Safe Compositional Specification of Networking Systems *

Azer Bestavros best@cs.bu.edu
Adam Bradley artdodge@cs.bu.edu
Assaf Kfoury kfoury@cs.bu.edu
Ibrahim Matta matta@cs.bu.edu

Computer Science Department
Boston University

ABSTRACT

The *science* of network service composition has emerged as one of the grand themes of networking research [17] as a direct result of the complexity and sophistication of emerging networked systems and applications. By "service composition" we mean that the performance and correctness properties local to the various constituent components of a service can be readily composed into global (end-to-end) properties without re-analyzing any of the constituent components in isolation, or as part of the whole composite service. The set of laws that govern such composition is what will constitute that new science of composition.

The heterogeneity and open nature of network systems make composition quite challenging, and thus programming network services has been largely inaccessible to the average user. We identify (and outline) a research agenda in which we aim to develop a *specification language* that is expressive enough to describe different components of a network service, and that will include *type hierarchies* inspired by type systems in general programming languages that enable the safe composition of software components. We envision this new science of composition to be built upon several theories, possibly including control theory, network calculus, scheduling theory, and game theory. In essence, different theories may provide different *languages* by which certain properties of system components can be expressed and composed into larger systems. We then seek to lift these lower-level specifications to a higher level by abstracting away details that are irrelevant for safe composition at the higher level, thus making theories scalable and useful to the average user. In this paper we focus on services built upon an overlay traffic management architecture, and we use control theory and QoS theory as example theories from which we lift up compositional specifications.

Keywords: Service Composition; Control Theory; QoS Theory; Type Systems

*This work was supported in part by NSF grants ITR ANI-0205294, ANI-0095988, ANI-9986397, and EIA-0202067.

1. INTRODUCTION

Specifying, designing, and developing correct, efficient, and resilient systems is a notoriously hard problem, particularly when placing these systems in open contexts in which they will interact with dynamic and unpredictable environments, peers, and adversaries. From convergence failure of the BGP routing protocol [6] to the interaction of congestion control/avoidance algorithms with each other to properties of peer-to-peer overlay networks and the applications they support [18, 19] to nascent architectures for sensor networks, questions of correctness in the face of the wide and deep unknowns of evolving network infrastructures and protocols remain among the most pressing challenges to networking research and development.

By "correctness" we mean simply that we can know with certainty some desirable invariants of a system based upon its specification or implementation. Many techniques are already available to describe, discuss, and deduce the invariants of a single software component: type systems, model checking, mathematical analyses and countless derivative tools allow us to speak confidently about many *local invariants*. While there are many interesting and useful properties of single software agents for which plausible verification systems do not yet exist (and may never exist), in principle we have a good handle on what invariant properties for single software components look like and how to go about establishing them. What we do not yet have is a solid grasp upon how to describe, discuss, and deduce the global invariants of open, extensible software systems, or how to (hopefully efficiently) bridge the gap between local and *global invariants*, where global invariants describe the acceptable range of behaviors and emergent properties when the components or agents making up a system interact.

A fundamental challenge in specifying an open system is to ensure that particular desirable global properties (*e.g.*, convergence upon a fair partitioning of bandwidth, absence of deadlock or livelock, finding a statistically useful estimate of sensor readings across an ad-hoc network, etc.) will necessarily arise from the verifiable properties of the individual parties to the system's execution. For example, consider AIMD (additive increase, multiplicative decrease) congestion control/avoidance in TCP; the original MIMD (multiplicative increase, multiplicative decrease) window control algorithm, correctly implemented at individual nodes of the system, failed to compose together with other instances of itself over network links in such a way that steady-state bandwidth would be fairly partitioned (a desirable global invariant).

This is not to suggest that there are no techniques for characterizing and analyzing composite systems in general. Far from it; such disciplines as control theory [15], network QoS

analysis [2, 11], scheduling [16], queuing theory, game theory, and our own CHAIN methodology [4] are rich in constructs and results well suited to exploring the properties of such systems, provided the questions are rightly framed. Consider, as an example, the application of control theory to inform the design of a new complex networked system. For the system's architect to take advantage of the proverbial "bag of control theoretic tricks" currently available, she must derive an abstract description of her system suitable for control theoretic analysis (i.e., in terms of transfer functions) which preserves the essential qualities of the original system (no small task, particularly for one unfamiliar with control theory's nuances), perform her analyses, and then re-iterate through a cycle of design and evaluation until a solution which is both practically workable and theoretically stable is found.

We envision a specification and development environment that is able to take results and contributions from valuable but less accessible (to the average programmer) approaches to composite system verification, and integrate their analyses automatically and mechanically into the design and implementation processes for composite and distributed applications. In essence, we seek to lower the bar of expertise required to take advantage of these tools by encouraging system designers and programmers to make intuitively clear claims about the behavior of individual component programs and desired invariants for the behavior of the complete system, automatically selecting appropriate algorithms for deducing whether stated local invariants hold, which other local invariants can be mechanically inferred, and whether these invariants are sufficient to deduce the veracity of invariants of the composite system as a whole.

Another important challenge that often hinders the use of disciplined approaches to the specification and development of systems in general—and of large-scale open networked systems such as those we envision in particular—is the issue of "scalability." Elegant techniques for reasoning about the correctness of large-scale software systems are often incapable of tracking any but the simplest "toy examples." That said, we note that successful specification and verification techniques are those that use domain-specific knowledge to reduce the size and complexity of the state space. This is precisely what motivates the approach we are proposing in this paper. Namely, we adopt a two-tiered approach. In the lower tier, we envision the use of an arsenal of domain-specific analytical tools from a variety of disciplines and theories to aid in establishing reduction rules (based on structure and relationships) for simple compositions. In the upper tier, we cast this knowledge in a type-theoretic framework which we use to reason about larger, more arbitrary compositions. This two-tiered approach was instrumental in allowing us to reason about arbitrary HTTP protocol compositions—and indeed to identify classes of configurations that are prone to deadlocks [3, 4]. One may view the lower-tier as providing an abstraction of the constituent subsystems—an abstraction in which properties that are "essential" for reasoning at the upper tier are preserved.

Furthermore, because lower-tier results are cast into a type-theoretic framework, we are able to capitalize upon the notion of a *subtype* to simplify the expression of lower-tier results and perhaps even the internal precision of the lower-tier mechanism. For example, while it may be possible for a control theoretic proof system to establish the precise time required for a system to converge upon a desired state, it is usually sufficient for our purposes to prove that the settle time belongs to some sufficiently descriptive member of a series of bounding classes ($t \leqslant 10ms$, $t \leqslant 20ms$, *etc*). In effect, we are replacing the "microscopic" view of system behaviors used within a particular discipline with a simply structured "macroscopic" descriptive view; this loss of precision introduces a small amount of "slack" into the system, but in so doing also allows us to use much more flexible, modular, and easily compared and integrated descriptions of component-wise and system behaviors.

The remainder of this paper is organized as follows. We begin in Section 2 by presenting a motivating application—that of building overlay traffic management services. We use our Internet Traffic Managers (ITM) architecture [1] and the eTCP tunneling service [7] as an example. Section 3 briefly introduces several well-established theoretical bases, namely control theory and QoS theory, for the analysis of composite systems such as overlays. Sections 4 and 5 outline the mapping of abstract theoretic models to type hierarchies and presents some of the type theoretic basis for the higher-level analysis of such systems.[1] Section 6 offers a survey of future directions for this work.

2. BUILDING BLOCKS FOR OVERLAY TRAFFIC MANAGEMENT

The scalability of the Internet (and any other large-scale extensible decentralized network) hinges upon our ability to tame the unpredictability associated with its open architecture. For example, the inherent burstiness of traffic at all layers of the architecture from link to network to transport to application leads to a dramatic trade-off between provisioning for peak demands and maximizing long-term utilization. This motivates exploration of basic control strategies for reducing traffic burstiness which can make this trade-off more manageable.

Such strategies could be implemented through services offered by an overlay management architecture. An example is our architecture of Internet Traffic Managers (ITMs)—special network elements strategically placed throughout the Internet (*e.g.*, in front of clients/servers or at exchange/peering points between administrative domains) [1]. We believe that the incorporation of such control functionalities will be key to the ability of the network infrastructure to sustain its own growth and to nurture the Quality-of-Service (QoS) needs of emerging applications by imposing a useful set of invariants upon traffic flowing into a network and (thereby) upon that traffic's behavior within the network itself. Such *safety* properties range from basics of protocol implementation and semantics to high-level statistical metrics of performance. Conceptually, this approach supplements the end-to-end Internet architecture with a composite edge-to-edge architecture where end-to-end services are assembled by composing controllers on the edges and boundaries of the Internet's constituent networks to act on the ends' behalf to exact some particular property from the network. The challenge in building ITMs is in ensuring they conform to their stated correctness conditions and performance goals, *i.e.*, ensuring that desirable global invariants are maintained by imposing local invariants upon the ITMs themselves.

[1] In a follow-up expanded report, we provide further details and evidence for several of our claims, as well as prove the soundness and completeness of the type system presented in Section 5.

