data
LOCAL INSTANCE Naturals

Segment ≜ [seq  : Nat,
           ack  : Nat,
           ctl  : [ack ↦ BOOLEAN ,
                   urg ↦ BOOLEAN ,
                   psh ↦ BOOLEAN ],
           wnd : Nat,
           urg  : Nat,
           dat  : SegmentData]
────────────────────────────────────────────────────────────────
```

```
┌──────────────────────────── MODULE Sender ────────────────────────────┐
┌──────────────────────────── MODULE Hidden ────────────────────────────┐
LOCAL INSTANCE Data
LOCAL INSTANCE Naturals
LOCAL INSTANCE WindowUpdate
LOCAL INSTANCE ApplicationUpdate

VARIABLES seq, dat, psh, urg, ack, wnd, app
VARIABLES maxdat, maxpsh, maxurg, maxack, maxwnd

Seq  ≜ INSTANCE MsgChannel WITH chan ← seq, Msg  ← Nat
Dat  ≜ INSTANCE MsgChannel WITH chan ← dat, Msg  ← SegmentData
Psh  ≜ INSTANCE MsgChannel WITH chan ← psh, Msg  ← Nat
Urg  ≜ INSTANCE MsgChannel WITH chan ← urg, Msg  ← Nat
Ack  ≜ INSTANCE MsgChannel WITH chan ← ack, Msg  ← Nat
Wnd  ≜ INSTANCE MsgChannel WITH chan ← wnd, Msg ← Nat
App  ≜ INSTANCE MsgChannel WITH chan ← app, Msg  ← ApplicationUpdate

Init ≜  ∧ maxdat  = {}
        ∧ maxpsh  = {}
        ∧ maxurg  = 0
        ∧ maxack  = 0
        ∧ maxwnd ∈ WindowUpdate
        ∧ maxwnd.seq = 0
        ∧ maxwnd.ack = 0

Update(u) ≜  ∧ maxdat ⊆ maxdat′
             ∧ maxpsh ⊆ maxpsh′
             ∧ maxurg ≤ maxurg′
             ∧ maxdat′ = u.dat
             ∧ maxpsh′ = u.psh
             ∧ maxurg′ = u.urg
             ∧ App!Send(u)
             ∧ UNCHANGED ⟨seq, dat, psh, urg, ack, wnd⟩
             ∧ UNCHANGED ⟨maxack, maxwnd⟩

Send(n, d) ≜  ∧ n ≥ maxack
              ∧ maxack = 0 ⇒ ⟨0, Syn⟩ ∈ d
              ∧ IF maxwnd.len > maxack THEN n + Length(d) ≤ maxwnd.len
                                       ELSE  n + Length(d) ≤ maxack + 1
              ∧ Shift(d, n) ⊆ maxdat
              ∧ Seq!Send(n)
              ∧ Dat!Send(d)
              ∧ Psh!Send(n + Length(d)) ≡ ∃ p ∈ maxpsh :  ∧ p > n
                                                         ∧ p ≤ n + Length(d)
              ∧ Urg!Send(maxurg) ≡ maxurg ≥ n
              ∧ UNCHANGED ⟨ack, wnd, app⟩
              ∧ UNCHANGED ⟨maxdat, maxpsh, maxurg, maxack, maxwnd⟩

