Chapter 2

Background

In order to provide a technical background for the subsequent three chapters we provide a brief overview of the following: (a) $\pi$-calculus and the Z specification language along with related research; (b) formal verification techniques; and (c) denial-of-service attacks and quality-of-service in protocol stacks.

2.1 $\pi$-calculus

The $\pi$-calculus [60] is a process calculus having communication and mobility as primitive notions. The syntax for the polyadic $\pi$-calculus [59] (Figure 2.1) permits a finite list of values to be communicated via channels. Summation is included in the syntax although it is possible to encode guarded forms of summation in a summation-free calculus [62]. Without

\[
\begin{align*}
P,Q &:= \bar{u}(\bar{w}).P & \text{Output action on channel } u; \\
| & \quad u(\bar{z}).P & \text{Input action on channel } u; \\
| & \quad P \mid Q & \text{Parallel composition} \\
| & \quad P + Q & \text{Summation} \\
| & \quad \nu n P & \text{Name generation and restriction} \\
| & \quad \text{if } u = v \text{ then } P \text{ else } Q & \text{Conditional} \\
| & \quad \text{recursion variable} \\
| & \quad \mu x.P & \text{Recursion} \\
| & \quad 0 & \text{Empty process}
\end{align*}
\]

Figure 2.1: $\pi$-calculus: Processes
loss of expressive power, the syntax uses an explicit recursion operator $\mu$ instead of replication [7]. A replicated process $!P$ may be represented via recursion as $\mu x. (P \mid x)$. As discussed later, recursion helps avoid several additional labelled transition rules needed to deal with process interactions occurring within replication. Moreover, correspondence between reduction and labelled $\tau$ transitions becomes simpler, and an important sub-class of the $\pi$-calculus – the finite control $\pi$-calculus [23] – is defined in terms of recursion. Several authors have used recursion instead of replication, either using recursion variables or using recursive parametric process definitions [77, 5, 44].

### 2.1.1 Operational Semantics

Operational semantics of $\pi$-calculus terms are given as either a collection of inference rules for performing reductions (unlabelled transitions) or a collection of inference rules for performing labelled transitions. In order to break the dependence of reduction rules on the rigid process structure, e.g., to allow non-adjacent terms to interact with each other, reductions are defined up to structural equivalence – an equivalence relation on processes. Structural equivalence rules are given in Figure 2.2. Apart from these rules, the rules of reflexivity, transitivity, and symmetry also apply to structural equivalence, as it is an equivalence relation.

Figure 2.3 gives the reduction rules [80] for $\pi$-calculus. The rule $R$-INTER is the communication rule, allowing input and output terms to interact. Thanks to the $R$-STRUCT rules the interacting terms need not be adjacent to each other. The $R$-REC rule unwinds the recursion; $R$-TAU removes a $\tau$ prefix; and $R$-IF as well as $R$-ELSE rules perform conditional transitions.
Figure 2.3: Reduction rules for $\pi$-calculus

Figure 2.4 shows labelled transition rules [80] for the $\pi$-calculus. Unlike reductions which are defined using an inductively defined reduction relation and the set of algebraic equivalence laws defined by the structural equivalence rules, labelled transitions are defined compositionally using a single inductive transition relation. The $L$-IF, $L$-ELSE, $L$-TAU and $L$-REC rules are the labelled equivalents of the corresponding unlabelled rules $R$-IF, $R$-ELSE, $R$-TAU, and $R$-REC, respectively. Application of the rule L-OUT results in an output action. The L-IN rule performs an input action, generating transitions labelled with all possible input values. This style of input transitions, wherein values are instantiated along with the input transitions, are called early transitions [80]. The $L$-OPEN rule allows restricted names to extrude their scope – an essential feature required for modeling dynamic systems. The $L$-PAR and $L$-SUM rules allow transition to occur within terms under parallel composition and summation, respectively. Finally, the $L$-COM rule allows input and output terms to interact, enabling communication. When the list of extruded names in the $L$-COM rule is non-empty it effectively behaves like the $L$-CLOSE rule found in [80].
Figure 2.4: Labelled transition rules for $\pi$-calculus
There have been a number of extensions to the $\pi$-calculus, but they do not address mobility with stateful channels. The applied $\pi$-calculus [2] allows communication of data items that are specified algebraically, i.e., as terms under equational theories. The language pi-F [94] is an extension of $\pi$-calculus with explicit fusions, and CC-Pi [16] extends pi-F with concurrent constrained programming. Both CC-Pi and $\pi Z$ have some notion of state-based blocking: in CC-Pi $ask$ and $tell$ actions may block based on the state of the constraint store, whereas in $\pi Z$ an input or output action would block based on channel state if the operation pre-condition is not satisfied.