2.1 eTCP: A Motivating Example

The best-effort nature of the Internet poses a significant obstacle to the deployment of many applications that require guaranteed bandwidth. Elastic TCP Tunneling (eTCP) [7] is a novel approach that enables two edge/border routers (ITMs) to use an adaptive number of TCP connections to set up a virtual tunnel maintaining the desired bandwidth between them. The number of TCP connections comprising this tunnel is elastic in the sense that it increases/decreases in tandem with competing cross traffic to maintain the targeted bandwidth level; the ingress and egress ITMs multiplex and demultiplex (respectively) over these TCP connections any packets belonging to the application requiring the bandwidth guarantee. Unlike many proposed solutions that aim to deliver soft QoS guarantees, the elastic-tunnel approach does not require any support from core routers (as with IntServ and DiffServ), is scalable in the sense that core routers maintain no per-flow state (as with IntServ), and is readily deployable in a variety of contexts, whether within a single ISP or across multiple ISPs.

The eTCP approach to delivering soft bandwidth guarantees between two points is to adaptively adjust the demand from the underlying best-effort network so as to match the requested QoS. We do so in a way that is consistent with the proper use of the network–namely, through the use of the Transmission Control Protocol (TCP) for bandwidth allocation. Specifically, to maintain guaranteed bandwidth between two points in the network, our approach calls for the establishment of an elastic tunnel between these points. An elastic tunnel is simply a set of TCP (or TCP-friendly) connections between two ITMs whose cardinality is dynamically adjusted in real-time so as to maintain a desirable target bandwidth.

Creating a suitable control function to drive the ordinality of the connection pool used to implement eTCP tunneling is no trivial problem. Our first instinct may be to implement a naïve controller which adjusts the width directly in proportion to the current width's under- or over-achievement of the targeted bandwidth, *i.e.*,

$$m(t+1) = \frac{B}{b(t)} m(t)$$

where $m(t)$ is the number of TCP connections at time t, B is the target bandwidth value, and $b(t)$ is the measured bandwidth achieved at time t. However, such a controller has a number of undesirable properties: it tends to react very aggressively to mismatches between the target and measured values, often introducing dramatic over-corrections and prolonged settling processes whenever the environment changes (including the startup transient).

Control theory offers us several more steady controllers with less overshoot and jitter, namely the **P** (proportional) and **PI** (proportional-integral) controls:

$$\textbf{P} \text{ control:} \quad m(t+1) = A_0(B - b(t))$$

$$\textbf{PI} \text{ control:} \quad m(t+1) = \sum_{i=0}^{\infty} A_i(B - b(t-i))$$

where A is a vector of weights (*gains*) determining how aggressively the controller should respond to the currently observed error (A_0, called α later in this paper) and compensate for the sum of previously observed errors ($A_1, A_2, \ldots$). As illustrated in Figure 1, **PI** is significantly more steady than the naïve controller, reducing burstiness experienced by routers along the tunnel's path, which *may* in turn have beneficial effects upon the ability of the network to predictably and stably meet its capacity demands.

2.2 eTCP: Towards Safe Implementation

eTCP tunnels are implemented using *itmBench*, a kernel- and user-space programming environment for ITMs [5]. In general, an itmBench service is a collection of event-handling functions which classify, monitor, process, and control packets and network abstractions (such as sockets and TCP connections) at several points in the packet-processing pipeline (pre-routing, application IP-in, IP-forward, application IP-out, and post-routing). For example, the eTCP service would comprise a classification function which would identify which packets should be scheduled over which eTCP tunnel (if any), a monitoring function which tracks the effective bandwidth being achieved by each tunnel, a processing function which does the actual packet scheduling for the tunnels (or preferentially drops packets), and a control function which adjusts the number of TCP connections within the tunnel.

While this architecture affords the programmer tremendous flexibility and power, it does not as yet afford us a great deal of analytical power with which to reason about the behavior of a single service or about the behavior of services when composed with one another. Is it "safe" for an itmBench programmer to replace single functions within a service, *e.g.*, replace a **PI** controller with a **PID** controller? How about to connect two eTCP tunnels end-to-end? Route the traffic making up one eTCP tunnel through another nested tunnel? What if other services (*e.g.*, DiffServ) are cascaded or nested within or without an eTCP tunnel?

Theories like control theory and QoS theory do afford us conceptual tools with which we can, by manual effort, begin to grapple with these questions. However, it requires the architect to bridge the chasm between the engineering details needed to develop an actual implementation (as sketched above) and the conceptual models needed to perform control- or QoS-theoretic evaluation of a system (sketched below). Our goal is to find ways to integrate these by taking advantage of natural subset and abstract limit descriptions available in the control and QoS theoretic spaces, as well as others.

3. THEORIES UNDERLYING COMPOSITIONAL ANALYSIS

Many conceptual tools exist for the analysis of the correctness of composite systems, or for understanding emergent properties thereof. While it is expected that network programmers would be well versed in the "art" of specifying and implementing specific functionalities (*e.g.*, programming a single **PI**-controlled eTCP tunnel between two ITMs), it is unrealistic to assume that they can (or should) master the analytical machinery (or theories) that enable the assessment of the composite system in which such functionalities are embedded (*e.g.*, an overlay of various **P**/**PI**-controlled eTCP tunnels between a large number of ITMs). That said, we believe that much of the benefits from using such theories could be still attained through proper support from the programming environment in which such services are developed. In this section we look at examples of theories that could be used towards that end. While we will devote much of our attention in this section (and indeed the whole paper) to the use of control theory as a tool for verifying the safety of a system specification, we do not believe that the techniques we describe are in

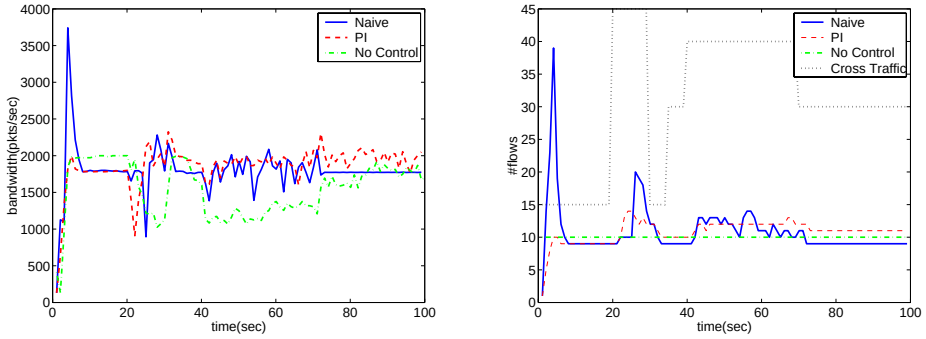

Figure 1: Achieved Bandwidth (left) and eTCP tunnel width (right) using the Naïve, PI, and no controls.

any way limited to control theory; as such, we also mention other theoretic frameworks which would work equally well in control theory's place (for assessing their own ranges of "correctness").

3.1 Control Theory

A controller is any artifact (whether actual software, a protocol, *etc.*) used to affect the parameters and behaviors of system components in response to observations of the system. TCP congestion control rules [9], admission control algorithms [14], size-aware routing [8], and the eTCP tunnel width controller [7] (which we will discuss below), are all but few examples of useful controllers we would like to be able to analyze in isolation and in composition with each other and other controllers. Notice that interesting controllers are often based upon *feedback*: the controller alters some parameter, makes an observation of the effects of that change, and adapts again based upon the comparison of that observation with some target result.

Say we are proposing a new controller for a novel network application. There are a number of questions we can ask about the behavior of that controller, many of which may be important to the stable and resilient behavior of our system:

- Does the system converge to the targeted value or not? Does it oscillate? Does it diverge?
- What is the mean steady-state error of the controller?
- How long does it take for the controller to first reach or cross its targeted value (rise time)?
- Can the controller cause the system to exceed its targeted value (under-damped) or not (over-damped)?
- How long does it take for the system to converge, within some acceptable margin of error, upon the targeted value (convergence time)?

Basic Feedback Control

Consider the simple controller depicted in Figure 2.

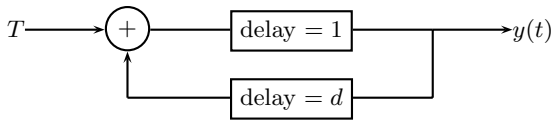

Figure 2: AIAD Controller with Feedback Delay d

The output of this system is given by

$$y(t+1) = \begin{cases} y(t-d)+1 & \text{if } y(t-d) < T \\ y(t-d) & \text{if } y(t-d) = T \\ y(t-d)-1 & \text{if } y(t-d) > T \end{cases}$$

Such a system is extremely simple to analyze; by depicting the evolution of $y(t)$ given a fixed target T and no delay as shown in Figure 3(left). We can see that this controller converges after $T+1$ units of time with zero steady-state error. But, when we add delay to the feedback path, this controller takes on some potentially undesirable properties as shown in Figure 3(right): while it still reaches T in $T+1$ time units, it then overshoots the target and enters a perpetual cycle with a period of $4d+2$, with d steady-state error.