Receive(n, a, w) ≜  ∧ IF a > maxack THEN maxack′ = a
                                    ELSE  UNCHANGED maxack
                    ∧ LET u ≜ [len ↦ w, seq ↦ n, ack ↦ a]
```

$$
\begin{array}{ll}
\quad\quad\textsc{in} & \lor\ \mathit{WndAscending}(u,\ \mathit{maxwnd}) \land \mathit{maxwnd}' = u \\
 & \lor\ \mathit{WndAscending}(\mathit{maxwnd},\ u) \land \textsc{unchanged}\ \mathit{maxwnd}
\end{array}
$$

$$
\begin{array}{l}
\land\ \mathit{Seq}\,!\,\mathit{Send}(n) \\
\land\ \mathit{Ack}\,!\,\mathit{Send}(a) \\
\land\ \mathit{Wnd}\,!\,\mathit{Send}(w) \\
\land\ \textsc{unchanged}\ \langle \mathit{dat},\ \mathit{psh},\ \mathit{urg},\ \mathit{app} \rangle \\
\land\ \textsc{unchanged}\ \langle \mathit{maxdat},\ \mathit{maxpsh},\ \mathit{maxurg} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Next} & \triangleq & \lor\ \exists\, u \in \mathit{ApplicationUpdate} : \mathit{Update}(u) \\
 & & \lor\ \exists\, n \in \mathit{Nat},\ d \in \mathit{SegmentData} : \mathit{Send}(n,\ d) \\
 & & \lor\ \exists\, n,\ a \in \mathit{Nat},\ w \in \mathit{WindowUpdate} : \mathit{Receive}(n,\ a,\ w)
\end{array}
$$

$$
\mathit{Liveness} \triangleq \forall\, n \in \mathit{Nat} : \exists\, m \in \mathit{Nat},\ d \in \mathit{SegmentData} : \mathrm{SF}_{\mathit{dat}}(\exists\, p \in \mathit{maxpsh} : p \geq n \land \mathit{Send}(m,\ d))
$$

$$
\begin{array}{lll}
\mathit{Spec} & \triangleq & \land\ \mathit{Init} \\
 & & \land\ \Box[\mathit{Next}]_{\langle \mathit{seq},\, \mathit{dat},\, \mathit{psh},\, \mathit{urg},\, \mathit{ack},\, \mathit{wnd},\, \mathit{app},\, \mathit{maxdat},\, \mathit{maxpsh},\, \mathit{maxurg},\, \mathit{maxack},\, \mathit{maxwnd} \rangle} \\
 & & \land\ \mathit{Liveness} \\
 & & \land\ \mathit{Seq}\,!\,\mathit{Spec} \\
 & & \land\ \mathit{Dat}\,!\,\mathit{Spec} \\
 & & \land\ \mathit{Psh}\,!\,\mathit{Spec} \\
 & & \land\ \mathit{Urg}\,!\,\mathit{Spec} \\
 & & \land\ \mathit{Ack}\,!\,\mathit{Spec} \\
 & & \land\ \mathit{Wnd}\,!\,\mathit{Spec} \\
 & & \land\ \mathit{App}\,!\,\mathit{Spec}
\end{array}
$$

$\textsc{variables}\ \mathit{seq},\ \mathit{dat},\ \mathit{psh},\ \mathit{urg},\ \mathit{ack},\ \mathit{wnd},\ \mathit{app}$

$$
\mathit{Hidden}(\mathit{maxdat},\ \mathit{maxpsh},\ \mathit{maxurg},\ \mathit{maxack},\ \mathit{maxwnd}) \triangleq \textsc{instance}\ \mathit{Hidden}
$$

$$
\begin{array}{ll}
\mathit{Spec} \triangleq & \boldsymbol{\exists}\, \mathit{maxdat},\ \mathit{maxpsh},\ \mathit{maxurg},\ \mathit{maxack},\ \mathit{maxwnd} : \\
 & \quad \mathit{Hidden}(\mathit{maxdat},\ \mathit{maxpsh},\ \mathit{maxurg},\ \mathit{maxack},\ \mathit{maxwnd})\,!\,\mathit{Spec}
\end{array}
$$

MODULE *WindowUpdate*

LOCAL INSTANCE *Naturals*

$WindowUpdate \triangleq [len : Nat,\ seq : Nat,\ ack : Nat]$

$$WndAscending(a,\ b) \triangleq \begin{array}{l} \lor\ a.seq < b.seq \\ \lor\ a.seq = b.seq \land a.ack \leq b.ack \end{array}$$

APPENDIX B

THE RECOVERING CONNECTION SPECIFICATION

The following pages carry the complete TLA+
specification of a recovering connection from chapter 6.

```
──────────────── MODULE RecApplicationReceiver ────────────────
──────────────────── MODULE Hidden ────────────────────

LOCAL INSTANCE Data
LOCAL INSTANCE Naturals
LOCAL INSTANCE ApplicationUpdate

VARIABLES f, chan, recack, dat, psh, urg, off

Replicas ≜ INSTANCE Replicas WITH r ← chan

Chan(n) ≜ INSTANCE MsgChannel WITH chan ← chan[n], Msg ← ApplicationUpdate
RecAck ≜ INSTANCE MsgChannel WITH chan ← recack, Msg ← Nat

Init ≜ ∧ dat = {}
       ∧ psh = {}
       ∧ urg = 0
       ∧ off  = [n ∈ Nat ↦ 0]

Update(u) ≜ ∧ Chan(f)!Send(Offset(u, off[f]))
            ∧ RecAck!Send(Length(dat′))
            ∧ dat ⊆ dat′
            ∧ psh ⊆ psh′
            ∧ urg ≤ urg′
            ∧ dat′ = u.dat
            ∧ psh′ = u.psh
            ∧ urg′ = u.urg
            ∧ UNCHANGED ⟨f, off⟩

Fail  ≜ ∧ Replicas!Fail
        ∧ off′ = [n ∈ Nat ↦ IF n < f′ THEN off[n] ELSE  Length(dat)]
        ∧ UNCHANGED ⟨chan, recack, dat, psh, urg⟩

Next  ≜ ∨ ∃ u ∈ ApplicationUpdate : Update(u)
        ∨ Fail

Spec  ≜ ∧ Init
        ∧ □[Next]_⟨f, chan, recack, dat, psh, urg, off⟩
        ∧ ∀ n ∈ Nat : Chan(n)!Spec
        ∧ Replicas!Spec
        ∧ RecAck!Spec
────────────────────────────────────────────────────────

