Abstract

A Dual-System Approach to Realistic Evaluation of Large-Scale Networked Systems

Richard Allan Alimi

2010

Many useful techniques and infrastructures have been developed to test the correctness and performance of computer systems before deployment. However, they are limited when applied to the large-scale networked systems that are prevalent today, from Internet routing to peer-to-peer (P2P) video delivery. First, testing infrastructures are typically smaller in size and scale than the production infrastructure, limiting the scale of possible test scenarios. Second, keeping the testing infrastructures and their configurations synchronized with the production infrastructures is difficult or impossible. Finally, capturing and integrating production workloads into the testing infrastructures can be challenging.

To address these shortcomings, we have developed novel techniques to conduct correctness and performance evaluations on production infrastructures for large, networked systems. We show how incorporation of domain-specific knowledge of a system being tested can provide an efficient testing infrastructure that is scalable, accurate, and easily integrates production workloads. Our techniques also avoid disruption to users of the production system.

In particular, we first present a general formulation of a dual system running both the production and the tested systems concurrently. We then present ShadowNet, a realization of the dual-system concept at the link layer and network layer. ShadowNet can be used by network providers (e.g., ISPs) to evaluate the correctness and performance of network configurations before deployment. We also present

PEAC, a realization of the dual-system concept at the application layer, to introduce additional capabilities. In particular, using P2P live streaming as an example, we show how the tested system can be used to handle the production system's workload while avoiding disruption to users. PEAC introduces a scalable and distributed control technique that allows a developer to construct test scenarios with a large number of nodes.

A Dual-System Approach to Realistic Evaluation of Large-Scale Networked Systems

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Richard Allan Alimi

Dissertation Director: Yang Richard Yang

December 2010

Contents

List of Figures

List of Tables

Acknowledgements

First of all, thank you to my advisor, Richard Yang. His guidance and insight have been crucial towards my growth both inside and outside of research. I have especially appreciated the freedom he has given to explore and manage my own research interests. Within his group, I have been fortunate to explore multiple fields from wireless networking to P2P to network management, thanks to Richard's broad knowledge and seemingly-endless thirst to explore and understand. Through many intense discussions, he has pushed me to find the core of a problem, clearly articulate it, and present a solution. Richard has also been a wonderful mentor outside of research. In particular, he urged and supported involvement in professional activities, which have been instrumental in my personal growth.

I also thank my committee members, Mike Fischer, Sanjai Narain, and Avi Silberschatz, for insightful discussions on the work presented in this dissertation. This dissertation has also greatly benefited from insights and and tireless work of Chen Tian, Ye Wang, and David Zhang (from PPLive). Many others deserve special thanks for helpful discussions and feedback: Lorenzo Alvisi, Jim Aspnes, Matthew Caesar, Yuan Dong, Bryan Ford, Charles Kalmanek, Karthik Lakshminarayanan, Li Erran Li, Jennifer Rexford, Ehab Al-Shaer, Zhong Shao, Hao Wang, Xiaowei Yang, and Xuan Zhang.

Beyond those mentioned above, I have also been fortunate to work with many wonderful collaborators, especially Lili Qiu, Ramachandran Ramjee, Dawei Shen, Jingpu Shi, Yanjun Sun, Harish Viswanathan, and Yin Zhang. Each one has brought forth unique knowledge and insights, and I have learned much from each of them.

There have been numerous exciting events during my time at Yale. Most notable was the intense excitement leading up to the P4P July 2008 field amongst the research group at Yale, Hao Wang, Ye Wang, Richard Yang, as well as Haiyong Xie and Arvind Krishnamurthy, the team at Pando including Laird Popkin, James Royalty, and Ramit Hora, and the team at Comcast including Chris Griffiths, Jason Livingood, Rich Woundy, and other members of the P4P Working Group including Marty Lafferty, Doug Pasko, and others. Beyond having the privilege to be part of an amazing team, I learned much from them about network operations and real P2P system deployments. I have also been fortunate to be able to continue many of these collaborations and friendships to the current day.

During my time at Yale I have both enjoyed the friendship of and benefited from discussions with members and visitors of the Yale LANS research group: Lijiang Chen, Hongqiang Liu, ZhiHui Lu, Chen Tian, Haiyong Xie, Hao Wang, Ye Wang, and Xuan Zhang. There have been many interesting projects and I look forward to keeping in touch. I am also thankful for friendships I have developed with others in the department, in particular Fitz Nowlan, Lev Reyzin, and Nikhil Srivastava, Xueyuan Su, and Andi Voellmy.

And last but certainly not least, thank you to my wife Jen Yamtich who has been by my side since the beginning graduate school, and to my parents, Paul and Maryann Alimi, who have provided constant love and support.

Richard Alimi

New Haven, CT, USA

September 2010

Chapter 1

Introduction

Large-scale, networked systems are crucial components in today's Internet. They support many necessary features from the basic functionality provided by computer networks to the applications run on top of these networks. However, to support increasing demand and diverse requirements, these systems have become increasingly complex. For example, they typically involve multiple interacting components, are designed to scale horizontally to many users, implement complex features, and handle failures on a regular basis. Each of these requirements may also interact in complex ways.

As a result of the increased complexity, many networked systems have become difficult to understand, contain bugs, or operate in sub-optimal states. Changes that attempt to fix bugs or improve performance may end up introducing new bugs or perform worse than before the change was made. For example, user forums for P2P applications such as PPLive (*e.g.*, [76]), BitTorrent (*e.g.*, [11]), and Skype (*e.g.*, [82]) show that updates may introduce additional bugs or reduce performance. Changes to networking infrastructure cause either reduced performance for end-users or even a failure for network traffic to reach its intended destination [52].

1.1 Need for Real Evaluations

To better understand networked systems and evaluate changes or alternative designs, it is necessary to have a platform for performing evaluations. Multiple techniques for testing networked systems have been proposed. Modeling and simulation approaches are useful for exploring impacts of certain designs or changes and gathering understanding of relationships between key properties in a system. Logging and replay frameworks are useful for debugging already-observed issues since they enable a developer to inspect the internal state and execution of a complex system. Lab testing and staging infrastructures are also frequently used to execute a system and observe behavior in an environment similar to the production environment.

Though existing techniques are useful, they are not sufficient for realistically evaluating correctness and performance of networked systems at a large scale. These existing techniques suffer from a single, basic problem: they do not have the capability to test how previously-unknown, but crucial, behaviors would affect a system if it were fully deployed. In particular, abstract models used by simulation and modeling approaches do not capture the full complexity of many networked systems and the environments in which they run. Logging and replay frameworks are limited to detecting bugs that have already been observed and are not suitable for testing alternative scenarios or system behaviors. Finally, though lab and staging infrastructures may be able to run a networked system in various testing scenarios, they typically do not scale to the size and full complexity of the production environment. Furthermore, if the test infrastructure interfaces with real users, there is little protection against disruption if the test system does not perform well.