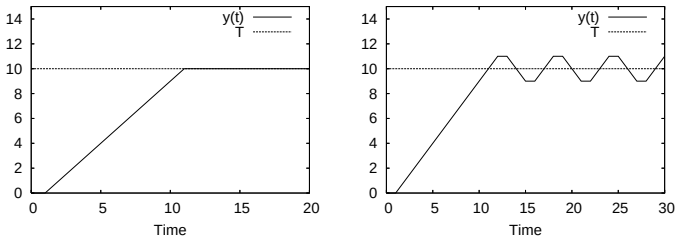

Figure 3: Time domain evolution of an AIAD Controller, $T = 10$, $d = 0$ (left) and $d = 1$ (right)

Consider also the MIMD controller depicted in Figure 4. The output of this system is given by

$$y(t+1) = y(t) + \alpha(T - y(t-d))$$

If there is no feedback delay ($d = 0$) and $0 < \alpha \leqslant 1$, this controller asymptotically approaches the target value, with error at time i of $(1-\alpha)^i \times T$, as depicted by the first curve in Figure 6.

Towards More Complex Controllers

All three of the above examples can be easily represented in the time domain, as we have done in Figure 3, making them very easy to characterize mathematically. For many interesting controllers, however, this approach to analysis becomes very difficult. The control function itself often has more elaborate shapes (AIMD, SIMD, P, PI, PID, *etc*). The input to the system may not be a step function (0 until time 0, T after time 0), but may rather be any function $x(t)$: steps, impulses, sinusoidal or other periodic waveforms, or general aperiodic

functions. We discuss examples of more complex controllers below.

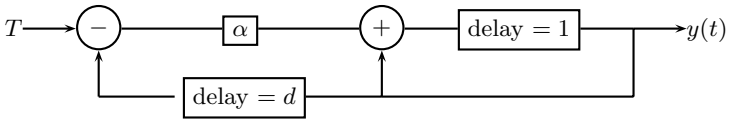

Figure 4: MIMD Controller with Feedback Delay d

Reasoning About Behaviors of Cascaded Controllers: Perhaps most challenging (and our principal concern for this paper), controllers are often cascaded, nested, and generally composed in a variety of ways, giving rise to obvious and subtle emergent behaviors which can be very difficult to reason about in the time domain. For example, consider a series of three MIMD controllers, the first taking a step input but the second taking its input from the output of the first, and the third taking its input from the output of the second as depicted in Figure 5.

Figure 5: Cascading MIMD Controllers

While we can describe with an equation the behavior of the first controller, the second and third controllers then convolve its output, producing a result we can derive numerically (Figure 6) but which lacks a convenient closed form with which we can reason about precise qualitative properties the system.

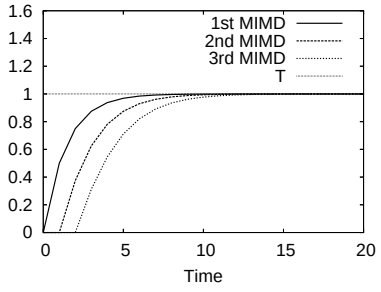

Figure 6: Output of Cascaded MIMD Controllers

Reasoning About Effects of Feedback Delay: Now consider the case of the MIMD controller (Figure 4) with non-zero feedback delay. This has significant qualitative effects upon the controller's properties, as illustrated in Figure 7; the system can easily become under-damped (overshooting its target and having to correct later), and if the delay (d) is sufficiently large with respect to α (the *gain* of the controller) the system will become unstable, periodically passing the target value on its way to infinitely increasing lower and upper extremes.

Again, deriving a closed form representation of this behavior in time is not a practical exercise for those who may be designing and specifying a system.

Reasoning About Effects of Complex Stimuli: Now envision the MIMD controller of Figure 4, but replacing T with $x(t)$, a periodic binary signal `111011101110`... (three "on"s followed by one "off"). The result has an interesting shape (Figure 8) which, while it can still be explored numerically, is thoroughly inamicable to traditional closed-form analysis in the time domain.

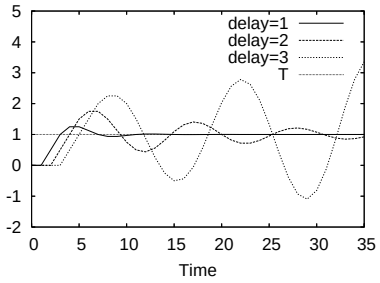

Figure 7: MIMD controller with Delay $d = \{1, 2, 3\}$

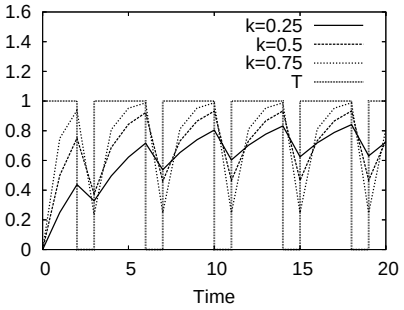

Figure 8: MIMD controller under periodic input

Principles of Control Theory

Such complex systems are much more straightforward to represent in the *frequency domain* than in the time domain [15]. By applying the Z transform (the discrete version of the Laplace transform) to a signal or function, we are able to manipulate those signals and functions in Z space:

$$\begin{aligned} Z(x(n)) &= X(z) \\ &= x(0) + z^{-1}x(1) + z^{-2}x(2) + \ldots + z^{-i}x(i) + \ldots \\ &= \sum_{i=0}^{\infty} z^{-i}x(i) \end{aligned}$$

This has the advantage of turning temporal convolutions (composition of controllers) into simple multiplications. Thus, we exchange control functions (as we have been using them above) for *transfer functions* which describe the relationship between a controller's input and output in the Z (frequency) domain rather than in time, as illustrated in Figure 9.

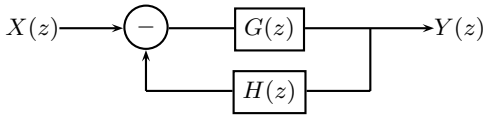

Figure 9: Transfer Function

Many of the functions we may wish to be able to model and consider have very simple closed forms in the Z domain. For

example:

$$\textbf{Impulse}\ \ x(i) = 1\ 0\ 0\ 0\ 0\ 0\ 0\ldots \Rightarrow X(z) = 1 \tag{1}$$

$$\textbf{Step}\ \ x(i) = 1\ 1\ 1\ 1\ 1\ 1\ 1\ldots \Rightarrow X(z) = \frac{1}{1-z^{-1}} \tag{2}$$

$$\textbf{On/Off}\ \ x(i) = 1\ 0\ 1\ 0\ 1\ 0\ 1\ldots \Rightarrow X(z) = \frac{1}{1-z^{-2}} \tag{3}$$

$$\textbf{Exponential}\ x(i) = 1\ a^1\ a^2\ a^3\ \ldots \Rightarrow X(z) = \frac{1}{1-az^{-1}} \tag{4}$$

Given the forward $G(z)$ and feedback $H(z)$ transfer functions in the closed loop controller in Figure 9, and a function representing the stimulus $X(z)$, one can derive the output $Y(z)$ as:

$$Y(z) = G(z)\,(X(z) - H(z)Y(z))$$

Taking $X(z)$ to be unity (*i.e.*, an impulse function in the time domain), we get the transfer function of the system (or equivalently, the impulse response of system):

$$\frac{Y(z)}{X(z)} = \frac{G(z)}{1 + G(z)H(z)}$$

P Controller: Consider the **P** controller, where the output is proportional to the error. For an output delay d, the time domain description of this controller is:[2]

$$y(i+d) = \alpha(x(i) - y(i)), \quad \alpha > 0$$

$y(t)$ can be transformed into a transfer function in the Z domain:

$$\frac{Y(z)}{X(z)} = \frac{\alpha z^d}{z^{d+1} + \alpha}\ . \tag{5}$$

Recall the Z transform of our input ($X(z)$, the step function) from Equation 2; we can now derive (in the Z domain) the output of the system ($Y(z)$) as the convolution (product) of the step input signal ($X(z)$) with the **P** controlled system (transfer function, $\frac{Y(z)}{X(z)}$),

$$Y(z) = \frac{\alpha z^{(d+1)}}{(z-1)(z^{(d+1)} + \alpha)} \tag{6}$$

With this result, control theory can immediately provide us with a qualitative description of the behavior of the system. For example, to demonstrate that the controller is stable, it is sufficient to demonstrate that the denominator's roots (*poles*) of the transfer function $\frac{Y(z)}{X(z)}$ lie within the unit circle, *i.e.*, $|z| < 1$. In the present case, this demands that $\alpha < 1$ (independent of d). Furthermore, because the roots are real, the system is *over-damped* (*i.e.*, it will not exceed its target value). It is also straightforward to determine the steady-state error of the system using the Discrete Final Value Theorem by taking the limit of $(1 - z^{-1})Y(z)$ as $z \to 1$ to find the steady-state value:

$$\lim_{z\to 1}(1 - z^{-1})Y(z) \ = \ (1 - z^{-1})\frac{\alpha z^{(d+1)}}{(z-1)(z^{(d+1)} + \alpha)} \ = \ \frac{\alpha}{1+\alpha}$$

Thus, the **P** controlled system has a steady state error of $(1 - \frac{\alpha}{1+\alpha})$ or $(1+\alpha)^{-1}$, independent of feedback delay.