VARIABLE f, chan, recack

Hidden(dat, urg, psh, off) ≜ INSTANCE Hidden

Spec ≜ ∃∃ dat, urg, psh, off : Hidden(dat, urg, psh, off)!Spec
────────────────────────────────────────────────────────
```

MODULE *RecApplicationSender*

MODULE *Hidden*

$\textsc{local instance}\ \mathit{Data}$
$\textsc{local instance}\ \mathit{Naturals}$
$\textsc{local instance}\ \mathit{ApplicationUpdate}$

$\textsc{variables}\ f,\ \mathit{chan},\ \mathit{recoff},\ \mathit{dat},\ \mathit{psh},\ \mathit{urg},\ \mathit{off}$

$\mathit{Replicas} \triangleq \textsc{instance}\ \mathit{Replicas}\ \textsc{with}\ r \leftarrow \mathit{chan}$

$\mathit{Chan}(n) \triangleq \textsc{instance}\ \mathit{MsgChannel}\ \textsc{with}\ \mathit{chan} \leftarrow \mathit{chan}[n],\ \mathit{Msg} \leftarrow \mathit{ApplicationUpdate}$
$\mathit{RecOff} \triangleq \textsc{instance}\ \mathit{MsgChannel}\ \textsc{with}\ \mathit{chan} \leftarrow \mathit{recoff},\ \mathit{Msg} \leftarrow \mathit{Nat}$

$$\begin{aligned}\mathit{Init} \triangleq\ & \land \mathit{dat} = \{\}\\ & \land \mathit{psh} = \{\}\\ & \land \mathit{urg} = 0\\ & \land \mathit{off}\ = [n \in \mathit{Nat} \mapsto 0]\end{aligned}$$

$$\begin{aligned}\mathit{Internal} \triangleq\ & \land \mathit{dat}' \in \mathit{Data}\\ & \land \mathit{dat} \subseteq \mathit{dat}' \land \mathit{dat} \neq \mathit{dat}'\\ & \land [\mathit{psh}' = \mathit{psh} \cup \{\mathit{Length}(\mathit{dat}')\}]_{\mathit{psh}}\\ & \land [\mathit{urg}' = \mathit{Length}(\mathit{dat}')]_{\mathit{urg}}\\ & \land \textsc{unchanged}\ \langle f,\ \mathit{chan},\ \mathit{recoff},\ \mathit{off}\rangle\end{aligned}$$

$$\begin{aligned}\mathit{Update} \triangleq\ & \land \mathit{Chan}(f)!\mathit{Send}(\mathit{Offset}([\mathit{dat} \mapsto \mathit{dat},\\ & \qquad\qquad\qquad\qquad\quad \mathit{psh} \mapsto \mathit{psh},\\ & \qquad\qquad\qquad\qquad\quad \mathit{urg} \mapsto \mathit{urg}],\ \mathit{off}[f]))\\ & \land \textsc{unchanged}\ \langle f,\ \mathit{recoff},\ \mathit{dat},\ \mathit{psh},\ \mathit{urg},\ \mathit{off}\rangle\end{aligned}$$

$$\begin{aligned}\mathit{Fail}(o) \triangleq\ & \land \mathit{Replicas}!\mathit{Fail}\\ & \land \mathit{RecOff}!\mathit{Send}(o)\\ & \land \mathit{off}' = [n \in \mathit{Nat} \mapsto \textsc{if}\ n < f'\ \textsc{then}\ \mathit{off}[n]\ \textsc{else}\ \ o]\\ & \land \textsc{unchanged}\ \langle \mathit{chan},\ \mathit{dat},\ \mathit{psh},\ \mathit{urg}\rangle\end{aligned}$$

$$\mathit{Next} \triangleq \mathit{Internal} \lor \mathit{Update} \lor \exists\, o \in \mathit{Nat} : \mathit{Fail}(o)$$

$$\begin{aligned}\mathit{Spec} \triangleq\ & \land \mathit{Init}\\ & \land \Box[\mathit{Next}]_{\langle f,\ \mathit{chan},\ \mathit{recoff},\ \mathit{dat},\ \mathit{psh},\ \mathit{urg},\ \mathit{off}\rangle}\\ & \land \forall\, n \in \mathit{Nat} : \mathit{Chan}(n)!\mathit{Spec}\\ & \land \mathit{Replicas}!\mathit{Spec}\\ & \land \mathit{RecOff}!\mathit{Spec}\end{aligned}$$

$\textsc{variables}\ f,\ \mathit{chan},\ \mathit{recoff}$

$\mathit{Hidden}(\mathit{dat},\ \mathit{urg},\ \mathit{psh},\ \mathit{off}) \triangleq \textsc{instance}\ \mathit{Hidden}$

$\mathit{Spec} \triangleq \boldsymbol{\exists}\, \mathit{dat},\ \mathit{urg},\ \mathit{psh},\ \mathit{off} : \mathit{Hidden}(\mathit{dat},\ \mathit{urg},\ \mathit{psh},\ \mathit{off})!\mathit{Spec}$

$\textsc{module}\ \mathit{RecConnection}$

$\textsc{variables}\ a,\ b$

$$\begin{array}{lcl}
\mathit{EndA} & \triangleq & \textsc{instance}\ \mathit{RecEndpoint}\ \textsc{with}\ \mathit{sndnet} \leftarrow a,\ \mathit{rcvnet} \leftarrow b \\
\mathit{EndB} & \triangleq & \textsc{instance}\ \mathit{Endpoint}\ \textsc{with}\ \mathit{sndnet} \leftarrow b,\ \mathit{rcvnet} \leftarrow a
\end{array}$$

$$\begin{array}{lcl}
\mathit{NetA} & \triangleq & \textsc{instance}\ \mathit{Network}\ \textsc{with}\ \mathit{snd} \leftarrow a,\ \mathit{rcv} \leftarrow b \\
\mathit{NetB} & \triangleq & \textsc{instance}\ \mathit{Network}\ \textsc{with}\ \mathit{snd} \leftarrow b,\ \mathit{rcv} \leftarrow a
\end{array}$$

$$\mathit{Spec} \ \triangleq\ (\mathit{NetA}!\mathit{Spec} \land \mathit{NetB}!\mathit{Spec}) \overset{+}{-\!\!\triangleright} (\mathit{EndA}!\mathit{Spec} \land \mathit{EndB}!\mathit{Spec})$$

$\mathit{Connection} \triangleq \textsc{instance}\ \mathit{Connection}$

$\textsc{theorem}\ \mathit{Spec} \Rightarrow \mathit{Connection}!\mathit{Spec}$