After considering the limitations of existing approaches, we observe that a scalable and realistic environment in which to test a networked system is the production environment itself. Testing within the production environment has the unique advantage of including realistic behaviors both of users and the environment in which the system will be running. However, there are challenges that must be addressed. First, users should be protected from the effects of unsuccessful tests. Second, test results should be accurate; they should be indicative of the behavior that would be observed if running alone on the production infrastructure. Third, the production infrastructure may have limited resources; these resources should be used efficiently.

To address these challenges, we introduce the *Dual-System Architecture* which can be used for testing networked systems in realistic environments with real users. With this architecture, the system is subjected to the full environment in which it will be running by *reusing* the production infrastructure. Designers, developers, and operators of large-scale networked systems can evaluate and improve performance and detect bugs in a realistic environment. By integrating testing and evaluation as a basic capability into a production system and taking advantage of domain-specific knowledge and novel algorithms, the Dual-System Architecture can obtain accurate evaluation results, protect users, and make efficient use of the production infrastructure's resources.

After illustrating the general Dual-System Architecture, we instantiate it in two diverse contexts. First, we present *ShadowNet*, a dual system for network configurations, that provides fundamental improvements to testing and deploying configuration changes to network infrastructure. Second, we present *PEAC*, a dual system for P2P live streaming applications, which extends the Dual-System Architecture to incorporate applications running on end-user machines.

Dual systems are a fundamental improvement to the state of art for testing large-scale networked systems. With both ShadowNet and PEAC, we illustrate both the diverse applicability and benefits to major systems in today's Internet.

1.2 Summary of Contributions

This dissertation introduces a novel architecture for testing real-world networked systems. We show that substantial benefits and capabilities are enabled by using the deployed infrastructure for both the production system and the system being tested.

Empowered by the Dual-System Architecture, we develop new techniques for testing real-world networked systems. We show that application of domain-specific knowledge enables both efficient use of the underlying infrastructure and testing accuracy, thus creating a realistic environment for large-scale testing.

Beyond presenting the general Dual-System Architecture and instantiations in two diverse settings, we make the following specific contributions in the context of ShadowNet:

- We introduce *packet cancellation*, a novel technique for testing performance impacts of configuration changes in a network infrastructure without affecting end-users whose traffic is being through the same infrastructure.

- We present algorithms for reachability analysis, illustrating how dual systems can integrate with existing modeling and invariant checking by using data captured from the live system.

- We develop transactional capabilities for network configuration management by introducing a commitment protocol that shields end-users from the convergence effects typically observed with changes to network configurations.

In the context of PEAC, we make the following additional contributions:

- We introduce the concept of allowing the system being tested to handle the production system's workload while avoiding disruptions to end-users.

- We develop a distributed test control mechanism that enables construction of test scenarios with a large number of distributed nodes (*e.g.*, clients watching a live streaming channel).

1.3 Roadmap

The dissertation is organized as follows. In Chapter 2, we present related work on testing in networked systems and illustrate their limitations. Chapter 3 introduces the overall architecture and design space for dual systems. Chapter 4 presents an instantiation of the dual-system architecture for network configurations. Chapter 5 then presents an extension to applications using P2P live streaming as an example. We discuss future directions and conclude in Chapter 6.

Chapter 2

Related Work

There are many existing techniques for testing networked systems, each providing unique capabilities and benefits. In this chapter, we survey existing techniques and discuss their limitations. It is important to note that dual systems are not meant to replace existing testing techniques, but rather to supplement them.

There is a large body of work on testing methodologies used in standard software engineering, from technologies (*e.g.*, unit testing) to organization structures (*e.g.*, separation of test engineers from software engineers). We focus on techniques for testing networked systems.

This chapter provides an overview of existing techniques for testing networked systems. We discuss both their advantages to illustrate where they may be complementary to dual systems, but also point out the limitations that dual systems can address.

2.1 Modeling and Simulation

Theoretical modeling and simulation techniques are characterized by capturing key properties of the full-scale system and creating an analytical model or an implementation focusing on key functionality. This has been a useful approach in many settings.

First, there are many generic simulation toolkits that have been developed and extended (*e.g.*, [71,80]). These toolkits implement common functionality (*e.g.*, network topologies and protocols) that approximates real-world settings. They frequently also provide interfaces to easily control and modify test scenarios. Using these toolkits, it is possible to develop a simulated version of a system and easily change parameters such as network topology, tests with machine failures, etc.

Other toolkits have been developed that enable simulation for particular types of networked systems. For example, GNS3 [37] enables one to deploy network configuration changes on an emulated topology. There exist toolkits for other types of networked applications as well (*e.g.*, [13,60] for P2P streaming systems).

Theoretical modeling and analysis has also been widely applied in many networked systems. Due to the complexity of many networked systems, it is challenging to develop a tractable model for the complete system. Thus, many models are developed for specific parts of a system. For example, models have been developed for network reachability (*e.g.*, [98]), security (*e.g.*, [43]), interdomain routing (*e.g.*, [26]), and content distribution in an application-layer overlay (*e.g.*, [12, 63, 74, 102, 104]). Even more comprehensive models have also been developed to capture interactions between different components both within the network infrastructure (*e.g.*, [28, 69]) and between network infrastructure and applications (*e.g.*, [8]).

Benefits: By focusing only on key functionalities of a system, modeling and simulation have two primary benefits. First, since only key characteristics are captured, it can take less time to identify the impacts of certain changes. Making a change to the full-scale system typically entails details such as designing test cases, regression tests, and handling error conditions. In addition, for networked systems in particular, it is typically faster to design, execute and analyze different test scenarios (*e.g.*, network conditions, failure scenarios, etc) since the physical infrastructure is not used.

Second, modeling and simulation can be helpful to understand relationships between system components and parameters. This can be enormously helpful to understand properties of particular system states without needing to execute them. For example, for network configuration managment, it has been noted that the configuration analysis tool NetDB provides AT&T significant cost savings [86].

Limitations: While modeling and simulation have their benefits, they also have important limitations. First, it can be difficult to determine exactly which key properties will affect the behavior of a complete system. The behavior of a networked system may not only depend on its own behavior, but also on the behavior of external systems that it uses (*e.g.*, DNS, BGP). Furthermore, even if certain key properties are determined to be important, one may be forced to make simplifying assumptions to keep the model tractable or simply because it is not known how to model certain behaviors. For example, measurements studies (*e.g.*, [24, 92]) have shown that the behavior of certain networks can be very different than previously assumed. As a result, a model or simulation may miss possibly-unknown interactions with properties that are not modeled or not modeled accurately [92].

Beyond choosing which properties to model, it can also be difficult or impossible to keep a simulation or model in sync with a deployed system. As a networked system evolves, a simulation or model must be kept in sync in order to produce meaningful results. For example, if a new version of Cisco IOS introduces a new feature or behavior (*e.g.*, [18]), the new feature must also be implemented in the corresponding simulator.

2.2 Logging/Replay Systems

Another useful technique for testing networked systems is logging and replay. Logging and replay consists of capturing data, typically an execution trace, from a running system in a production or staging environment, and then processing it offline. In an offline environment, it is possible to analyze the captured trace (*e.g.*, for a visual display) or even replay the system to replicate the observed behavior and potentially ease the debugging process.