Now consider a case analogous to that presented in Figure 5, in which we cascade n **P** controlled systems. The transfer function of such a system would be:

$$\frac{Y(z)}{X(z)} = \frac{\alpha_1}{(z+\alpha_1)}\frac{\alpha_2}{(z+\alpha_2)}\cdots\frac{a_r}{(z+\alpha_r)}\cdots\frac{\alpha_n}{(z+\alpha_n)} = \prod_{i=1}^{n}\frac{\alpha_i}{(z+\alpha_i)}$$

Note that for this system to be stable, it is sufficient to prove that

$$\max_{r=1}^{n}(\alpha_r) < 1\ .$$

For composition of **P** controlled systems, testing stability does not require that we retain the gain valuess of each individual controller within a composite controller; we only need the maximal gain value across the set of constituent controllers to make this assessment.

The precise steady-state value (given a step input of 1) is:

$$\lim_{z\to 1}(1 - z^{-1}) \times \frac{z}{z-1} \times \prod_{r=1}^{n}\frac{\alpha_r}{(z+\alpha_r)} \quad = \quad \prod_{r=1}^{n}\frac{\alpha_r}{(1+\alpha_r)}$$

with a steady-state error of $1 - \prod_{r=1}^{n}\frac{\alpha_r}{(1+\alpha_r)}$. Notice that the error is bounded from above by the *minimum* gain value:

$$1 - \prod_{r=1}^{n}\frac{\alpha_r}{(1+\alpha_r)} \leqslant 1 - \left(\frac{1}{1+\min_{r=1}^{n}(\alpha_r)}\right)^n$$

While the latter expression clearly represents a looser bound than the former, it also succeeds in hiding much of the detail of the makeup of a composite system, making it much easier for us to imagine taking that result and plugging it into a higher-level framework for evaluating a broad set of correctness criteria for a system.

PI Controller: Assume that all members of the vector A have value k, *i.e.*, the time-domain control function is

$$y(t+1) = \sum_{i=0}^{t} k(x(i) - y(i))$$

Then the transfer function is

$$\frac{Y(z)}{X(z)} = \frac{k}{z + (k-1)}$$

which is *stable* if $k < 2$.

Because the **PI** controller compensates for the accumulated error of the system, we would expect for its steady-state error to go to zero, which is indeed the case:

$$\lim_{z\to 1} 1 - \left((1 - z^{-1})\frac{k}{z+(k-1)}\frac{z}{z-1}\right) = \lim_{z\to 1}\left(1 - \frac{k}{z+(k-1)}\right) = 0$$

Composing P with PI: Suppose we now compose a **P** and **PI** controlled systems, in that order (denoted as "$\mathbf{P} \oplus \mathbf{PI}$"). Then the transfer function for $\mathbf{P} \oplus \mathbf{PI}$ is:

$$\frac{Y_{\mathbf{P}}(z)}{X_{\mathbf{P}}(z)} \times \frac{Y_{\mathbf{PI}}(z)}{X_{\mathbf{PI}}(z)} = \frac{\alpha}{z+\alpha} \times \frac{k}{z+(k-1)}$$

Now, using a step function for input (recall Equation 2), we derive the output of the system in the Z domain as follows.

[2]We are simplifying the exposition in this paper by assuming the input to the controller simply uses a measurement of its own output. In reality, the output of a controller usually affects a system (called *plant* in control-theoretic terms), then the output of the system is measured and fed-back to the controller.

$$
\begin{aligned}
Y(z) &= \frac{\alpha}{z+\alpha} \times \frac{k}{z+(k-1)} \times \frac{z}{z-1} \\
&= \frac{\alpha k z}{(z-1)(z+\alpha)(z+(k-1))}
\end{aligned}
$$

which indicates that the system is stable if $\alpha < 1$ and $0 < k < 2$. Notice that this result is (perhaps counter-intuitively at first) independent of the order of the controllers.

The steady-state value of $\mathbf{P} \oplus \mathbf{PI}$ is:

$$
\begin{aligned}
\lim_{z\to 1}(1-z^{-1})Y(z) &= \lim_{z\to 1}(1-z^{-1})\frac{\alpha k z}{(z-1)(z+\alpha)(z+(k-1))} \\
&= \lim_{z\to 1}\frac{(z-1)}{z}\frac{\alpha k z}{(z-1)(z+\alpha)(z+(k-1))} \\
&= \frac{\alpha}{(1+\alpha)}
\end{aligned}
$$

This is identical to the steady-state value of $\mathbf{P}$ in isolation, and therefore that the total steady-state error is also the same; this makes intuitive sense, because $\mathbf{PI}$ tends in steady state toward its input with zero cumulative error. Once again, it seems that (at least for assessing some first-order properties like steady-state error values and accumulated error) it may be permissible to discard information about the internals of a composite controller ($\mathbf{PI}$s) without losing the ability to decide the correctness of the system.

3.2 QoS Theory

Another example of theories that could be leveraged to create typing hierarchies is that of *network calculus or QoS theory*. QoS (Quality-of-Service) theory has taken off over a decade ago [10] and has quite matured since then in the form of algebras that support the composition of various components to achieve a composite predictable performance behavior from networking systems. These algebras [2, 11] include both *deterministic* calculus and *statistical* calculus—the former deals with *hard* guarantees on performance measures such as maximum delay bound or no message loss, whereas the latter deals with *probabilistic* guarantees on performance measures such as a bound on the tail of the message delay distribution or message loss probability (*e.g.*, Probability that message delay is greater than D is less than ϵ).

For illustration, consider deterministic QoS calculus. The basic elements of that theory are the so-called *arrival envelope* and *service curve*. Intuitively, the *arrival envelope* $A(t_1, t_2)$ describes the worst-case (maximum) amount of data that a traffic source component could inject into another network component during the time period $[t_1, t_2)$. This traffic source component could be a user source that is external to the networking system, or it could be another neighboring network component that is serving as relay of user traffic. The *service curve* $S(t_1, t_2)$ describes the worst-case (minimum) amount of data that a message service component could serve from a specific set of traffic streams (flows) during the time period $[t_1, t_2)$. Given these two worst-case descriptions, we can characterize worst-case performance parameters and metrics such as maximum message backlog in the system, minimum required service capacity, minimum required buffer space, etc. For example, the worst-case (maximum) data backlog during $[t_1, t_2)$, $Q(t_1, t_2)$, is simply $A(t_1, t_2) - S(t_1, t_2)$.

Consider two example components: *traffic shapers* and *schedulers*. A *traffic shaper* is a component which takes a traffic flow as input and produces as output another flow that is shaped according to a certain characterization. Many types of shapers exist. For example, a *leaky peak-rate* shaper (L-shaper) would enforce a maximum rate, say R_i, on the output flow i, or conversely, a minimum spacing between any two consecutive messages of flow i. A *token-bucket* shaper (TB-shaper) would shape the output flow i so it won't inject more than $\sigma_i + \rho_i \times (t_2 - t_1)$ messages in $[t_1, t_2)$, where σ_i is a maximum message burst size and ρ_i is the (long-term) average message rate of flow i. A *delay-jitter* shaper (DJ-shaper) would delay a message if it comes earlier than an expected arrival time, thus every message experiences a fixed delay. Traffic shapers can be composed to yield properties that are composite of properties of constituent shapers. For example, a TB-shaper followed by an L-shaper yields a shaper which enforces both a maximum and an average message rate. See Figure 10.

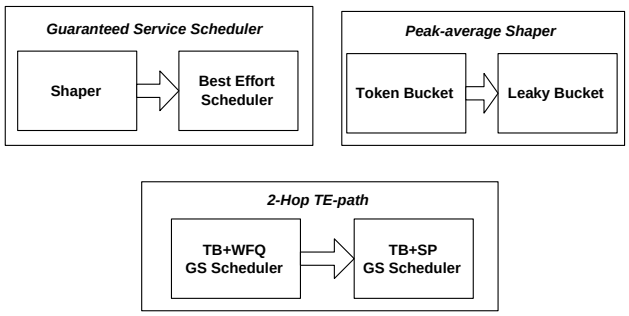

Figure 10: Composition of QoS Components

In addition to traffic shapers, important components of any QoS architecture are *message schedulers*. A scheduler determines the order in which messages are sent out onto the next network component. Many types of schedulers exist. For example, a *Weighted Fair Queueing* (WFQ) scheduler approximately emulates a Generalized Processor Sharing (GPS) scheduler where different traffic flows (or sets of flows), viewed as "fluid" flows (rather than the realistic view of message flows), share the maximum service capacity in proportion to certain weights. A *Static Priority* (SP) scheduler maintains a certain number of priority levels, where flows that map to the highest priority are served first, and hence these flows would experience lowest delays. Whether a scheduler provides a certain share of a resource's capacity (*e.g.*, WFQ) or a certain low-delay service (*e.g.*, SP), such service is "best-effort" in nature, *i.e.* the performance is not predictable since the number of flows sharing a scheduler and their traffic behavior may change over time. Thus, this type of schedulers are termed *best-effort (BE) schedulers*.