Polyadic synchronization is investigated in [17], where the synchronization of two processes on a finite list of channels is discussed. It would be interesting to investigate polyadic synchronization on stateful channels offered by $\pi Z$. Many such extensions may be expressed as Psi-calculi [11] using nominal data terms, assertions about data, and generalized form of agents. In $\pi Z$, a transition involving a state-holding term is based upon an action obligation, which is a pair of input-output actions that the environment must fulfil. This is a key notion that we introduce. It is absent in other $\pi$-calculi, including the Psi-calculus, and is the main reason why $\pi Z$ and its labelled transitions can’t be formulated in the Psi-calculus framework.

A method of combining CSP and the B-method is examined in [91]. A combination of the $\pi$-calculus and the B method is proposed in [47]. Circus [95] is a specification language that combines Z, CSP, and imperative constructs.

The principal difference between these approaches and our work on the specification language $\pi Z$ is that we examine integration of Z abstract types in the setting of mobile channels offered by the $\pi$-calculus. Our usage of abstract types as stateful channels is novel and is useful for specifying durable communication activities occurring in mobile systems. Moreover, working within the $\pi$-calculus framework allows us to rigorously explore important properties such as the bisimilarity.

### 2.2 Z language

Z is a formal specification language based on first-order logic and axiomatic set theory [96]. We may define stateful abstract types in Z using state and operation schema constructs.
The state of an abstract type is specified using state schema and operations on the abstract type are specified using operation schemas. Figure 2.5 illustrates the Z syntax using abstract type definition of bounded buffer that queues data items. The construct Buffer[D] is the state schema parameterized by item type D, defining state components items and size. The condition #items \leq size is the state invariant. Operation schema definitions for put and get operations are also parameterized by item type D. Primed components in operation schemas represent after-state and unprimed components represent before-state. The notation ΔBuffer[D] is a shortcut for introducing primed and unprimed state components into the operation schema definition. An operation’s input parameters are suffixed with ‘?’ and output results are suffixed with ‘!’.

The state schema definition of a Z abstract type is interpreted as the set of all bindings (labelled tuples) that satisfy the schema definition (Definition 1).

**Definition 1** (Interpretation of state schema).

Given a state schema definition s (of some abstract type) having components \(\tilde{l} (l_1, ..., l_n)\) with types \(L_1, ..., L_n\) respectively, its interpretation \(I(s)\) is given by the set of labelled tuples that satisfy:

\[
I(s) = \{(l_1 \mapsto v_1, ..., l_n \mapsto v_n) \mid s\{\tilde{v}/\tilde{l}\}\}
\]

where \(v_i \in L_i\).

Intuitively, an operation schema relates the operation’s parameter values and before-state to result values and after-state. Therefore, an operation schema is interpreted as a relation on the sets of bindings defined by the state schema as well as the sets of parameter and result values (Definition 2).