Replay tools typically strive to ease debugging for parallel and networked systems by providing *deterministic* replay. Networked systems are notoriously difficult to debug due to race conditions and non-deterministic behavior, making it difficult to detect and replicate bugs.

Numerous approaches have been used to capture execution traces, spanning hardware-level (*e.g.*, [7]), OS-level (*e.g.*, [25]), user-space (*e.g.*, [36]), and even cross-layer designs (*e.g.*, [30, 81]). The various approaches frequently make tradeoffs in terms of logging overhead, types of applications they can support, whether they can replay a trace deterministically, and even relaxations on the replayed execution trace (*e.g.*, [4]).

Benefits: Logging and Replay have two primary benefits. First, they focus on using the actual system instead of a model. This allows developers to use the tools directly with existing systems with little or no modification. Indeed, some logging/replay tools are even targeted to be low-enough overhead for production systems.

Second, logging and replay tools typically provide repeatability. When debugging a complex networked system, the ability to replicate a bug and inspect the exact execution leading to an error can be extremely useful.

Limitations: Despite their benefits, logging and replay systems do have limitations for testing large-scale networked systems. The primary limitation is that they can only consider scenarios that actually occurred; they are designed to capture execution traces and are not focused on testing alternative scenarios (*e.g.*, by changing the behavior of the environment of the system itself). The ability to test alternative system behaviors or environments is important when evaluating a networked system (*e.g.*, to evaluate behavior under particular failure scenarios).

2.3 Lab and Staging Infrastructures

Another technique for evaluating networked systems is deployment at a smaller scale before making the modified system visible to all users. Using this approach, the goal is to subject the system to an environment very similar that of the full production environment.

Deploying applications at a smaller scale can be accomplished in multiple ways. First, the system can be deployed to a staging environment that is intended to replicate or approximate the production environment. For example, Cisco maintains a testing facility called NSite [19] on which network devices and configurations can be tested before deployment. Infrastructures such as Emulab [97] provide a configurable environment for networked systems, and other techniques such as PlanetLab [17], VINI [10], and CABO [27] provide a virtualized environment. DieCast [41] is unique approach in which the full system is emulated on a smaller set of physical resources.

Another methodology used in practice is deploying changes to a limited set of users. For example, PPLive has commented that before distributing a new version of their client software, they first designate a particular live streaming channel as a "test" channel. Users who join the test channel are upgraded to the new software, and PPLive monitors the performance for clients in the test channel. Even Google's large-scale distributed web infrastructure uses a similar approach [85] in which a portion of users may be included in one or more concurrently-running experiments.

Benefits: The primary benefit of lab and staging environments is that the actual networked system is executed without requiring modeling. In particular, it may be possible to capture bugs or test the system's performance under synthetic workloads or workloads captured from production systems.

Another major benefit of staging environments using real users is that they can incorporate actual user behaviors, thus testing the full end-to-end workflow including the real networking environment.

Lab and staging infrastructures also offer some control over the testing environment and parameters. For example, if using Cisco's NSite [19] facility, one could test the behavior of a network configuration under various router failure scenarios. It may also be possible to control the testing scenario when deploying to a limited set of users as well. For example, PPLive may choose the select a test channel based on the expected size of the channel or the types of users expected to view the channel.

Limitations: While lab and staging infrastructure have benefits, they also have important limitations. First, deploying and maintaining an alternate staging infrastructure may be costly. Such a staging infrastructure should be consistent with the production network in terms of not only configuration and topology, but also hardware, operating system, and software versions, since behavior may change between versions (*e.g.*, [18]). Keeping such an environment synchronized with a production environment can be a large undertaking.

Second, alternate staging environments may not fully capture the complexities of the production environment. For example, networked applications such as video streaming can be sensitive to network management policies (*e.g.*, Comcast PowerBoost [20] and rate limit [9]) implemented by some ISPs or poor peering between some ISPs may have large impacts on performance.

Third, allowing the tested system to serve real users may be risky if the system being tested does not perform as desired. Though it may be possible to reduce exposure by early detection of such cases, dissatisfied users may abandon use of a product or move to a competitor.

2.3.1 Summary

In this chapter, we have discussed existing approaches towards testing networked systems. Though each of them has unique benefits, there are important limitations motivating the need for accurately testing networked systems on the production infrastructure, while simultaneously protecting users being served by the production system.

Chapter 3

Dual-System Architecture

To address the limitations of existing testing techniques for networked systems, our novel Dual-System Architecture provides unique but complementary capabilities. In this chapter, we present the key architectural components and concepts for dual systems.

A key insight towards supporting dual systems is that consideration for a networked system's internal semantics can provide substantial benefits. Furthermore, despite diverse application semantics, there is a common, abstract framework for dual systems. When instantiating dual systems in different contexts, we observe that this architecture is capable of simultaneously achieving efficient resource usage, a realistic and accurate experimental platform, and avoiding disruption to users. By supporting these capabilities, networked systems can be safely tested using the production infrastructure achieving both a realistic and scalable testing environment.

The goal of the architecture presented in this chapter is to illustrate the key architectural components of dual systems that enable these benefits. It is important to note that though the implementation of particular components may differ when applied to different systems, the architectural components remain the same.

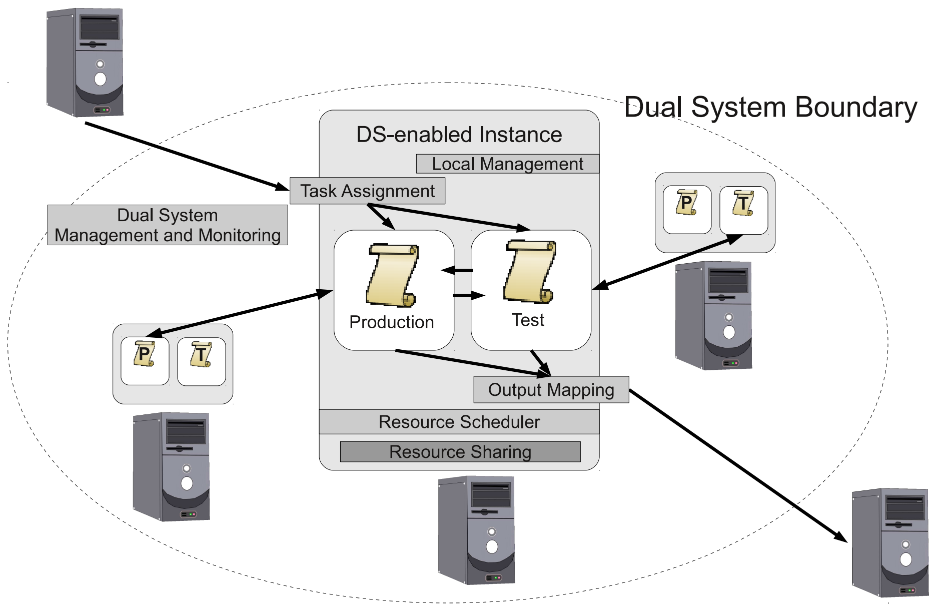

Figure 3.1: Components of the Dual-System Architecture.

Figure 3.1 illustrates the Dual-System Architecture, and remainder of this chapter presents the key components of this architecture.