However, a BE-scheduler can be composed with a certain traffic shaper to elevate its type to a so-called *guaranteed-service (GS) scheduler*, which provides certain guarantees on performance. For example, the composition of a TB-shaper and a WFQ-scheduler *can* yield a GS-scheduler which guarantees a maximum message delay. Specifically, for a flow i shaped by a TB-shaper (σ_i, ρ_i) served by a WFQ-scheduler at a minimum share of ρ_i, the maximum delay experienced by any message from flow i is given by $\frac{\sigma_i}{\rho_i} + \frac{M}{C}$, where M is the maximum message size and C is the maximum service capacity[3]. The proof

[3]For such maximum-delay guarantees to work, an additional component, called *admission controller*, should ensure that

of such maximum-delay property follows a worst-case analysis as mentioned earlier. Specifically, over any time period $[t_1, t_2)$, assuming "fluid" (ideal) GPS, we have:

$$\begin{aligned} Q(t_1,t_2) &= A(t_1,t_2) - S(t_1,t_2) \\ &\leq [\sigma_i + \rho_i \times (t_2 - t_1)] - \rho_i \times (t_2 - t_1) \\ &= \sigma_i \end{aligned}$$

Thus, the maximum delay under GPS is upper-bounded by $\frac{Q(t_1,t_2)}{\rho_i} = \frac{\sigma_i}{\rho_i}$. Since WFQ is a message-oriented non-preemptive approximation of GPS, there is an approximation error of $\frac{M}{C}$, which accounts for the transmission (service) time of one message served earlier than under GPS. Thus, under WFQ, the maximum message delay is upper-bounded by $\frac{\sigma_i}{\rho_i} + \frac{M}{C}$.

Similar to this TB+WFQ composition, a TB+SP is another possible composition which yields another type of GS-scheduler. Traffic-engineered (TE) paths could also be composed out of such GS-schedulers. Figure 10 illustrates an example TE-path composed of TB+WFQ followed by TB+SP, whose properties can be computed from the individual properties of its constituents—for example, the TE-path guarantees a maximum delay that is simply the sum of the maximum message delays guaranteed by each of the TB+WFQ and TB+SP GS-schedulers.

Not all compositions would yield desirable global properties. For example, traffic shapers that only enforce an average message rate on their output flows would not compose with BE-schedulers to yield hard guarantees on performance. This is because a peak (maximum) message rate is required to capture the worst-case behavior of an incoming traffic stream. Even worse, compositions may yield undesirable emergent behaviors, *i.e.* the composition is worse than its parts. For example, the composition of longer chains of service components, *e.g.*, a series of L+SP, may yield worse maximum delay-jitter (and hence increased buffer space requirement to avoid message losses) than individual components as traffic burstiness may increase due to possible accumulated interactions between flows.

The recent work of Shin and Lee [16] is an example of a system which yields abstract descriptions of the properties of complex components. Rather than working directly with the internal details of real-time scheduling systems (workloads, algorithms, and actual schedules), their periodic resource model affords a straightforward expression of the needs and capabilities of real-time schedulers in terms of a small set of linear equations. Schedulability is then expressed in terms of necessary and sufficient conditions upon these equations, allowing us to completely set aside all internal details of the system while retaining the ability to precisely determine whether a set of schedulers can be composed under a super-scheduler in a such a way that they will still guarantee their deadlines will be met.

4. CONTROLLERS AS TYPED GADGETS

Theories like control theory and QoS theory allow us to abstract out properties of our building blocks, with the hope that it will be possible to reason at a higher level about these abstractions. These abstractions in turn have a number of uses. We may impose restrictions on inputs that ensure that (output of) a controller (or a controlled system) satisfies some invariants (*e.g.*, minimum period of stimulus). compositions of controllers or QoS components by composing the properties of constituent ones (*e.g.*, maximum steady-state error by taking minimum gain). Constraints could be imposed upon compositions to satisfy some invariant of the composition (*e.g.*, bounded feedback delay).

As control and QoS theoretic results may afford us more detail than we actually require, there is nothing to prevent us from "loosening" descriptions of systems and components where those less precise descriptions are sufficient to demonstrate some desirable invariants. For example, it may suffice simply to know whether the controller is over-damped or not, or whether the total steady-state error decreases exponentially, polynomially, or at all with time, or whether the aggregate signaling path delay exceeds some threshold, or some combination of such properties which circumscribe a larger range of systems than the particular precise characterization we are able to derive. This simpler abstraction, in turn, may be suitable for export into non-control/QoS-theoretic domains which are interested in reasoning about high-level qualitative properties of the components and their interactions without getting bogged down in the minutiae of the particular available analysis technique applied to derive those underlying conclusions.

This approach is the very essence of a *type*: an abstract description of some object within a system which captures interesting invariants while discarding ancillary details which may clutter the higher-level abstract analysis.

All we require of any proof, characterization, or analysis system for its results to be integrated into this model is the ability to structure its space of characterization results and requirements as a taxonomic partial order, with the less restrictive characterizations (*supertypes*) as parent nodes to more restrictive (*subtype*) ones. This space may include both qualitative and quantitative dimensions; for example, for a particular safety criteria it may suffice to treat **P** controllers as a special case of **PI** controllers[4], and in which we define classes based upon upper bounds on their constituent gains (gain of less than 0.8, *etc.*). This creates the simple two-dimensional type lattice shown in Figure 11. Another example is a lattice of types representing the minimum and maximum gains present in composite controllers made up of individual **P** controllers, useful for assessing a loose bound on steady-state error and whether the system is always stable or not. The resulting type lattice is shown in Figure 12.

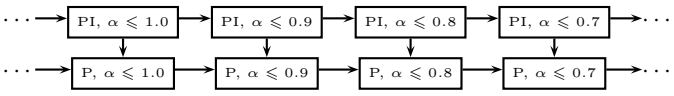

Figure 11: Type lattice for P and PI controllers.

These are simple and intuitively clear examples of type spaces that can be easily derived from control theoretic abstractions and results. We have already discussed several other possibilities, such as the effects of the **PI** controller disappearing in steady state when composed with a **P** controller and the delay insensitivity of **P** in steady state. These results, as well as results implying the stability and dampedness of a controller,

the sum of the total shares $\sum \rho_i$ allocated to flows going through the scheduler does not exceed C.

[4]A **P** controller is a **PI** controller where only the 0^{th} member of the A vector is non-zero.

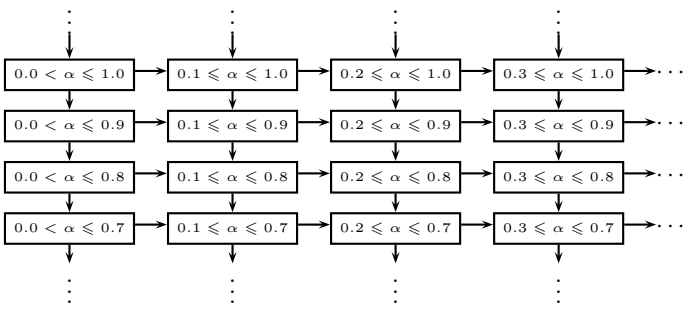

Figure 12: Type lattice of P controller compositions.

can easily be integrated as dimensions in a control-theoretic type lattice. Similarly, a QoS-theoretic type lattice can be obtained to infer different properties such as a bound on bandwidth or delay as we compose several QoS components.

We now turn toward an examination of the mechanisms needed to integrate these results into a model of a larger composite system.

5. SAFELY COMPOSING TYPED GADGETS

In order to develop a system which can mechanically infer properties and results from types like those sketched above in Section 4, we must formally define a domain of applications and rigorously specify rules for the construction and inference of types based upon the elements of those applications. In this section we begin by describing a conceptual structure for *flows*, the building blocks from which composite applications are constructed, including rules structuring the composition of such flows. We then present the syntax by which types are assigned to flows, and offer rules for assigning types to the various expression forms making up the specification language, followed by a brief discussion of the forms of type analysis we can apply over this class of specifications.

5.1 Flows

Figure 13 illustrates the structure of the basic compositions of two controllers or *flows*, A and B, with the types of their corresponding sockets: $f1$, $f2$, $f3$ and $f4$ for the forward (signaling) channel, $b1$, $b2$, $b3$ and $b4$ for the backward (feedback) channel. bn to refer to the sockets

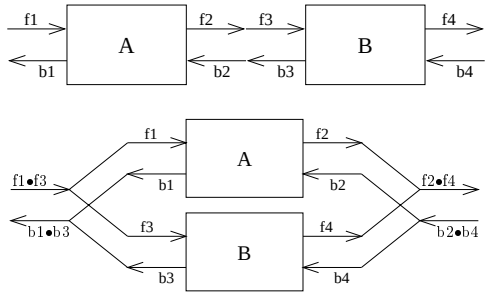

Figure 13: Sequential (top) and Parallel (bottom) Composition.

Intuitively, the forward channel is a typed signal; its type may include such properties as its convergence (or lack thereof), rise time, settle time, whether or not it is over-damped, its steady-state error, *etc.* The backward channel is similarly typed to reflect the origins of the feedback signals it carries, whether and how much delay the feedback signal(s) experience, *etc.* As such, a flow is analogous to a function which takes two arguments (the incoming sockets) and returns two values (the outgoing sockets), and we would expect the typing of flows to reflect such a functional relationship. Any single "type" we may assign to or infer for a flow (*e.g.*, "a **P** controller with $\alpha = 0.4$") will correspond with a 4-tuple of such types describing the allowable inputs and range of outputs that controller is compatible with and capable of.