**Definition 2** (Interpretation of operation schema).

Given an operation schema \(p\) on state schema \(s\), with inputs \(\tilde{i} (i_1, ..., i_k)\) and respective types \(I_1, ..., I_k\), and outputs \(\tilde{o} (o_1, ..., o_m)\) and respective types \(O_1, ..., O_m\), its interpretation \(I(p)\) is given by the relation:

\[
I(p) = \{(W, \tilde{u}, W', \tilde{v}) \mid p\{\tilde{w}/\tilde{l}, \tilde{u}/\tilde{i}, \tilde{w}'/\tilde{l}', \tilde{v}/\tilde{o}\}\}
\]
Figure 2.5: Z abstract type definition of bounded buffer.
where $W = (l_1 \mapsto w_1, \ldots, l_m \mapsto w_m) \in \mathcal{I}(s)$, $W' = (l_1 \mapsto w'_1, \ldots, l_m \mapsto w'_m) \in \mathcal{I}(s)$, $u_i \in I_i$ and $o_i \in O_i$.

Note that an operation is not defined if before-state violates the state invariant, i.e., the before-state $W \notin \mathcal{I}(s)$, where $s$ is the operation’s state schema. In terms of the relational interpretation, an operation $p$’s pre-condition is the set of parameter values and before-states for which there exists some after-state and result values as per the relation $\mathcal{I}(p)$. Naturally, in order to perform an operation, the operation’s parameter values and before-state must satisfy the operation’s pre-condition.

These interpretations are consistent with Z schema interpretations found in the literature, e.g., [96]. For simplicity of presentation, definitions 1 and 2 assume that other Z symbols such as axiomatically-defined functions do not appear in schema definitions. If these symbols do appear, they must be replaced with their set-oriented interpretations to obtain a fully relational interpretation of state and operation schemas. A complete denotational semantics for Z specifications is given in [45].

The use of abstract types in concurrent systems without channel mobility is well-known – Ada tasking being a prime example [93]. These approaches essentially view instances of concurrent abstract types as processes (or tasks) having internal state, capable of performing several operations. CSP semantics for Z is given in [73], wherein Z abstract types are interpreted as CSP processes. ZCCS [34] proposes a combination of Z and CCS by using Z for specifying data items carried by CCS channels.

CSP-OZ [29] allows CSP processes to be embedded within Object-Z [86] classes, enabling specification of systems of concurrent objects. CCZ [35] allows Z abstract type operations to be invoked directly from within CSP expressions. A combination of Object-Z and timed CSP is proposed in [56]. They view operations as terminating CSP processes, associating channels (with time constraints) for individual operation parameters. A comparison of various approaches of combining Z with value-passing process algebra may be found in [30]. Mobile-OZ [88] attempts to bring mobility primitives into the Object-Z language. Objects represent agents, which may move between locations by use of mobility primitives.
2.2.1 Data Refinement

Typically, when we construct software systems, we start with a high-level specification of the system and then refine the specification with more implementation details, till we finally reach the desired concrete program [3]. A high-level specification may be thought of as an abstract program, in which many implementation details have been omitted. We may construct specifications and programs from abstract datatypes. The key question in this context is, given two specifications (programs) when can we say that one of them is a correct refinement of the other. Since programs are constructed from abstract types, the notion of program refinement is closely connected to the notion of data refinement.

We first introduce a formalized notion of program refinement [26]1. Consider a program $P(A)$ built from an abstract type $A$ and a program $P(C)$ built from a more concrete type $C$. Let $AI$ ($CI$) be the initialization operation that maps input (observable) values into internal representation of $A$ ($C$). Similarly, let $AF$ ($CF$) be the finalization operation that maps internal representation of $A$ ($C$) back to output (observable) values. We say that $P(C)$ refines $P(A)$ if input-output observations from $C$ are consistent that from $A$. This is formalized by relational inclusion as given in Figure 2.6.

![Figure 2.6: Program refinement.](image)

An abstract type $A$ is defined by the tuple $(AI, A_j \in J, AF)$, where $AI$ ($AF$) is the initialization (finalization) operation that maps the initial input values (internal representation of $A$) to the internal representation of $A$ (final output values). We say that two abstract types $C$ and $A$ are compatible if they share the same sets of initial input values and final output values as well as same indexing set for the set of operations. In other words, for every operation $A_j$ of $A$ there is a unique corresponding operation $C_j$ of $C$.