```
MODULE RecEndpoint

MODULE Hidden

VARIABLES f, sndpeers, rcvpeers, recoff, recack
VARIABLES sndint, rcvint, sndapp, rcvapp, sndnet, rcvnet

SndApp ≜ INSTANCE RecApplicationSender WITH chan ← sndapp
RcvApp ≜ INSTANCE RecApplicationReceiver WITH chan ← rcvapp

Peers ≜ INSTANCE RecPeers WITH sndnet ← sndpeers, rcvnet ← rcvpeers

SndInt ≜ INSTANCE Network WITH snd ← sndint, rcv ← sndpeers
RcvInt ≜ INSTANCE Network WITH snd ← rcvint, rcv ← rcvpeers

Middleware ≜ INSTANCE RecMiddleware WITH sndext ← sndnet, rcvext ← rcvnet

Spec ≜ ∧ Middleware!Spec
       ∧ SndApp!Spec
       ∧ RcvApp!Spec
       ∧ SndInt!Spec
       ∧ RcvInt!Spec
       ∧ Peers!Spec


VARIABLES sndnet, rcvnet

Hidden(f, sndpeers, rcvpeers, recoff, recack,
       sndint, rcvint, sndapp, rcvapp) ≜ INSTANCE Hidden

Spec ≜ ∃ f, sndpeers, rcvpeers, recoff, recack,
         sndint, rcvint, sndapp, rcvapp :
           Hidden(f, sndpeers, rcvpeers, recoff, recack,
                  sndint, rcvint, sndapp, rcvapp)!Spec
```

```
┌──────────────────────────── MODULE RecMiddleware ────────────────────────────┐
┌────────────────────────────── MODULE Hidden ─────────────────────────────────┐

LOCAL INSTANCE Data
LOCAL INSTANCE Segment
LOCAL INSTANCE Naturals
LOCAL INSTANCE WindowUpdate

VARIABLES f, sndint, rcvint, sndext, rcvext, recoff, recack
VARIABLES sndoff, sndwnd, sndack, rcvack, recovering

SndReplicas ≜ INSTANCE Replicas WITH r ← sndint
RcvReplicas ≜ INSTANCE Replicas WITH r ← rcvint

SndInt(n) ≜ INSTANCE MsgChannel WITH chan ← sndint[n], Msg ← Segment
RcvInt(n) ≜ INSTANCE MsgChannel WITH chan ← rcvint[n], Msg  ← Segment

SndExt  ≜ INSTANCE MsgChannel WITH chan ← sndext, Msg ← Segment
RcvExt  ≜ INSTANCE MsgChannel WITH chan ← rcvext, Msg  ← Segment

RecOff  ≜ INSTANCE MsgChannel WITH chan ← recoff, Msg  ← Nat
RecAck  ≜ INSTANCE MsgChannel WITH chan ← recack, Msg  ← Nat