5.2 Specification of Global Flows

We represent a specification of a network application as a *global flow*. A global flow (or simply a *flow*) is a composite object, built up from *local flows* and *flow variables*. Local flows are single components for which we posess complete type information (*e.g.*, A and B in Figure 13(a)); these generally represent either a particular physical or logical intermediary or endpoint to a specified system. Flow variables are "place holders" in a specification representing components which are not yet known or fully specified; they could represent unknown clients or servers at the endpoints of a specification, unknown intermediary services or transport networks in the middle, or any compositions thereof. In general, global flows can be composed with each other, with local flows, or with flow variables to create larger global flows.

We represent global flows syntactically using the following BNF:

x, y, z	$\in$ FlowVar			flow variable
A, B, C	$\in$ LocalFlow			local flow
$\mathcal{A}, \mathcal{B}, \mathcal{C}$	$\in$ GlobalFlow	$::=$	$A \mid x$	
		$\mid$	$\mathcal{A}; \mathcal{B}$	sequential flow
		$\mid$	$\mathcal{A} \| \mathcal{B}$	parallel flow
		$\mid$	**let** $x = \mathcal{A}$ **in** $\mathcal{B}$	let-binding

Intuitively, a *sequential flow* $\mathcal{A}; \mathcal{B}$ is a composition like that in Figure 13(a) in which two flows are placed (logically) adjacently and joined, while a *parallel flow* $\mathcal{A} \| \mathcal{B}$ is one in which two flows are placed (logically) parallel, offering simultaneous rather than serial service and data flow (see Figure 13(b)). The sequential operator ";" and the parallel operator "$\|$" have the same precedence, and both associate to the left, i.e.,

$$\begin{array}{lll} A_1; A_2; A_3 & \text{means} & ((A_1; A_2); A_3) \\ A_1 \| A_2 \| A_3 & \text{means} & ((A_1 \| A_2) \| A_3) \\ A_1 \| A_2; A_3 \| A_4 & \text{means} & (((A_1 \| A_2); A_3) \| A_4) \ . \end{array}$$

Here are examples of flow specifications:

1. $A_1 \| A_2; A_3$
2. $x; A_4; y$
3. **let** $x = A_1 \| A_2; A_3$ **in let** $y = A_5 \| A_6$ **in** $x; A_4; y$

The first flow is *closed*, because it does not mention free variables; the second is *open*, because it mentions free variables x and y; and the third is *closed*, because it does not mention free variables (variables x and y are *bound* by **let**s). The third flow specification above is unambiguously parsed as

$$\textbf{let } x = A_1 \| A_2; A_3 \textbf{ in } (\textbf{let } y = A_5 \| A_6 \textbf{ in } x; A_4; y)$$

and can be "executed" to produce the equivalent specification:

$$A_1 \| A_2; A_3; A_4; (A_5 \| A_6)$$

The matching parentheses in this expression cannot be omitted because of the associativity rule stated above.

Substitution and flow variables

In order to reason about gaps in a global-flow specification (flow variables), we must first formalize what it means for those gaps to be filled. This is done by defining *substitution*: Substituting $\mathcal{A}$ for x in $\mathcal{B}$ is written $[x := \mathcal{A}]\mathcal{B}$. The notion is made precise by induction on the definition of flows:

$$\begin{aligned}
[x := \mathcal{A}]y &= \begin{cases} \mathcal{A} & \text{if } x = y, \\ y & \text{if } x \neq y, \end{cases} \\
[x := \mathcal{A}]A &= A, \\
[x := \mathcal{A}](\mathcal{B}\,;\mathcal{C}) &= ([x := \mathcal{A}]\mathcal{B})\,;([x := \mathcal{A}]\mathcal{C}), \\
[x := \mathcal{A}](\mathcal{B}\|\mathcal{C}) &= ([x := \mathcal{A}]\mathcal{B})\|([x := \mathcal{A}]\mathcal{C}), \\
[x := \mathcal{A}](\textbf{let}\ \ y = \mathcal{B}\ \ \textbf{in}\ \ \mathcal{C}) &= (\textbf{let}\ \ y' = [x := \mathcal{A}]\mathcal{B}\ \ \textbf{in}\ \ [x := \mathcal{A}]\mathcal{C}^*),
\end{aligned}$$

where $\mathcal{C}^*$ is $\mathcal{C}$ with every *free* occurrence of y renamed into the fresh variable y', which will guarantee two important (but not necessarily intuitively obvious) properties: (1) $\mathcal{A}$ will not be substituted for a bound occurrence of x in the subexpression $\mathcal{C}$ of (**let** $y = \mathcal{B}$ **in** $\mathcal{C}$) in case $x = y$, and (2) no free occurrence of y in $\mathcal{A}$, if any, will be captured by the outer let-binding after the substitution. Intuitively, this simply ensures that unbound variables remain unbound and that bound variables are substituted using the appropriate let.

5.3 Syntax of types

We represent constraints placed upon the behaviors of any component of the system using varieties of types. Types can be socket types (forward or backward), plain types (forward or backward), or flow types, where types and flow types are built up from socket types.

A *socket type* is a description of a single logical "entry" or "exit" point for a flow. The type itself may be drawn from one or several of the theoretical infrastructures discussed in Section 3 or suggested in Section 6; for example, on a forward socket of an eTCP tunnel node it may describe the steady-state error of a tunnel, whether the adaptation is monotonic (overdamped) or not, or the convergence rate, and on a backward socket it may describe such properties as cumulative feedback delay or feedback origins. A *plain type* is an ordered list of socket types, describing a (perhaps composite) socket, *i.e.*, a socket which actually corresponds with entry and exit points to one or more parallel flows (as in Figure 13(b)). A *flow type* is a 4-tuple representing both the forward and backward entry and exit points to a (perhaps composite) flow. This represents the complete type specification of a component in the flow specification. We will present these tuples graphically as two-by-two matrices so each element's position in the matrix corresponds with its graphical placement in the flow of Figure 13.

The syntax of types, and the metavariables ranging over their different categories, are given by the following BNF definition:

$$\begin{aligned}
r &\in \mathsf{FwSocketType} \\
s &\in \mathsf{BwSocketType} \\
t &\in \mathsf{SocketType} && ::= r \mid s \\
\rho &\in \mathsf{FwType} && ::= r \mid r\,\rho \\
\sigma &\in \mathsf{BwType} && ::= s \mid s\,\sigma \\
\tau &\in \mathsf{Type} && ::= \rho \mid \sigma \\
T &\in \mathsf{FlowType} && ::= \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}
\end{aligned}$$

Note that $\mathsf{SocketType} \subset \mathsf{Type}$, but that $\mathsf{Type} \cap \mathsf{FlowType} = \varnothing$, *i.e.*, it is safe to promote a socket type to a plain type but that no plain type (of itself) constitutes a flow type.

Notice that the number of socket types in a plain type occupying one element of a flow type need not match the number of socket types in another element; this allows us to construct laterally asymmetric flows, like the split and combine flows illulstrated in Figure 14. We could also hypothetically construct vertically asymmetric flows (*e.g.*, with more forward than backward sockets on the left side), but we forbid these as a matter of convention.[5]

A useful operation on plain and flow types is concatenation, denoted "$\bullet$", defined in the obvious way: if $\tau_1 = t_1 \cdots t_m$ and $\tau_2 = t_{m+1} \cdots t_n$ with $m, n \geqslant 1$, then $\tau_1 \bullet \tau_2 = t_1 \cdots t_n$, and extended to flow types:

$$\begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \bullet \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} = \begin{bmatrix} \rho_1 \bullet \rho_3 & \rho_2 \bullet \rho_4 \\ \sigma_1 \bullet \sigma_3 & \sigma_2 \bullet \sigma_4 \end{bmatrix}$$

This allows us to easily construct types for parallel (Figure 13(bottom)) and asymmetric (Figure 14) flows.

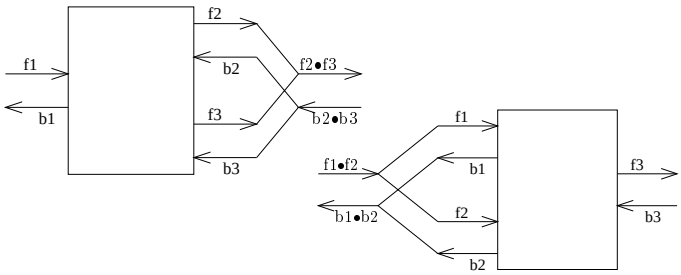

Figure 14: Splitting and Combining Flows

5.4 Subtyping

Suppose that each forward socket type is drawn from a partial order like those described above in Section 4 (*e.g.*, describing gain bounds or the dampedness of the controlled value), and that each backward socket type is drawn from a similar partial order (*e.g.*, describing maximum feedback delay). We say that Δ is the set of such subtyping assumptions on forward and backward socket types, and that each individual rule is written as $r_1 <: r_2$ for some $r_1, r_2 \in \mathsf{FwSocketType}$ (read as "r_1 is a subtype of r_2") or as $s_1 <: s_2$ for some $s_1, s_2 \in \mathsf{BwSocketType}$ (read as "s_1 is a subtype of s_2"). We extend the subtyping relation to types and flow types, using the following axioms:

$$\frac{\{t_1 <: t_2\} \subseteq \Delta}{\Delta \vdash t_1 <: t_2} \qquad \frac{\tau \in \mathsf{Type}}{\Delta \vdash \tau <: \tau} \qquad \frac{T \in \mathsf{FlowType}}{\Delta \vdash T <: T}$$

and the following inference rules:

$$\frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta \vdash \tau_2 <: \tau_3}{\Delta \vdash \tau_1 <: \tau_3} \qquad \frac{\Delta \vdash T_1 <: T_2 \qquad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3}$$

$$\frac{\Delta \vdash r <: r' \qquad \Delta \vdash \rho <: \rho'}{\Delta \vdash r\,\rho <: r'\,\rho'} \qquad \frac{\Delta \vdash s <: s' \qquad \Delta \vdash \sigma <: \sigma'}{\Delta \vdash s\,\sigma <: s'\,\sigma'}$$

$$\frac{\Delta \vdash \rho_1' <: \rho_1 \qquad \Delta \vdash \rho_2 <: \rho_2' \qquad \Delta \vdash \sigma_1 <: \sigma_1' \qquad \Delta \vdash \sigma_2' <: \sigma_2}{\Delta \vdash \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} <: \begin{bmatrix} \rho_1' & \rho_2' \\ \sigma_1' & \sigma_2' \end{bmatrix}}$$