---

1 The primary source texts for this section are [26] and [96].
We say that an abstract type $C$ refines another abstract type $A$, if for all corresponding pairs of programs $(P(C), P(A))$ that can be built from $C$ and $A$, $P(C)$ refines $P(A)$, wherein $P(C)$ is obtained by replacing $A$’s operations in $P(A)$ with the corresponding operation of $C$.

Naturally, the above criterion for data refinement is not an effective procedure since we can construct infinitely many programs using abstract types. We need a local criterion that can look at individual operations of $A$ and $C$ and then decide whether $C$ does refine $A$. In order to do so, we first define an abstraction relation (called the retrieve relation [96]) that relates concrete internal values of a $C$ with the internal values of $A$. The condition for refinement of abstract types is that the diagram given in Figure 2.7 must weakly commute, i.e., images under arrow compositions are related by set inclusion rather than set equality.

![Diagram](image)

Figure 2.7: Simulation of abstract types.

There are different ways in which the conditions for simulation could be specified based on the idea of weak commutativity of the diagram in Figure 2.7. $C$ $L$-simulates $A$ w.r.t $\alpha$ if and only if [26]:

1. $CI \subseteq AI; \alpha^{-1}$

2. for all $j \in J$: $\alpha^{-1}; C_j \subseteq A_j; \alpha^{-1}$

3. $\alpha^{-1}; CF \subseteq AF$

$C$ $L^{-1}$-simulates $A$ w.r.t $\alpha$ if and only if [26]:

1. $CI; \alpha \subseteq AI$

2. for all $j \in J$: $C_j; \alpha \subseteq \alpha; A_j$

3. $CF \subseteq \alpha; AF$
L-simulation is also called downward simulation and L\(^{-1}\)-simulation is also called upward simulation. Although both L-simulation and L\(^{-1}\)-simulation are sound criteria for data refinement they are not complete. One way to achieve completeness is to use a combination of both (see [26]).

We may now go onto a syntactic characterization of data refinement. In order to show that a Z abstract type \(C\) is a correct refinement of another Z abstract type \(A\), we first define a retrieve relation \(R\) relating their state components, and then prove the following refinement conditions [75]:

1. If CI and AI are concrete and abstract initialization schema, then: \(\forall CI \cdot \exists AI \cdot R\).
2. The concrete operation can be invoked whenever abstract operation can: \(\forall R]\pre AO \cdot pre CO\).
3. When abstract operation can be invoked, it will give a result consistent with the concrete operation: \(\forall R; CO]\pre AO \cdot \exists A' \cdot AO \land R'\)

### 2.3 Formal Verification

Formal verification techniques can be broadly categorized into two major approaches: model-checking and theorem-proving [39]. Model checking [20] is an algorithmic decision procedure that checks whether a given system model, typically in the form of a transition system, satisfies a temporal logic formula. There are broadly two classes of temporal logics: linear time and branching time [51]. Computational tree logic (CTL) is one of the popular branching time temporal logics [18]. We briefly examine this logic here.

Given a set of atomic propositions \(AP\), the formal syntax of CTL formulae is given below.

- Every atomic proposition \(p \in AP\) is a CTL formula.
- if \(\phi_1\) and \(\phi_2\) are CTL formulae, then so are: \(\neg \phi_1, \phi_1 \land \phi_2, \phi_1 \lor \phi_2, \phi_1 \implies \phi_2, \mathbb{P}\phi_1, \mathbb{P}(\phi_1 \cup \phi_2)\), where \(\mathbb{P} \in \{\mathbb{A}, \mathbb{E}\}\) and \(S \in \{\mathbb{F}, \mathbb{G}, \mathbb{X}\}\).