3.1 DS-Enabled Instance

The first component of the Dual-System Architecture is a *DS-Enabled Instance*, which is a single physical or logical entity (*e.g.*, network device, running copy of an application) capable of concurrently executing both production and test versions of a system.

Within the dual system $D = (I_P, I_T)$, there exists a set of instances of the production system I_P and a set of instances of the test system I_T.

Instances of the production system may communicate within the Dual-System Boundary (defined later), and instances of the test system may communicate within the Dual-System Boundary.

Dual systems provide capabilities to exploit commonalities between two versions of the same system. In particular, a DS-Enabled Instance supports efficient state management and resource sharing.

3.1.1 Merged State

Both the production system and test system may have certain state information in common. Let S_P and S_T denote the state maintained by both the production and test system. It may be more efficient to store $S_P \bigcup S_T$ together than to store S_P and S_T independently. For example, the benefits of this approach have been illustrated in ShadowNet and related work [34], where the memory used for such state is both limited and expensive due to speed requirements.

3.1.2 Resource Scheduler

When running both the production and test systems concurrently, the physical resources (*e.g.*, CPU, memory, storage, network) used by both systems must be monitored and controlled. As a simple example, a DS-Enabled Instance may be designed to provide a higher priority to handling tasks that will be visible to end-users.

Beyond simple resource sharing policies, by considering the semantics of the system, resource sharing can be achieved that would be impossible by treating the networked system as a black box. This capability is used in ShadowNet to support cases when sending both production and testing traffic may oversaturate network link capacity.

3.2 System Boundary, Tasks, and Outputs

Next, the Dual-System Architecture must consider how to map its internal state and behavior to a consistent *view* seen by external entities (*e.g.*, users or other systems).

Specifically, there may be a set of external systems X which rely on output from D and/or send input (*tasks*) to D. The *Dual-System Boundary* is defined as the interface between D and X. Note that external systems may be dual systems themselves, but they observe a single, consistent view from D.

When both the production and test system are running concurrently, each may be capable of handling tasks from external systems, and each may be capable of producing outputs to external systems. Let $T_{X \to D}$ denote the set of tasks assigned to the dual system from external systems. Similarly, let $O_{D \to X}$ denote the outputs sent to external systems.

The task assignment component may deliver tasks from $T_{X \to D}$ to the production system, test system, or both. It may also delay delivery for tasks or discard them. Similarly, the output mapping component may construct the output $O_{D \to X}$ from the production system and/or test system.

For a proper interactions with external systems, the output $O_{D \to X}$ from D should be consistent with the output that would be observed if only a single system were used. However, it is important to note that the output $O_{D \to X}$ does *not* need to be identical to that produced by either the production or the test system. In particular, the dual system permits the output from both the production and test system to be merged.

Beyond mapping tasks from external systems to the production and test systems, the Dual-System Architecture permits tasks to be redistributed between the production and test systems.

3.3 Dual-System Management

Finally, the Dual-System Architecture includes a management and monitoring component. This component handles the deployment and control functions (*e.g.*, start, stop, etc) for the production and test systems. It also provides monitoring capabilities for resource usage and other runtime metrics of interest.

Beyond standard management and control functions, the management component allows the task assignments and resource usage to be changed dynamically as the production and test systems are running. The ability to shift task assignments enables ShadowNet to enable new network configurations across a network without disruption to user traffic.

3.4 Discussion

Though there are similarities to virtualization, there are important differences. Indeed, an approach to testing networked systems may be to construct a virtualized environment in which the production and test systems run independently. Even in such environments, it is possible to to achieve certain degrees of resource sharing (*e.g.*, [59]), but such techniques intentionally avoid consideration for internal semantics of a system. As a result, many virtualization techniques are immediately usable in a variety of settings.

Instead, the insight provided by the Dual-System Architecture is that there can be substantial benefits to considering the internal semantics of a networked system. In particular, the task assignment, output mapping, and task redistribution components are not provided by standard virtualization techniques, since they may harm the isolation that virtualization typically tries to provide.

Consider a generic example illustrated in Figure 3.2. In this example, there

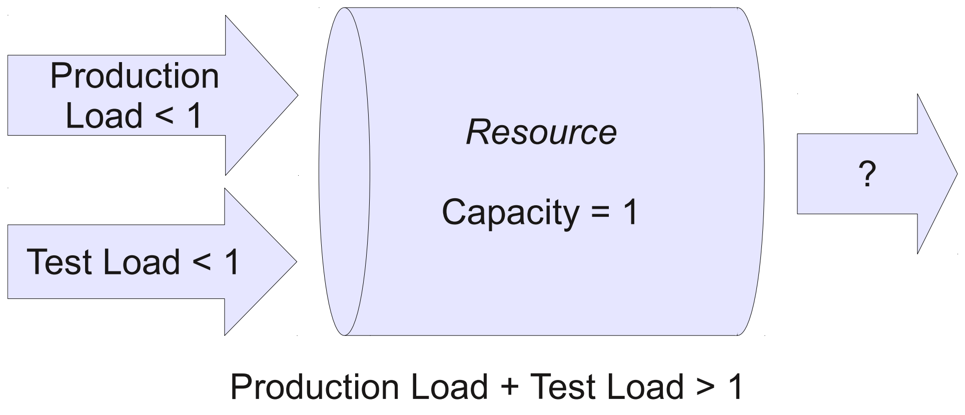

Figure 3.2: Resource Sharing oblivious to internal semantics.

is a resource on the production infrastructure (*e.g.*, bandwidth), with normalized capacity 1, that is shared by the production and test systems. Individually, each system induces a load less than 1, so the load may be satisfied if each system were running alone. However, the total load when both are running concurrently cannot be satisfied. There are then two options when both production and test systems run concurrently: drop a portion of the production system's load, or drop a portion of the test system's load. However, dropping a portion of the production system's load may cause disruption to users, and dropping a portion of the test system's load may impact the accuracy of performance tests executing on the test system. In this dissertation, we see that considering internal semantics and requirements of the production and test systems can be a powerful technique in resolving this conflict and achieving both requirements. This is exemplified by the Adaptive Task Reassignment scheduler in PEAC and Packet Cancellation algorithm in ShadowNet.

Chapter 4

ShadowNet: Dual System for Network Configuration

We first present ShadowNet, an instantiation of Dual Systems for network configurations.

4.1 Background and Motivation

Modern IP networks are becoming increasingly complex to configure, as these networks continue to evolve to offer multiple services (*e.g.*, both routing and access control), integrate equipment from multiple vendors, and conduct continuous performance and feature tuning. As a result, it is difficult to generate and maintain the configuration even for a moderately-sized network. A recent survey [73] found that configuration errors are a large portion of operator errors, which are in turn the largest contributor to failures and repair time. Another survey [52] found that more than 60% of network downtime is due to human configuration errors. It further showed that more than 80% of IT budgets are allocated towards maintaining the status quo, a percentage that will only increase due to “increased complexity, lower budgets, and continued business demand.”