Init ≜  ∧ sndoff   = [n ∈ Nat ↦ 0]
        ∧ sndwnd ∈ WindowUpdate
        ∧ sndwnd.seq = 0
        ∧ sndwnd.ack = 0
        ∧ sndack = 0
        ∧ rcvack  = 0
        ∧ recovering = FALSE

Update(a) ≜  ∧ RecAck!Send(a)
             ∧ rcvack′ ≥ rcvack
             ∧ rcvack′ = a
             ∧ UNCHANGED ⟨f, sndint, rcvint, sndext, rcvext, recoff⟩
             ∧ UNCHANGED ⟨sndoff, sndwnd, sndack, recovering⟩

Fail ≜  ∧ SndReplicas!Fail
        ∧ RcvReplicas!Fail
        ∧ RecOff!Send(sndack)
        ∧ sndoff′ = [n ∈ Nat ↦ IF n < f′ THEN sndoff[n] ELSE  sndack]
        ∧ recovering′
        ∧ UNCHANGED ⟨sndint, rcvint, sndext, rcvext, recack⟩
        ∧ UNCHANGED ⟨sndwnd, sndack, rcvack⟩

Recover(s) ≜  ∧ ⟨0, Syn⟩ ∈ s.dat ∧ rcvack > 0
              ∧ SndInt(f)!Send(s)
              ∧ RcvInt(f)!Send([seq  ↦ rcvack − 1,
                                ack  ↦ 1,
                                ctl  ↦ [ack ↦ TRUE, urg ↦ FALSE, psh ↦ FALSE],
                                wnd ↦ sndwnd.len,
                                urg  ↦ 0,
                                dat  ↦ {}])
              ∧ UNCHANGED ⟨f, sndext, rcvext, recack, recoff⟩
```

```
                 ∧ UNCHANGED ⟨sndoff, sndwnd, sndack, rcvack, recovering⟩

SndSeg(s) ≜ ∧ ⟨0, Syn⟩ ∉ s.dat ∨ rcvack = 0
            ∧ ¬recovering′
            ∧ SndInt(f)!Send(s)
            ∧ SndExt!Send([s EXCEPT !.seq = @ + sndoff[f],
                                    !.ack = rcvack])
            ∧ UNCHANGED ⟨f, rcvint, rcvext, recack, recoff⟩
            ∧ UNCHANGED ⟨sndoff, sndwnd, sndack, rcvack⟩

RcvSeg(s) ≜ ∧ RcvExt!Send(s)
            ∧ RcvInt(f)!Send([s EXCEPT !.ack = sndack′ − sndoff[f]]) ≡ ¬recovering
            ∧ IF s.ctl.ack
                THEN ∧ IF s.ack > sndack THEN sndack′ = s.ack
                                         ELSE UNCHANGED sndack
                     ∧ LET u ≜ [len ↦ s.wnd + s.ack,
                                seq ↦ s.seq,
                                ack ↦ s.ack]
                        IN   ∨ WndAscending(u, sndwnd) ∧ sndwnd′ = u
                             ∨ WndAscending(sndwnd, u) ∧ UNCHANGED sndwnd
                ELSE UNCHANGED ⟨sndwnd, sndack⟩
            ∧ UNCHANGED ⟨f, sndint, sndext, recack, recoff⟩
            ∧ UNCHANGED ⟨sndoff, rcvack, recovering⟩

Next ≜ ∨ ∃ a ∈ Nat : Update(a)
       ∨ ∃ s ∈ Segment : ∨ Recover(s)
                         ∨ SndSeg(s)
                         ∨ RcvSeg(s)
       ∨ Fail

Spec ≜ ∧ Init
       ∧ □[Next]_⟨f, sndint, rcvint, sndext, rcvext, recoff, recack, sndoff, sndwnd, sndack, rcvack, recovering⟩
       ∧ ∀ n ∈ Nat : SndInt(n)!Spec
       ∧ ∀ n ∈ Nat : RcvInt(n)!Spec
       ∧ SndReplicas!Spec
       ∧ RcvReplicas!Spec
       ∧ RecOff!Spec
       ∧ RecAck!Spec
       ∧ SndExt!Spec
       ∧ RcvExt!Spec

VARIABLES f, sndint, rcvint, sndext, rcvext, recoff, recack

Hidden(sndoff, sndwnd, sndack, rcvack, recovering) ≜ INSTANCE Hidden

Spec ≜ ∃ sndoff, sndwnd, sndack, rcvack, recovering :
            Hidden(sndoff, sndwnd, sndack, rcvack, recovering)!Spec
```

```
──────────────────────── MODULE RecPeers ────────────────────────
LOCAL INSTANCE Naturals

VARIABLE f, sndapp, rcvapp, sndnet, rcvnet

SndApp ≜ INSTANCE Replicas WITH r ← sndapp
RcvApp ≜ INSTANCE Replicas WITH r ← rcvapp
SndNet ≜ INSTANCE Replicas WITH r ← sndnet
RcvNet ≜ INSTANCE Replicas WITH r ← rcvnet