The logical operators (\(\neg, \lor, \land, \implies\)) have their usual meaning. The operators \(\mathbb{A}\) and \(\mathbb{E}\) are path quantifiers, where \(\mathbb{A}\) read as: for all paths and \(\mathbb{E}\) reads as: there exists a path.
Following are called the state operators: (a) $\mathcal{F}$: sometime in the future, (b) $\mathcal{G}$: globally, (c) $\mathcal{X}$: in the next state, and (d) $\mathcal{U}$: until.

Formal semantics for CTL is defined with respect to labelled transition structures, formally defined by the triple: $\mathcal{M} = (S, T, L)$, where:

1. $S$ is a finite set of states.
2. $T$ is a total binary relation on $S$ ($T \subseteq S \times S$). $T$ is called the transition relation.
3. $L$ is the labelling function that assigns atomic propositions to states. $L : S \rightarrow 2^{AP}$.

CTL formulae are interpreted using the computation tree that we obtain by unwinding the transition structure $T$. A path in this computation tree is an infinite sequence of states, $(s_0, s_1, s_2, \ldots)$ such that $(\forall i) s_i T s_{i+1}$. The notation $\mathcal{M}, s_0 \models \phi$ to denote that the formula $\phi$ holds at state $s_0$ in the structure $\mathcal{M}$. The relation $\models$ is inductively defined as:

- $s_0 \models p$ iff $p \in L(s_0)$
- $s_0 \models \neg \phi$ iff not: $s_0 \models \phi$
- $s_0 \models \phi_1 \land \phi_2$ iff $s_0 \models \phi_1$ and $s_0 \models \phi_2$
- $s_0 \models \phi_1 \lor \phi_2$ iff $s_0 \models \phi_1$ or $s_0 \models \phi_2$
- $s_0 \models \mathcal{AX} \phi$ iff for all states $s'$ such that $s_0 T s'$, $s' \models \phi$
- $s_0 \models \mathcal{EX} \phi$ iff there exists a state $s'$ such that $s_0 T s'$, $s' \models \phi$
- $s_0 \models \mathcal{A}(\phi_1 \lor \phi_2)$ iff for all paths $(s_0, s_1, \ldots)$ $(\exists i \geq 0) s_i \models \phi_2 \land (\forall j : 0 \leq j < i) s_j \models \phi_1$
- $s_0 \models \mathcal{E}(\phi_1 \lor \phi_2)$ iff for some path $(s_0, s_1, \ldots)$ $(\exists i \geq 0) s_i \models \phi_2 \land (\forall j : 0 \leq j < i) s_j \models \phi_1$

The rest of the connectives may be expressed in terms of the above basic connectives using the following logical equivalences:

- $\phi_1 \implies \phi_2 \iff \neg \phi_1 \lor \phi_2$
- $\mathcal{AF} \phi \iff \mathcal{A}(\mathcal{T} \mathcal{U} \phi)$. We say that $\phi$ is inevitable.
- $\mathcal{EF} \phi \iff \mathcal{E}(\mathcal{T} \mathcal{U} \phi)$. We say that $\phi$ is possible.
- $\mathcal{EG} \phi \iff \neg \mathcal{AF} \neg \phi$. There exists a path where $\phi$ holds at every state.
- $\mathcal{AG} \phi \iff \neg \mathcal{EF} \neg \phi$. We say that $\phi$ holds globally.

Unlike CTL, which is a branching-time temporal logic, LTL is a linear-time temporal logic. There is an implicit universal quantification over all paths in LTL. We cannot make
statements like ‘possibly $F$’ in LTL since the statement requires existential quantification over paths. SPIN [10] is classic example of a model checking tool for LTL. SPIN models are developed in a custom language called Promela. SPIN has been employed in verification of protocol specifications. Promela++ [9] is a language based on Promela that allows construction of protocol code from verifiable specifications. It allows model checking of only control constructs and embeds ‘C’ code that is hidden from the verifier. Apart from the fact that they employ a special-purpose language and their approach does not allow verification of the embedded ‘C’ code, it is not clear whether this method would scale for large protocol stacks. Model checkers [99] exist for verifying $\pi$-calculus models, making it an attractive choice for modeling and verifying mobile systems.