4.1.1 Complexity of Network Configuration

Network configuration management is a difficult and complex task. We posit that the reason for human error is not primarily due to carelessness or insufficient knowledge, but rather the complexity of the environment that must be managed.

First, the final configuration depends on the whole network processing environment: the hardware, firmware, and software features (including the bugs!) of the routers. Typical networks are heterogeneous networks consisting of equipment from multiple vendors with distinct hardware, firmware and software features. As an example of the complexity, a survey [67] of 31 production networks found that over 200 different software versions were running on multiple hardware platforms. As another example, some routers may also offer special non-standard features (*e.g.*, Cisco-specific BGP decision steps in addition to the conventional BGP decision process [98]). As yet another example, the Cisco document [18] reports a common OSPF routing problem related with forwarding addresses. The reachability issue was caused by a bug in Cisco IOS before Release 12.1(3).

Second, the interactions of multiple services can be a source of configuration errors. Today’s networks are complex and certain behaviors may only arise when two features interact. As a simple example, the routing protocol can compute a backup path but all packets rerouted to the backup path can be dropped by a packet filter. Most tools typically focus on a single piece or set of functionality (*e.g.*, routing, access control, or QoS).

Finally, on the forwarding plane, the performance depends on traffic demand pattern, FIBs, hardware capability, and software implementations. If a tool conducts performance evaluation based on a very coarse-grained model (*e.g.*, a link is characterized by two simple numbers such as propagation delay and bandwidth), performance problems may not be revealed.

4.1.2 State of the Art

One way to reduce configuration errors is to use configuration generation tools (*e.g.*, [6]) and/or validate the configuration files using static analysis or simulation (*e.g.*, [26, 28, 43, 69, 98]). Although these tools can be quite useful, for example, it has been noted that the configuration analysis tool NetDB provides AT&T significant cost savings [86], these tools are inherently limited in the problems that they can detect. In particular, since configuration files alone do not determine the behaviors of a network, analyzing only the configuration files based on an abstract model of the network and equipment behaviors may leave many problems undetected (*e.g.*, due to forgotten network equipment or network connectivity).

Recognizing the limitations of static analysis and simulation tools, some network operators and equipment vendors build test networks. For example, Cisco has built the NSite [19] facility to test network configurations before actual deployment. However, for most companies, the cost of maintaining a testbed sufficiently similar to the operating network is prohibitive.

Given the limitations of these existing approaches, configuration modifications are frequently deployed into the operating networks without realistic testing. As a contrast, software developers depend mostly on debuggers and actually running their programs before deployment. They run unit and regression tests for correctness and conduct stress tests to validate the programs under load. It would be difficult to imagine the extent of software errors if programs were deployed after only passing through analysis or simulation tools without actually running on the target platform.

However, there is no such capability for IP network configuration [70].

4.1.3 ShadowNet Overview

To address the complexity of managing network configurations, we propose a novel capability called *shadow configurations*. With shadow configurations, a network operator may specify two configuration files for a router: one real (current) and one shadow (alternate). The shadow configuration files on a set of routers form a shadow configuration that the network operator intends to replace the current configuration files. The operator can test the shadow configuration files on the actual network without enabling them as the network's real configuration. Running on the existing network infrastructure, this capability is low cost, and thus may be utilized in daily operations. During the testing process, the current network configuration is still running and forwards real traffic; the shadow configuration carries only testing traffic and will not cause disruptions to the operation of the current configuration, even if there are errors in the shadow configuration. The operator conducts correctness and performance tests on the shadow configuration. Our usage of the term "shadow" is motivated by computer graphics, where instead of directly modifying the current display buffer, the display system often uses a shadow buffer to compute the next frame to be displayed.

In particular, by running a set of configuration files directly on the actual network to which they will be applied, a shadow configuration allows a network operator to evaluate the integrated effects of alternate configuration files, router software implementation (including incorrect protocol implementations!), the physical network status, and dynamic information such as imported external route advertisements. Many integrated effects on routing are naturally summarized by the forwarding information base (FIB) at each router. We take advantage of the compact FIB representation and develop techniques to analyze the FIBs for configuration validation and adjustment.

Further exploring its benefits, we show how shadow configuration allows a network operator to evaluate, before actual deployment in the real network, whether a set of configuration changes will have the desired effect on network performance. Such realistic performance evaluation reduces the dependency on unrealistic models or assumptions of router processing or the network. Also, the availability of the on-going real traffic in the actual network allows the operator to duplicate a controlled portion of it as testing traffic in addition to generated testing traffic. This reduces the need to generate realistic testing traffic patterns. One potential issue of conducting testing on the shadow configuration is that if we naively send both shadow and real traffic, the combined traffic may overload some network links. Thus, we develop a novel technique, referred to as packet cancellation, to allow both real and shadow traffic to be forwarded in parallel without overloading the network.

After the operator is satisfied with the new configuration, she can simply quickly and smoothly swap the real and shadow configurations with minimal network disruptions. We develop a commitment capability for shadow configurations to reduce the effects of churn and convergence. This usage pattern can be viewed as "two-phase commitment" for network configurations.

To demonstrate feasibility, we extend the Linux kernel and implement necessary components to support shadow configurations in both Quagga [77] and XORP [44] software routers. We show that shadow configurations can be implemented efficiently, with only 12 additional lines of code on the kernel's forwarding fast path for packets not using packet cancellation, and *no* code changes to routing processes. The FIB memory increase to support both real and shadow configurations is less than 35% for the worst-case router under a variety of shadow configurations for a large US tier-1

ISP; the average is much smaller, less than 7%. At run time, our shadow-enabled forwarding engine under heavy traffic has no more than 1.2% CPU usage overhead with a shadow configuration installed.

We also demonstrate the usage of shadow configurations. We show in real implementation that the commitment ability avoids the transient routing convergence period under router maintenance, shutdown and OSPF weight changes. We demonstrate our packet cancellation technique in a usage scenario where the operator tests the impact of a new configuration on a streaming video application. In this case, the combined (raw) shadow and real traffic intensity can be as high as 1.05 times the capacity of some links. However, packet cancellation shields real traffic from shadow traffic while at the same time, the measured performance of the shadow video streams is close to the case when it is using the network alone (difference is less than 1%).

In summary, we have made the following contributions:

- We propose the novel capability of shadow configurations.
- We develop novel techniques for configuration analysis, evaluation and management.
- We provide an implementation and demonstrate that the shadow configuration capability can be implemented with low overhead.

4.2 Motivating Usage Scenarios

To drive our system design, we conducted a survey of operator configuration usage scenarios. Below, we list several key usage scenarios that we would like to support using shadow configurations. The objective of the list is not to be complete, but to motivate our design.

Equipment Maintenance, Testing: A network operator may need to shutdown a running router or link for maintenance. For example, many hardware and software updates suggest that a router or network interface card be taken offline during the process. To prepare for a shutdown, the operator makes the shadow configuration the same as the real configuration except that the link or router to be shutdown does not appear in the shadow configuration. The operator evaluates the shadow configuration, makes potentially necessary adjustments, and then commits it as the real configuration.