Peers(n) ≜ INSTANCE Peer WITH sndapp ← sndapp[n],
                              rcvapp ← rcvapp[n],
                              sndnet ← sndnet[n],
                              rcvnet ← rcvnet[n]

Spec ≜ ∧ SndApp!Spec
       ∧ RcvApp!Spec
       ∧ SndNet!Spec
       ∧ RcvNet!Spec
       ∧ ∀ n ∈ Nat : Peers(n)!Spec
─────────────────────────────────────────────────────────────────
```

MODULE *Replicas*

LOCAL INSTANCE *Naturals*

VARIABLE $f,\ r$

$\mathit{TypeInvariant} \triangleq f \in \mathit{Nat} \land r = [n \in \mathit{Nat} \mapsto r[n]]$

$\mathit{Init} \triangleq f = 0 \land \mathit{TypeInvariant}$

$\mathit{Fail} \triangleq f' = f + 1 \land \textsc{unchanged}\ r$

$$\begin{aligned}\mathit{Transition} \triangleq\ & \land \forall\, n \in \mathit{Nat} : n \neq f \Rightarrow \textsc{unchanged}\ r[n] \\ & \land \mathit{TypeInvariant} \\ & \land \textsc{unchanged}\ f\end{aligned}$$

$\mathit{Next} \triangleq \mathit{Fail} \lor \mathit{Transition}$

$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle f,\, r \rangle}$

BIBLIOGRAPHY

Martín Abadi and Leslie Lamport (1991). "The Existence of Refinement Mappings". In: *Theor. Comput. Sci.* 82.2.

Andrei Agapi, Ken Birman, Robert M. Broberg, Chase Cotton, Thilo Kielmann, Martin Millnert, Rick Payne, Robert Surton, and Robbert van Renesse (Sept. 2011). "Routers for the Cloud". In: *Internet Computing* 15.5.

N. Aghdaie and Y. Tamir (Apr. 2001). "Client-Transparent Fault-Tolerant Web Service". In: *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference (IPCCC 2001)*. Phoenix, AZ.

N. Aghdaie and Y. Tamir (Jan. 2009). "CoRAL: A transparent fault-tolerant web service". In: *Journal of Systems and Software* 82 (1).

Bowen Alpern and Fred B. Schneider (1984). *Defining Liveness*. Tech. rep. Ithaca, NY, USA.

L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov (Apr. 2001). "Wrapping Server-Side TCP to Mask Connection Failures". In: *Proc. of Infocom 2001*. Anchorage, Alaska, pp. 329–338.

Tom Anderson, Scott Shenker, Ion Stoica, and David Wetherall (Jan. 2003). "Design Guidelines for Robust Internet Protocols". In: *SIGCOMM Comput. Commun. Rev.* 33.1.

Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Trans. on Dependable and Secure Computing* 1.1.

Steve Bishop, Matthew Fairbairn, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough (Draft). *Engineering with Logic: Rigorous Specification and*

Validation for TCP/IP and the Sockets API. URL: http://www.cl.cam.ac.uk/~pes20/Netsem/paper3.pdf.

Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough (Mar. 2005a). *TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification : Volume 1: Overview*. Tech. rep. UCAM-CL-TR-624. University of Cambridge, Computer Laboratory.

Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough (Mar. 2005b). *TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification : Volume 2: The Specification*. Tech. rep. UCAM-CL-TR-625. University of Cambridge, Computer Laboratory.

Olivier Bonaventure, Clarence Filsfils, and Pierre Francois (Oct. 2007). "Achieving Sub-50 Milliseconds Recovery Upon BGP Peering Link Failures". In: *IEEE/ACM Transactions on Networking* 15.5.

R. Braden (Oct. 1989). *Requirements for Internet hosts—Communication Layers*. RFC 1122.

R. Braden (July 1994). *T/TCP—TCP Extensions for Transactions: Functional Specification*. RFC 1644.

N. Burton-Krahn (2002). "HotSwap – Transparent Server Failover for Linux". In: *Proc. of USENIX LISA 2002: Sixteenth Systems Administration Conference*.

R. Bush and D. Meyer (Dec. 2002). *Some Internet Architectural Guidelines and Philosophy*. RFC 3439.

B. Carpenter (June 1996). *Architectural Principles of the Internet*. RFC 1958.

Tushar Deepak Chandra and Sam Toueg (Mar. 1996). "Unreliable failure detectors for reliable distributed systems". In: *J. ACM* 43.2, pp. 225–267.

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield (May 2005). "Live Migration of Virtual Machines". In: *Proceedings of the USENIX Symposium on Networked Systems and Implementation*. Boston, MA.

David D. Clark (1988). “The Design Philosophy of the DARPA Internet Protocols”. In: *SIGCOMM*.

David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden (June 2005). “Tussle in cyberspace: defining tomorrow’s internet”. In: *IEEE/ACM Trans. Netw.* 13.3.

Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield (Apr. 2008). “Remus: High Availability via Asynchronous Virtual Machine Replication”. In: *Proceedings of the USENIX Symposium on Networked Systems and Implementation*. San Francisco, CA.

Edsger W. Dijkstra (May 1976). *The effective arrangement of logical systems*. EWD 562.

Edsger W. Dijkstra (1982). “On the role of scientific thought”. In: *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, pp. 60–66.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson (Apr. 1985). “Impossibility of distributed concensus with one faulty process”. In: *Journal of the ACM* 32.2.

Mohammed G. Gouda and Chung-Kuo Chang (Jan. 1986). “Proving Liveness for Networks of Communicating Finite State Machines”. In: *ACM Trans. Program. Lang. Syst.* 8.1, pp. 154–180.

Joshua D. Guttman and Dale M. Johnson (Oct. 1994). “Three applications of formal methods at MITRE”. In: *Formal Methods Europe*.

Bing Han (Dec. 2004). “Formal specification of the TCP service and verification of TCP connection management”. PhD thesis. University of South Australia.

Y. Huang and C. M. R. Kintala (June 1993). “Software Implemented Fault Tolerance: Technologies and Experience”. In: *Proceedings of 23rd International Symposium on Fault Tolerant Computing (FTCS-23)*, pp. 2–9.

V. Jacobson, R. Braden, and D. Borman (May 1992). *TCP Extensions for High Performance*. RFC 1323.

P. Jakama (Nov. 2008). *Revised Default Values for the BGP ‘Minimum Route Advertisement Interval’*. Internet-Draft `draft-jakama-mrai-02`.

C. B. Jones (Oct. 1983). “Tentative Steps Toward a Development Method for Interfering Programs”. In: *ACM Trans. Program. Lang. Syst.* 5.4.

E. Keller, Jennifer Rexford, and J. van der Merwe (Apr. 2010). “Seamless BGP Migration with Router Grafting”. In: *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*. San Jose, CA.

Eric Keller, Minlan Yu, Matthew Caesar, and Jennifer Rexford (2009). “Virtually eliminating router bugs”. In: *CoNEXT*.

J. Kempf and R. Austein (Mar. 2004). *The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture*. RFC 3724.

R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith (2003). “Transparent TCP Connection Failover”. In: *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)*. IEEE Computer Society, pp. 383–392. ISBN: 0-7695-1952-0.

C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian (2000). “Delayed Internet Routing Convergence”. In: *Proc. ACM SIGCOMM*, pp. 175–187.

Leslie Lamport (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Wyatt Lloyd and Michael J. Freedman (2011). “Coercing clients into facilitating failover for object delivery”. In: *DSN*, pp. 157–168.

M.-Y. Luo and C.-S. Yang (2001). “Constructing Zero-Loss Web Services”. In: *Proceedings of IEEE INFOCOM*. Anchorage, AK, pp. 1781–1790.

Nancy Lynch and Frits Vaandrager (1995). “Forward and Backward Simulations Part I: Untimed Systems”. In: *Information and Computation* 121.

M. Marwah, S. Mishra, and C. Fetzer (June 2003). “TCP Server Fault Tolerance Using Connection Migration to a Backup Server”. In: *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)*. San Francisco, CA: IEEE Computer Society.

M. Marwah, S. Mishra, and C. Fetzer (June 2005). “A System Demonstration of ST-TCP”. In: *Proc. of the 2005 IEEE International Conference on Dependable Systems and Networks (DSN 2005)*. Yokohama, Japan: IEEE Computer Society.

M. Marwah, S. Mishra, and C. Fetzer (2006). “Fault-tolerant and scalable TCP splice and web server architecture”. In: *Proc. of the 25th Symposium on Reliability in Distributed Software (SRDS'06)*. IEEE Computer Society, pp. 301–310. ISBN: 0-7695-2677-2.

M. Marwah, S. Mishra, and C. Fetzer (2008). “Enhanced server fault-tolerance for improved user experience”. In: *Proc. of the International Conference on Dependable Systems and Networks (DSN'08)*. IEEE Computer Society, pp. 167–176.

M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow (Oct. 1996). *TCP Selective Acknowledgment Options*. RFC 2018.

M. Orgiyan and C. Fetzer (2001). “Tapping TCP Streams”. In: *Proc. of the International Symposium on Network Computing and Applications (NCA'01)*. IEEE Computer Society, pp. 278–289. ISBN: 0-7695-1432-4.

Information technology—Open Systems Interconnection—Basic Reference Model: The Basic Model (1994). ISO/IEC 7498-1, ITU-T Recommendation X.200.

J. Paris, A. Valderruten, and V. M. Gulias (Oct. 2005). “Developing a functional TCP/IP stack oriented towards TCP connection replication”. In: *Proceedings of the 3rd International IFIP/ACM Latin American conference on Networking (LANC'05)*. Cali, Columbia.