Horus and its related Ensemble project [15] focus on building group communication protocol stacks using a component-based approach. Horus components were built in ‘C’ whereas Ensemble components were built in OCaml and could be formally verified using NuPRL logic system. Their objective was to build robust group communication systems, and do not deal with low-level aspects of IP packet processing which form the core of internet protocol stacks. Our split verification approach allows construction of complete internet protocol stacks using a popular general-purpose language such as C++ and permits verification of all aspects of the stack in a scalable manner.

The primary limitation of model checking is the state explosion problem [54]. Significant research has gone into mitigating this problem. Symbolic model checking using binary decision diagrams (BDD) offers a more efficient procedure when compared with explicit model checking. Abstraction and compositional verification [54] can be employed to reduce the number of states that must be model-checked. Assume-guarantee reasoning introduced by Pnueli [66] is a powerful compositional reasoning that could be applied to state transition systems that are composed in serial or parallel, in order to reduce the complexity of model-checking. Another complexity-reducing technique in model-checking is to exploit symmetry [19] in the finite state model. We employ a combination of theorem-proving and model-checking in a novel manner to handle verification of complex protocol stacks.

Theorem proving may be done using a formal language such as Z [87] or using more general-purpose logic systems such as higher order logics (HOL). The formal specification
language Z has been used for verification of a number of relatively large problems [32] [31]. We use Z as our proof language. A number of interactive theorem provers are available that can partially automate the theorem proving task. HOL-Z [50] is an effort to embed Z in higher order logic system. We used Z/Eves interactive proof assistant [57] for the present work.

There have been various attempts to combine model-checking and theorem proving. Predicate abstraction [24] enables creation of an abstract model (with a small number of states) of a concrete system specification by using abstraction predicates. Another direction of research has been to translate theorem-proving problem to model-checking problem. For example, ZSAL [85] translates Z specifications that have been traditionally verified using theorem-proving techniques to SAL which can be verified using model-checking. Symbolic model prover [13] defines a proof system for first-order branching time μ-calculus, which allows theorem-proving to be used in combination with model-checking. We combine theorem-proving and model-checking in a novel manner, which we call split verification in which primitive operation-level properties are established via theorem-proving and are promoted to stack-level primary operations via model-checking on an abstract operations model.

PVS [64][69] provides a powerful integrated environment for constructing specifications in higher-order logic system and performing type-checking, theorem-proving, and executing various decision procedures including model-checking. Working in the PVS environment would allow us to perform operation-level theorem-proving and stack-level model-checking in a single integrated framework.

The work on Java PathFinder (JPF) [92] pioneered research in applying formal verification techniques based on model-checking in a modern programming language setting (Java). Hatcliff et al [41] proposed techniques such as program slicing to reduce model-checking complexity for large programs. Several researchers have also worked on formalizing semantics of imperative languages such as C and C++ using higher-order logic (HOL) [63]. This allows partially automated application of theorem-proving techniques to prove properties about imperative programs. We use C++ as the language for stack construction and use novel split verification approach to deal with program complexity.

Ridge, Norrish, and Sewell [72] have constructed formal specification of TCP protocol
in HOL, which may be used to verify conformance of protocol stacks. They use their specification to verify conformance of actual protocol stacks by capturing experimental traces. Our goal is to construct a formally verified protocol stack, rather than protocol specification. Their protocol specifications may be translated to Z and used as abstract specifications to be used in our proof-by-refinement approach.

Two recent efforts in formal verification of large-scale software are: (a) formally verified CompCert compiler by Leroy [53]; and (b) seL4 micro-kernel verification of Klein et al. [49]. CompCert translates programs in Clight language (a large subset of C) to a subset of PowerPC assembly language (called PPC in Leroy’s paper), through 8 intermediate languages. Formal operational semantics is defined for the source, target, and all the intermediate languages. The compiler has been proven to possess the semantic preservation property, i.e., the semantics of source program is preserved in the target program. The verified part of the compiler, which translates from Clight abstract syntax to PPC abstract syntax, is written in the pure functional specification language of Coq proof assistant.