As another example of an equipment shutdown usage scenario, a surveyed operator commented that he needs to periodically shutdown a primary link to test if its backup link is operational and will be used after network reconvergence. Since the capacity of the backup link may be lower than the primary link, such tests may cause network disruptions. With shadow configurations, he can just shutdown the primary link in the shadow configuration and test if the backup link will be used in the shadow configuration after reconvergence.

After the maintenance or addition of a new network device, the operator includes the device in the shadow configuration, evaluates the effects, and makes adjustments before switching to the new configuration. This step can be particularly useful as multiple surveyed network operators commented that it is common for issues to arise after a maintenance upgrade.

Configuration Parameter Tuning: Many network operators need to tune configuration parameters to address performance or security issues. For example, a network operator may conduct traffic engineering to improve network performance, and many traffic engineering techniques (*e.g.*, [29,31,72,78]) require the modification of configurable parameters (*e.g.*, OSPF weights or egress point selections). However, such parameter adjustments may cause disruptions due to human error or routing reconvergence. As another example, a network operator may change its network access control list. However, such changes may lead to network disruption due to misconfigurations or unexpected interaction with routing. Shadow configurations support such tuning of parameters and testing correctness and performance.

Network Diagnosis: One problem with network diagnosis is that it is difficult to conduct root-cause analysis (*e.g.*, end-to-end performance violations). Although many network diagnosis techniques have been proposed lately (*e.g.*, [14, 23, 46, 49, 53, 56, 79, 83, 84]), a major limitation of network support is that they cannot easily conduct unit or "destructive" testing [86] as is done in software debugging. Shadow configurations allow a network operator to construct a shadow network on a subset of the network, and compare the differences in the real and shadow configurations to help with root-cause analysis. In particular, the delta [100] testing technique for software debugging can be particularly helpful to the automation of configuration debugging.

Feature/Service Testing: A network operator may be reluctant to enable new features (*e.g.*, queue management or scheduling algorithms) or services (*e.g.*, VoIP) on her operational routers due to concerns of unknown performance impacts, as many factors affect network performance [16, 66, 93, 101]. Shadow configurations allow the operator to conduct an evaluation in the shadow configuration. She can finally commit the shadow configuration as the real one once the integration is verified to work correctly.

4.3 System Overview

We now present an overview of our system. The key components in our system are shown in Figure 4.1. We focus on a high-level overview in this section. Details and implementation of several components will be discussed in the following sections.

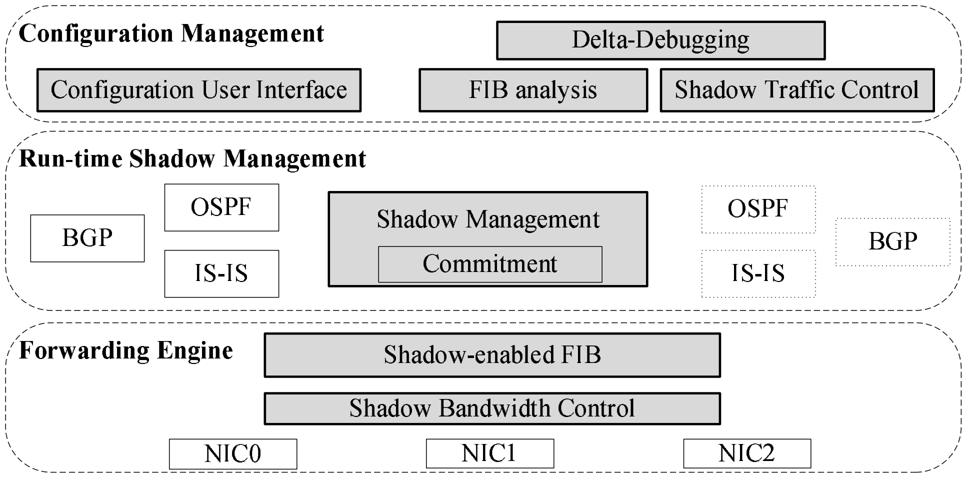

Figure 4.1: System architecture for network management with shadow configurations.

We divide our system into three layers: (1) forwarding engine; (2) run-time shadow management layer; and (3) configuration management.

4.3.1 Forwarding Engine

Foundation

The key component is a forwarding engine supporting both real and shadow configurations. In this discussion, we focus on the forwarding information base (FIB) for presentation, but note that the forwarding engine handles other items such as access control lists (ACLs) as well.

Let $\{1, \cdots, N\}$ be the set of routers in a configuration. Let $C = \{C_1, \cdots, C_N\}$ be their configuration files. In abstraction, the control plane converts the configuration files into a configuration for the forwarding plane: $C \Rightarrow \{fib_i\}_i$, where fib_i is the FIB at router i. The FIB entries at an interface maps a destination IP address to an output interface.

We refer to a set of connected routers running a shadow configuration as a *shadow-running network* or *srnet* for short. In this dissertation, we consider only the case

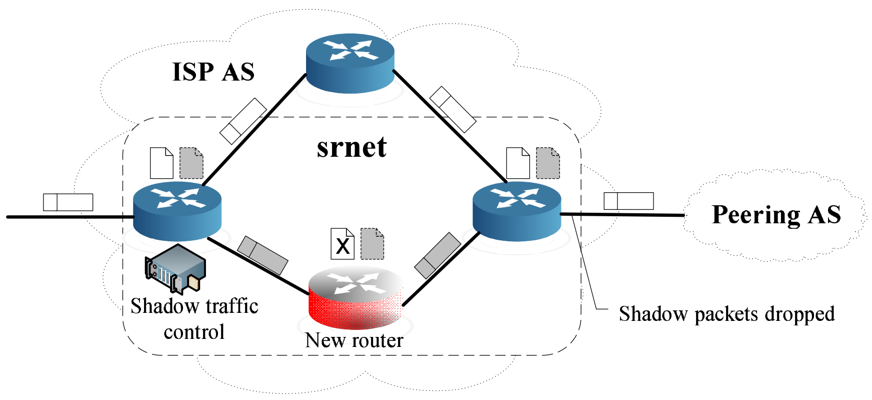

Figure 4.2: Network with an srnet being used to install a new router to support new services. The new router has its real configuration disabled during installation.

where a srnet belongs to a single autonomous system (AS). A srnet is likely to be the whole IP network of the AS, but can be only a subnet. The latter possibility gives flexibility such as incremental deployment. A router i inside a srnet has two configuration files: C_i^r for the real configuration and C_i^s for the shadow. In the forwarding engine, it will then have two FIBs, (fib_i^r, fib_i^s), for the real and shadow respectively.

A link (interface) may leave a srnet, and we refer to such a link as a border link. The FIB at such a border link will need to contain ingress and egress policies for how to handle incoming and exiting shadow packets. Figure 4.2 shows a network containing a srnet.

When a packet arrives at a border link of a router i, the router uses the ingress policy to determine whether it should apply fib_i^r or fib_i^s. We refer to a packet forwarded using the shadow configuration as a *shadow packet*, and a packet forwarded according to the real configuration as a *real packet*. Router i uses a *shadow bit* in the IP header to indicate whether it is a shadow packet or a real packet.