D. L. Parnas (1972). “On the criteria to be used in decomposing systems into modules”. In: *Commun. ACM* 15.12.

D. Pei, B. Zhang, D. Massey, and L. Zhang (Feb. 2006). “An analysis of convergence delay in path vector routing protocols”. In: *Computer Networks: The International Journal of Computer and Telecommunications Networking* 50.3.

D. Pei, X. Zhao, D. Massey, and L. Zhang (2004). “A study of BGP path vector route looping behavior”. In: *Proceedings of the 24th International Conference on Distributed Computing Systems*, pp. 720–729.

Transmission Control Protocol (Sept. 1981). RFC 793. Information Sciences Institute.

Thomas Ridge, Michael Norrish, and Peter Sewell (Feb. 2009). *TCP, UDP, and Sockets: Volume 3: The Service-level Specification*. Tech. rep. UCAM-CL-TR-742. University of Cambridge, Computer Laboratory.

A. Sahoo, K. Kant, and P. Mohapatra (June 2006). "Characterization of BGP Recovery Time under Large-Scale Failures". In: *IEEE International Conference on Communications (ICC'06)*, pp. 949–954.

J. H. Saltzer, D. P. Reed, and David D. Clark (1984). "End-to-End Arguments in System Design". In: *ToCS* 2.4.

S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter (Jan. 2007). *Graceful Restart Mechanism for BGP*. RFC 4724.

Claude Shannon (1948). "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3.

Z. Shao, H. Jin, and J. Wu (Nov. 2006). "AR-TCP: Actively Replicated TCP Connections for Cluster of Workstations". In: *Workshop on Frontier of Computer Science and Technology (FCST '06)*. Fukushima, Japan, pp. 3–10.

G. Shenoy, S. K. Satapati, and R. Bettati (2000). "HydraNet-FT: Network Support for Dependable Services". In: *Proc. of the International Conference on Distributed Computing Systems (ICDCS'00)*, pp. 699–706.

Mark Anthony Shawn Smith (Sept. 1997). "Formal Verification of TCP and T/TCP". PhD thesis. Massachussets Institute of Technology.

Mark A. Smith and K. K. Ramakrishnan (Apr. 2002). "Formal Specification and Verification of Safety and Performance of TCP Selective Acknowledgment". In: *IEEE/ACM Trans. Netw.* 10.2, pp. 193–207. ISSN: 1063-6692. URL: http://dl.acm.org/citation.cfm?id=508325.508329.

A. C. Snoeren, D. G. Andersen, and H. Balakrishnan (2001). “Fine-Grained Failover Using Connection Migration”. In: *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*. San Francisco, CA, pp. 221–232.

Florin Sultan, Aniruddha Bohra, Stephen Smaldone, Yufei Pan, Pascal Gallard, Iulian Neamtiu, and Liviu Iftode (2005). “Recovering Internet Service Sessions from Operating System Failures”. In: *IEEE Internet Computing* 9.2, pp. 17–27.

F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode (July 2002). “Migratory TCP: Connection Migration for Service Continuity over the Internet”. In: *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS ’02)*.

David L. Tennenhouse (1989). “Layered Multiplexing Considered Harmful”. In: *International Workshop on High Speed Networking*.

Paulo Veríssimo (2006). “Travelling through Wormholes: a new look at Distributed Systems Models”. In: *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)* 37.1.

Giovanni Vigna (2003). “A topological characterization of TCP/IP security”. In: *Formal Methods Europe*.

Raio Y. Zaghal and Javed I. Khan (2005). *EFSM/SDL modeling of the original TCP standard (RFC 793) and the congestion control mechanism of TCP Reno*. Tech. rep.

D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud (2003). “Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP”. In: *Proc. of the International Conference on Dependable Systems and Networks (DSN’03)*. Los Alamitos, CA, USA: IEEE Computer Society. ISBN: 0-7695-1952-0.

D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud (2009). “Practical and low-overhead masking of failures of TCP-based servers”. In: *Transactions on Computer Systems* 27.2.

V. C. Zandy and B. P. Miller (2002). “Reliable Network Connections”. In: *Proceedings of ACM MobiCom*. Atlanta, GA, pp. 95–106.

R. Zhang, T. Abdelzaher, and J. Stankovic (2004). “Efficient TCP Connection Failover in Server Clusters”. In: *Proceedings of IEEE INFOCOM*. Hong Kong, pp. 1219–1228.

X. Zhao, D. Pei, D. Massey, and L. Zhang (Oct. 2003). “A study on the routing convergence of Latin American networks”. In: *LANC 2003*. La Paz, Bolivia.