The purpose of our split-verification approach is to establish properties for a program constructed in an imperative high-level language (we use imperative subset of C++). From an application viewpoint, our work is complementary to Leroy’s work. Once properties have been established for the source program, if the program is passed to a compiler such as CompCert that preserves the semantics, then we have the additional guarantee that the executable program also satisfies these properties. From a verification viewpoint, CompCert verification is performed in a pure functional setting of the Coq’s specification language, whereas the split verification is intended for a large-scale, imperative setting.

The seL4 verification employs three levels of refinement: (a) abstract specification in Isabelle/HOL, (b) executable specification in Isabelle/HOL generated from a Haskell prototype, and, (c) C implementation translated to Isabelle/HOL based on formal semantics defined for a large subset of C. Again, from an application viewpoint, a verified protocol stack is complementary to a verified micro-kernel towards the goal of constructing a fully verified system. The primary goal of the seL4 verification effort is to prove that executable specification is a correct refinement of abstract specification and that the C implementation is a correct refinement of the executable specification. In order to prove this, seL4 relies
on the notion of forward simulation between concrete and abstract state machines. We also rely on proving refinement theorems in Z for forward simulation to establish operation-level properties with respect to abstract operation specifications. The primary difference is that we employ refinement theorem proving only at operation-level and use model checking to promote these properties to stack-level, whereas seL4 employs only theorem proving.

2.4 Protocol Stacks: Performance, QoS, and DoS attacks

Performance as measured in terms of quality-of-service is of critical importance to real-time protocol stacks. An attacker may attempt to degrade quality-of-service by launching a denial-of-service type of attack. The Internet research community has known and has studied various forms of DoS attacks and defenses for more than a decade. Errin Fulp and his colleagues [33] examined denial-of-service (DoS) attacks on real-time media that target the quality-of-service (QoS) network infrastructure and suggested countermeasures. We examine a specific class of DoS attacks, indirect contention-in-hosts (ICiH) on real-time media that target end hosts.

Stanislav Shalunov and Benjamin Teitelbaum [83] argue that protection from worst-case DoS attack scenarios is a defining characteristic of network QoS services. When evaluating QoS pathway, one can also consider the ICiH attack scenarios that we present in this section. Clay Shields [84] formally defines network DoS (NDoS) attacks and proposes a taxonomy for them. The attack scenarios that we describe fit into this taxonomy, although Shields doesn't consider them specifically.

Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao [65] provide a survey of network-based defense mechanisms that mitigate DoS attacks. It includes protocol-based attacks such as TCP SYN (synchronize) flood and ICMP (internet control message protocol) flood. However, they don't consider ICiH. Sachin Garg, Navjot Singh, and Timothy Tsai [36] propose a scheme for enhancing tolerance of DoS attacks on Secure RTP (SRTP) based on first-level weak message authentication before the second-level strong authentication. Their countermeasures don't mitigate ICiH.

Vijay Gill [37] proposes priority queuing in routers so that high-priority legitimate packets
(for example, Border Gateway Protocol updates) are preferentially processed even when attack packets are present. We consider attacks that indirectly target RTMS (real-time media service) at end hosts rather than the routing infrastructure and develop a formal framework for analyzing such attacks. The mitigation principles we consider in the main text are based on threading models rather than priority queues. SYN-flood attacks [28] are classic examples of direct attacks on a service causing resource exhaustion for that service. One could use techniques such as SYN cookies to mitigate such attacks. We focus on ICiH, which indirectly degrades QoS metrics for real-time media rather than directly causing memory-resource exhaustion. Our study of performance under indirect-contention-in-hosts (ICiH) attacks and the analytic framework that we develop based on the notion of operation traces are novel.