When another router j receives a packet, it checks whether the packet is a real packet or a shadow packet, and uses the corresponding forwarding table. If it is a shadow packet and is leaving the srnet, the egress policy is applied (*e.g.*, dropped).

Shadow Bandwidth Control

With both shadow and real traffic using the same network, we need a shadow bandwidth control component to regulate the bandwidth sharing. In particular, testing the performance of a shadow configuration should not cause disruption to the real traffic. We focus on network bandwidth, but one could also consider processing bandwidth. For example, per-packet processing such as IP lookup may be the bottleneck.

We support three modes of shadow bandwidth control:

- Priority: real traffic has higher priority than shadow;
- Partition: each configuration is allocated a portion of bandwidth;
- Packet cancellation.

Priority and partition modes can be useful, for example, when the payload must be carried end-to-end to include the responses of end hosts and servers or when evaluating deep-packet inspection. In the partition mode, the network operator can specify non-work conserving scheduling for shadow packets to provide "scaled-down" testing bandwidth and arrival processes. Note that if the objective is to predict shadow configuration network performance, then the first mode is less useful as the bandwidth allocated to the shadow configuration is unpredictable.

Packet cancellation is designed to allow an operator to conduct stress tests on the shadow configuration to reveal issues under higher load. The operator can certainly try to wait for a time period when the real traffic is low. However, there may not exist such a time period, or the real traffic load that the operator would like to duplicate for testing happens only when the real traffic is relatively high. Packet cancellation has the following two objectives:

- Performance of the real traffic is not severely degraded;
- Performance measurements taken from shadow packets should be close to the mea-

surements that would be observed if the shadow configuration were the only active configuration.

Packet cancellation is presented in Section 4.5.

4.3.2 Run-time Shadow Management

Our next layer provides two main functions:

- It provides a run-time and management environment for real and shadow configurations and routing processes (*e.g.*, multiplexing of control packets, CPU and bandwidth management). We discuss one implementation in Section 4.7, including a technique for exchanging information with routers outside a srnet (*e.g.*, with BGP) that presents a single consistent view to the outside world.
- It provides a commitment capability to smoothly swap the configurations, which is important for many usage scenarios. This is especially desirable because the convergence process is a major source of disruption: reconvergence after a configuration change can cause network outages [3] or additional configuration errors [51]. We present our commitment protocol in Section 4.6.

4.3.3 Configuration Management Layer

This layer provides multiple utilities to take advantage of and control the capability of shadow configurations. We have implemented the following tools:

- Configuration user interface (CUI): the operator is presented with two command-line terminals, one real and the other shadow. Using this interface, a network operator issues traditional router commands such as `traceroute` and `ping`. Additional commands are provided to control our commitment protocol.
- Shadow traffic control (STC): the operator is allowed to specify shadow traffic (*e.g.*,

real traffic to be duplicated to the shadow configuration and intensity of generated shadow traffic) and collect traces.

We have also investigated other useful tools:

- Shadow configuration analysis using FIB (SCAF): a tool to detect routing loops and reachability issues. We give more detail on this tool in Section 4.4.
- Shadow regression tester (SRT): a tool to play test cases (*e.g.*, reachability of important applications at important locations).
- Configuration delta debugging (CDB): a tool based on the observation that by comparing the FIB and performance of the real configuration with the shadow configuration, we can automate a large fraction of configuration diagnosis.

4.4 Runtime FIB Analysis

In this and the next two sections, we present the details of shadow configuration analysis using FIB, packet cancellation, and shadow commitment. They are presented in this order as this is a common order in many usage scenarios.

4.4.1 Objectives and Overview

With the availability of a shadow configuration, the network operator can analyze $\{fib_i^s\}_i$ *before* they become installed for real packet forwarding.

In particular, we investigate how to detect forwarding loops using the collection of FIB states. As made evident by measurement results [45] and online detection algorithms [96], forwarding loops happen frequently in real networks. Since routing loops can cause unnecessary load and dropped packets, detecting loops caused by a new configuration before its actual deployment can have great value. Our system computes the set of destination addresses as well as routers present in the loop, providing the network operator with detailed information from which she can debug the problem.

We also detect reachability issues, another common type of configuration errors [57, 98]. Some reachability issues can be extremely challenging to detect using any static analysis or simulation tools because they depend on software implementation. For example, the Cisco document [18] reports a common OSPF routing problem before Cisco IOS Release 12.1(3) related with forwarding addresses. The reachability issue noted was caused by the software implementation of a Cisco-specific optimization, and thus can be difficult to isolate using only configuration files. By looking directly at the FIB states, our system can bypass detailed modeling and abstractions, and provide the network operator useful reachability information to help debug the problem.

Note that for presentation clarity, we consider only unicast addresses; we assume that there exists a unique nexthop in each FIB for a single destination address. Also note that it is straightforward to add other forwarding mechanisms (*e.g.*, label switched paths) to our analysis.

4.4.2 Representative IP Addresses

A major complexity in reachability and forwarding loop analysis is that FIB lookup in modern routers is implemented using longest prefix matching, and different routers in the same network may have different sets of destination IP prefixes.

To use existing efficient graph algorithms to check reachability and forwarding loops, we first preprocess FIBs to compute *representative* IP addresses. With representative IP addresses, FIB analysis is done on individual IP addresses, without the need to handle longest prefix matching.

Consider a simple example where each FIB table in the network consists of the following destination IP prefixes: a default route (*i.e.*, 0.0.0.0/0), 10.1.0.0/16, and 10.1.0.0/24. Then if we verify that there is no reachability or routing loop problem for each IP address in the set {0.0.0.0, 10.1.0.0, 10.1.1.0, 10.2.0.0}, then the network has no reachability or routing loop problem.

The algorithm $findrepip$ (Figure 4.3) computes the set A of representative IP addresses for a network. The algorithm computes the set A_i of representative addresses for each FIB fib_i^s. The set A for the whole network is obtained by merging the A_i's. To make the merging efficient using a priority queue, the algorithm maintains each A_i to be sorted.

When processing each entry in fib_i^s, the algorithm adds to A_i up to two addresses. The first is the beginning address of the destination prefix associated with the entry, and the second is the beginning address of the *next* range that could come after the entry's destination prefix.