[5] If such a structure is useful, it can be represented by noting the type of the socket which would be removed as "**void**".

Intuitively, we are simply declaring that every socket type, type, and flow type is (reflexively) a subtype of itself and offering mechanisms for extending subtype relationships from constituent types to composite types, *i.e.*, socket subtype relations can be composed to form plain subtype relations and plain subtype relations can be composed to form flow subtype relations.

For example, consider the upper-and-lower gain bound types diagramed in Section 4; because we know that $(0.3 \leqslant \alpha \leqslant 0.7) <: (0.2 \leqslant \alpha \leqslant 0.8)$ and that $(0.2 \leqslant \alpha \leqslant 0.9) <: (0.1 \leqslant \alpha \leqslant 1.0)$, we can judge that the the parallel socket

$$(0.3 \leqslant \alpha \leqslant 0.7) \bullet (0.2 \leqslant \alpha \leqslant 0.9)$$

is a subtype of the parallel socket

$$(0.2 \leqslant \alpha \leqslant 0.8) \bullet (0.1 \leqslant \alpha \leqslant 1.0)$$

(*i.e.*, if we judge a socket to have the former type, it is safe to treat it as having the latter type).

Those familiar with formal type systems will recognize that subtyping on flow types (the last rule) is *contravariant* in the two types along the first diagonal and *covariant* in the two types along the second diagonal; if we think of a flow as analogous to a function, this corresponds intuitively with the sockets' roles as *input* and *output* sockets, respectively. Thus, again using the bounded-gain type system from Section 4, we can judge that

$$\begin{bmatrix} (0.1 \leqslant \alpha \leqslant 1.0) & (\alpha = 0.9) \\ (0.4 \leqslant \alpha \leqslant 0.6) & (0.2 \leqslant \alpha \leqslant 0.7) \end{bmatrix} <: \begin{bmatrix} (0.1 \leqslant \alpha \leqslant 0.5) & (0.8 \leqslant \alpha \leqslant 0.9) \\ (0.4 \leqslant \alpha \leqslant 0.6) & (0.4 \leqslant \alpha \leqslant 0.5) \end{bmatrix}$$

(*i.e.*, any flow with the first flow type can be safely used in a context requiring a flow conform to the second flow type).

5.5 Typing rules

The point of assigning types to flows and their constituent elements is to enable us to abstract away the internals of each component and perform compositional analysis to assess whether the pieces of a global flow specification will be able to interact according to the declared composition structures, *i.e.*, whether the pieces "fit" together, and what "shape" any gaps (flow variables) in the specification might have. This is done in two stages: first, we specify typing rules which formally encode how types can be assigned to global flows as they are built up from local flows, flow variables, and other global flows; second, we discuss the formulation of practical algorithms for deducing these conclusions from a specification provided by the system architect or programmer.

Figure 15 presents a formal declaration of rules for typing global flows based upon complete or partial knowledge of the types of their constituent flows. As previously, Δ is the set of subtype declarations, T is a flow type, $\mathcal{A}$ is a global-flow specification, A is a local flow, $\mathsf{type}()$ is a function assigning types ($\mathsf{FlowType}$) to local flows, and Γ (the *type environment*) is a partial function assigning types ($\mathsf{FlowType}$) to flow variables ($\mathsf{FlowVar}$).

We say that $\mathcal{A}$ *type checks* if there is a type environment Γ and a flow type T such that if we assume Γ the rules in Figure 15 can be applied to derive the conclusion that $\mathcal{A}$ has the flow type T.

For typing rules and conclusions to become usable in a real system, we must define a practical type analysis procedure or algorithm (call it $\mathcal{P}$) which, given an arbitrary global flow $\mathcal{A}$ as input, always terminates and returns one of two kinds of results:

1. $\mathcal{P}(\mathcal{A}) =$ '$\mathsf{no\ solution}$', meaning that $\mathcal{A}$ is not typable.
2. $\mathcal{P}(\mathcal{A}) = \langle \tilde{\Gamma}, C, \tilde{T} \rangle$, where $\tilde{\Gamma}$ is a type environment, C is a consistent set (possibly empty) of constraints, and $\tilde{T}$ is a type.

Intuitively, the result '$\mathsf{no\ solution}$' indicates that some composition within $\mathcal{A}$ was inferred to have a potential incompatibility. For example, in the composition $A; B$, assume that A's forward outgoing type is a single socket indicating a **PI** controller with gain $\alpha = 0.3$, while B's forward incoming type is a single socket with a type requiring a **PI** controller with gain $\alpha \geq 0.6$; clearly these parameters are incompatible, so their composition could introduce instability or other incorrectness into the system.

As another more subtle example, assume that A's forward outgoing socket had a type indicating a **PI** controller with gain $0.2 \leq \alpha \leq 0.5$ but B's forward incoming socket had a type requiring a **PI** controller with gain $0.4 \leq \alpha \leq 0.8$; even though it is *possible* the elements could interact properly (if it happens that $0.4 \leq \alpha \leq 0.5$), it is also possible they will not (if $\alpha < 0.4$ or $0.5 < \alpha$).

The second (and preferred) result represents our goal: a description (called $\tilde{T}$) of the type of global flow $\mathcal{A}$ and descriptions (the set C) of the "shapes" of all flow variables within $\mathcal{A}$. Intuitively, C offers (perhaps partial) types for each flow variable appearing in $\mathcal{A}$, *e.g.*,

$$x : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$$

such that we are assured that any flow B with type T

$$T <: \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$$

can safely be substituted for x in $\mathcal{A}$, *i.e.*, it is safe to *instantiate* the specification as **let** $x = B$ **in** $\mathcal{A}$.

There are different architectural approaches to implementing the procedure $\mathcal{P}$ which carries out the type analysis of a global flow specification. We classify $\mathcal{P}$ algorithms as belonging to one of two families: *non-compositional* or *compositional*.

Non-compositional Type Analysis

A non-compositional $\mathcal{P}$ algorithm can only complete if its operand is a global-glow containing no flow variables. Intuitively, this algorithm is capable of checking the correctness of a complete end-to-end description of a service (flow), inferring the invariants (types) which hold for the sockets at the end points, but is not capable of working around gaps in the specification which represent unknown services or components. Clearly this approach is very useful in closed-system analysis, but its utility is limited when considering open systems which may be arbitrarily extended in the future; each such modification requires the whole-system analysis be performed again from scratch. This also makes it difficult to perform meaningful type analysis in the presence of information hiding, where certain components of a service may wish to reveal only necessary constraints for other services which they can embed or be embedded within without revealing their own structure, because the full system description musts be acquired by the single instance of $\mathcal{P}$ to complete the analysis.

$$\text{(var)}\quad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(sub)}\quad \frac{\Gamma, \Delta \vdash \mathcal{A} : T \quad \Delta \vdash T <: T'}{\Gamma, \Delta \vdash \mathcal{A} : T'}$$

$$\text{(local)}\quad \frac{\mathsf{type}(A) = T}{\Gamma, \Delta \vdash A : T} \qquad \text{(par)}\quad \frac{\Gamma, \Delta \vdash \mathcal{A} : T \qquad \Gamma, \Delta \vdash \mathcal{B} : T'}{\Gamma, \Delta \vdash \mathcal{A} \| \mathcal{B} : T \bullet T'}$$

$$\text{(seq)}\quad \frac{\Gamma, \Delta \vdash \mathcal{A} : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \qquad \Gamma, \Delta \vdash \mathcal{B} : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} \qquad \Delta \vdash \rho_2 <: \rho_3 \qquad \Delta \vdash \sigma_3 <: \sigma_2}{\Gamma, \Delta \vdash \mathcal{A}\,;\mathcal{B} \; : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}}$$

$$\text{(let)}\quad \frac{\Gamma, \Delta \vdash \mathcal{A} : T \qquad \Gamma \cup \{x : T'\}, \Delta \vdash \mathcal{B} : T'' \qquad \Delta \vdash T <: T'}{\Gamma, \Delta \vdash \textbf{let} \;\; x = \mathcal{A} \;\; \textbf{in} \;\; \mathcal{B} : T''}$$

Figure 15: Typing Rules for Flows

Compositional Type Analysis

A compositional $\mathcal{P}$ algorithm is much more flexible in that it can analyze global-flows containing flow variables. Rather than simply deriving types for the endpoints of a flow, compositional $\mathcal{P}$ is also able to infer the "shape" of gaps in the service description represented by flow variables. This supports a modular approach to analyzing flows and assembling services, in which local information hiding is acceptable and analyses of new extensions to the system can be naturally integrated with existing modular analyses without requiring another full iteration of bottom-up type inference and checking.

6. CONCLUSION AND FUTURE WORK

In this paper we presented the skeleton of an approach to building more reliable, stable, and robust software for use in open and extensible system environments and architectures. This section offers a roadmap of ways in which we envision "fleshing out" this approach.

Theoretical Infrastructure

Computation in Types: Some varieties of types, *e.g.* those reflecting the delay in a feedback signal, may include numerical values which must be operated upon as part of the type inference operation itself (*i.e.*, composing two flows with delays d_1 and d_2 yields a composite flow with delay $d_1 + d_2$); we must devise suitable rules for handling these cases and prove that they do not break our system.

Transfer Function Identification/Inference: In addition to exporting declared controller types for compositional analysis, it may be possible under limited conditions (*e.g.*, within a domain-specific language designed for this purpose) to recognize some control functions in their temporal form within a program's syntax and to identify their correspondence with well-known controller types (**P**, **PI**, **PID**, *etc.*) based upon the program syntax which implements them and (perhaps) corresponding parameter values or ranges, effectively deriving the basis for control-theoretic analysis of control functions from the implementations themselves.

Pragmatics and Applications

This paper has provided a high-level sketch of the structure of type spaces based upon control theoretic results. These sketches offer more of a compass arrow than a roadmap toward developing useful type taxonomies which preserve properties which can and should be meaningfully stated, inferred, composed, and tested in practical programming environments.

Useful Control Theoretic Types: We need to define a working set of meaningful properties which our type inference and resolution engine can work with. Careful thought must be given to the meaning of a "subtype" in a control-theoretic context, whether describing a forward input or output value or the forward or backward dimension of a flow or the whole flow. It will be especially interesting to establish which subtype and supertype relations do and do not hold when particular safety criteria are being targeted and to develop type inference strategies which are best able to take advantage of this knowledge along with programmer-supplied constraints upon the salient set of invariants to be enforced.

Additional Sources for Type Models: While this paper has focused upon control theory as a basis for forming type taxonomies to describe the performance of components in extensible/open systems, the type engine and general architecture are not uniquely tailored to control theory; as such, many other systems for expressing safety and correctness properties could also be employed instead of (or in addition to) control theory to define the type space and the algorithms needed to reconcile and infer types. Three examples follow.

1. Queuing Theory:
The analysis of composite systems is common in queuing theory; sequences of queues in open and closed loops are commonly used to describe a whole range of environments and systems. Queuing system descriptions are themselves a kind of type; "$M/M/3/20/100/FIFO$" is a queue taking a memoryless (Poisson) input, memoryless (exponential) service times, three parallel servers, a maximum queue length of 20, a maximum population in the system of 100, and a first-in first-out scheduling discipline. This nomenclature lends itself naturally to defining subtype relationships (*e.g.*, "$M/M/1$" <: "$M/G/1$" <: "$G/G/k$"). Depending upon the metrics of interest, loose bounds can easily be used to simply describe complex combinations of queues; for example, two parallel $M/M/1$ queues will have no better average wait time than a single $M/M/2$ queue taking the combination of the two queues' inputs. Similarly, since the output process of an $M/M/1$ queuing system is Poisson (memoryless), two $M/M/1$ queues can be arranged serially and still be described as taking memoryless input and producing memoryless output (although the output rate will be the least of the arrival and departure rates between the two).

2. Scheduling Theory:
Related to queuing theory is the description of scheduling systems for periodic or deadline-driven tasks. Such models could be used to describe packet transmission patterns over sockets (whether the minimum required outgoing capacity or incoming rate) or co-scheduling requirements (*e.g.*, in a sensor network environment where nearby nodes wish to schedule transmissions so as to minimize collisions). Some expressions of periodic schedules also lend themselves naturally to subtype relationships; for example, if a process is able to produce no fewer than a units of output every b units of time, it is also able to produce na output per nb time for any integer $n \geqslant 1$.

3. Model Relations:
The notions of forward-simulation, backward-simulation, bisimulation, *etc.* [12, 13] could have useful applications in describing subtypes for stateful processing nodes in a composite system. Each forward/backward socket pair of a flow could be annotated with an *event signature* describing its expected output alphabet, its (causal) correspondence with expected input values, and similarly the (causal) correspondence of the input alphabet with output symbols, effectively forming a partial finite-state machine with which all potential peers must be able to safely interact without violating any of its stated expectations (invariants).

Summary

Many in the networking community have recently been advocating a vision of flexible networking architectures, which are programmable to fit the needs of a specific application or a class of applications. This means that the user should be able to easily program such architectures such that specific components are composed to make up the whole system. Individual components, operating at different levels of the architecture, would adapt their local control rules in their interaction with other components and the changing environment. These local control rules, when composed, must finally lead to global properties that satisfy end-to-end services. Such properties should include safety measures including predictability of performance, trust and progress.

Towards that end, this paper introduces a research agenda that aims at defining a compositional specification language and its associated type hierarchies inspired by typing in general programming languages. This high-level specification would hide from the user low-level compositional specifications—derived from theories such as control and QoS theories—by only exposing "looser" specifications, *e.g.* whether the composition of two controllers yields an over-damped or under-damped system. This is an ambitious research agenda which we believe is at the core of the "Science of Networking Design".

7. REFERENCES

[1] CS Department Boston University. The Internet Traffic Managers Project. http://www.cs.bu.edu/groups/itm/.

[2] Jean-Yves Le Boudec and Patrick Thiran. Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Springer-Verlag Lecture Notes on Computer Science 2050.

[3] Adam Bradley, Azer Bestavros, and Assaf Kfoury. Safe Composition of Web Communication Protocols for Extensible Edge Services. In *Proceedings of the 7th International Web Caching and Content Delivery Workshop*, Boulder, CO, August 2002.

[4] Adam D. Bradley, Azer Bestavros, and Assaf J. Kfoury. Systematic Verification of Safety Properties of Arbitrary Network Protocol Compositions Using CHAIN. In *IEEE Conference on Network Protocols (ICNP)*, Atlanta, GA, November 2003.

[5] Gali Diamant, Leonid Veytser, Ibrahim Matta, Azer Bestavros, Mina Guirguis, Liang Guo, Yuting Zhang, and Sean Chen. itmBench: Generalized API for Internet Traffic Managers. Technical Report BU-CS-2003-032, Boston University, Computer Science Department, Boston, MA 02215, December 2003.

[6] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. he Stable Paths Problem and Interdomain Routing. In *IEEE/ACM Transactions on Networking*, volume 10, April 2002.

[7] Mina Guirguis, Azer Bestavros, Ibrahim Matta, Niky Riga, Gali Diamant, and Yuting Zhang. Providing Soft Bandwidth Guarantees Using Elastic TCP-based Tunnels. In *Proceedings of ISCC '2004: The Ninth IEEE Symposium on Computers and Communications*, Alexandria, Egypt, June 2004. (Extended version available as Boston University, CS Dept Technical Report BUCS-TR-2003-028).

[8] Liang Guo and Ibrahim Matta. Scheduling Flows with Unknown Sizes: An Approximate Analysis. In *Proceedings of ACM SIGMETRICS '2002*, Marina Del Rey, CA, June 2002. Poster.

[9] Shudong Jin, Liang Guo, Ibrahim Matta, and Azer Bestavros. A Spectrum of TCP-friendly Window-based Congestion Control Algorithms. *IEEE/ACM Transactions on Networking*, 11(3), 2003.

[10] Jim Kurose. Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks. *Computer Communication Review*, 23(1):6–15, January 1993.

[11] Jorg Liebeherr, Stephen Patek, and Almut Burchard. A Calculus for End-to-end Statistical Service Guarantees. Technical report, University of Virginia, 2001.

[12] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[13] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[14] Ibrahim Matta and Azer Bestavros. A Load Profiling Approach to Routing Guaranteed Bandwidth Flows. In *Proceedings of Infocom'98: The IEEE International Conference on Computer Communication*, San Fransisco, CA, April 1998.

[15] K. Ogata. Modern Control Engineering, 4th Ed. Prentice Hall, 2002.

[16] Insik Shin and Insup Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *24th IEEE International Real-Time Systems Symposium (RTSS'03)*, 2003.

[17] Ioannis Stavrakakis, Ibrahim Matta, and Michael Smirnov (organizers). COST-IST (EU)—NSF-ANIR (USA) NeXtworking workshop 2003. Crete, Greece, June 23-25, 2003.

[18] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *Proceedings of ACM SIGCOMM*, Pittsburgh, PA, August 2002. ACM.

[19] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, 2001. ACM.