4.4.3 Computing Reachability and Loops

Once the set of representative addresses is found, each address can be analyzed using standard graph algorithms to detect reachability issues and forwarding loops:

1. Reachability: (1) set of routers R_a that can reach address a; and (2) set of routers W_a with FIB entries for address a but cannot reach address a;

2. Forwarding loops: sets of routers L_a participating in forwarding loops for address a.

Figure 4.4 shows the $checkfib$ algorithm to compute reachability and forwarding loops for each representative address. For each representative address a, $checkfib$ first constructs the forwarding graph induced by the FIBs in the network. This

```
findrepip($\{fib_i^s\}_i$) – Compute representative address set $A$
01. foreach $fib_i^s$ do
02.   $A_i \leftarrow \emptyset$ //sorted rep addr for $fib_i^s$
03.   foreach entry $e$ in $fib_i^s$ do
04.     $A_i \leftarrow A_i \cup \{\min\{e.addr\_range\}\}$
05.     if $\max\{e.addr\_range\} \neq 222.255.255.255$ then
06.       $A_i \leftarrow A_i \cup \{1 + \max\{e.addr\_range\}\}$
07.     endif
08.   endfor
09. endfor
10. // Merge rep addr into single sorted list
11. $A \leftarrow priority\_queue\_merge(\{A_i\}_i)$
12. return $A$
```

Figure 4.3: Algorithm for computing representative addresses given $\{fib_i^s\}$.

```
checkfib($A$, $\{fib_i\}_i$) – Compute
  Loops $L_a$ for each rep addr $a$
  Routers $R_a$ with reachability to each rep addr $a$
  Routers $W_a$ with FIB entries for $a$ but no reachability to $a$
01. $(L_a, R_a, W_a) \leftarrow (\emptyset, \emptyset, \emptyset)$ for $a \in A$
02. foreach $a \in A$ do
03.   $(d_a, G_a) \leftarrow makegraph(a, \{fib_i\}_i)$
04.   $R_a \leftarrow computereachable(d_a, \{fib_i\}_i)$
05.   // Find routers with FIB entries but no reachability
06.   $W_a \leftarrow N_a - R_a$
07.   // Find induced subgraph on nodes without reachability
08.   $G'_a \leftarrow inducedsubgraph(G_a, W_a)$
09.   $L_a \leftarrow findloops(G'_a)$
10. endfor
```

Figure 4.4: Algorithm for checking FIB consistency for rep addr A.

is achieved by invoking *makegraph*, which is shown in Figure 4.5. We define the forwarding graph to have a directed edge $(i \rightarrow j)$ if router i is reachable from router j. That is, router j has an FIB entry for the destination address with nexthop i. Thus, *makegraph* computes this induced subgraph $G_a = (N_a, E_a)$ and also locates the router d_a in the network that is the destination for the particular address. For simplicity, we assume that no two routers are configured as the destination for the

```
makegraph($a$, $\{fib_i\}_i$) – Compute
  Forwarding graph $G_a = (N_a, E_a)$
  Destination router $d_a$
01. $d_a \leftarrow EXTERNAL$
02. $(E_a, N_a) \leftarrow (\emptyset, \emptyset)$
03. **foreach** router $i \in N$ **do**
04.  $n_i \leftarrow lookup(fib_i, a)$
05.  **if** $i = n_i$ **then** // Found destination router
06.     $d_a \leftarrow i$
07.  **elseif** $n_i \neq -1$ **then** // Update graph
08.     $N_a \leftarrow N_a + i$
09.     $E_a \leftarrow E_a + (n_i \rightarrow i)$
10.  **endif**
11. **endfor**
12. $G_a = (N_a, E_a)$ // Construct graph
13. **return** $(d_a, G_a)$
```

Figure 4.5: Subroutine for constructing forwarding graph for destination address a.

same address; such cases are directly reported to the operator. One subroutine invoked by *makegraph* is the *lookup* subroutine, which executes an FIB lookup for a particular FIB and input address. It returns -1 if there is no forwarding entry for the address, and the router itself if the router is the final destination for the address. If there is no router in the network that is the final destination for the address, the destination router is set to the virtual node $EXTERNAL$. The $EXTERNAL$ node in the graph is implicitly reachable from all egress routers in the network.

After computing the forwarding graph, $checkfib$ invokes the *computereachable* subroutine and which uses a depth-first search on the forwarding graph starting at d_a to compute the set of routers that have paths to the destination. It is simple to see that no loops can be be present in this set, since any router participating in a loop cannot have a link to the destination address. Because of the definition of the forwarding graph, any router visited in the search has a path to the destination. Since *computereachable* is a depth-first search, its running time is $O(N + E)$.

In addition to R_a, the reachability analysis also computes the set W_a of routers in the network that have an FIB entry for the destination address a, but do not have a forwarding path to the destination. This property is of interest to a network operator since it could indicate inconsistency between their intentions and the state of the network. It could also be used to verify access control rules implemented by filtering route advertisements. It is simple to compute since it is the set of routers in the forwarding graph G_a from which d_a is *not* reachable. Since each router can be assigned an integer value $1, \ldots, |N|$, this set difference is computable in time $O(N)$.

To locate loops, we next produce the subgraph G'_a induced by routers without reachability. Loops in G'_a are located by the $findloops$ subroutine, which detects loops by performing repeated depth-first searches on the routers in the graph. When a backedge in a depth-first search is found, the set of routers in the loop is added to the result set. By overlaying the queue of unprocessed routers as a doubly-linked list on top of the routers, the overall cost of repeated depth-first searches can be remain as $O(N + E)$ since finding the next router to start the next DFS becomes a constant-time operation. Enumerating the routers in a loop is an $O(N)$ operation, making the total running time $O(N + E + NC)$ where C is the number of loops found.

4.5 Performance Testing with Packet Cancellation

With a consistent and reachable forwarding state, the network operator next might ask, "If I adopt this alternate configuration on my network, how will it perform?" Such a question is important when deploying new services such as voice or streaming media, or when the operator may want to evaluate the likely impacts of the new configuration on service level agreements.

At this point, the reader might suggest that since the operator already has the FIBs of the shadow configuration, she may compute or simulate the performance characteristics using a traffic demand matrix. This is certainly a feasible approach and our system can support it. Such computation- or simulation-based approaches, however, implicitly rely on a model for packet processing inside each router for features such as QoS or any particular queue management techniques. New techniques such as traffic shaping or differentiated services would require modifications to the model [22]. On the other hand, enabling direct measurements allows processing within the routers to be treated as a black box.

4.5.1 Overview

Recall that the objectives of packet cancellation are that (a) both real and shadow traffic are forwarded according to their original queue management schemes, and (b) shadow packets are (typically) only delayed by other shadow packets while real packets are (typically) only delayed by other real packets.

This mode uses two techniques: packet (payload) cancellation and a virtual clock. The key insight is that the payload of shadow data packets may not always need to be transmitted; that is, when the focus of an evaluation is on network performance metrics such as delay, the shadow data packets then are not intended to be received by end hosts. Thus, we need only to (1) retain the information relevant to forwarding the traffic within the network, and (2) know the correct payload size so that gathered performance measurements remain meaningful.

Given the preceding insight, we allow a router to append the header of a shadow packet to a real packet before it is transmitted over the link. The input interface at the receiving router removes the appended shadow header, and processes it accordingly. If the shadow traffic is delayed too much by the real traffic, we can append

```
pktsched() – packet cancellation and scheduling.
01. if not $empty(Q_r)$ then
02.     $p \leftarrow dequeue(Q_r)$ // Select real packet
03.     // Append shadow packet headers
04.     for $1 \ldots MAX\_CANCELLABLE$ do
05.         if not $virtual\_clock\_expired(peek(Q_s))$
06.             break
07.         $p \leftarrow append(p, ip\_hdr(dequeue(Q_s))$
08.     endfor
09.     $transmit(p)$
10. elseif not $empty(Q_s)$ then
11.     // Send shadow packet if available
12.     if $virtual\_clock\_expired(peek(Q_s))$
13.         $transmit(dequeue(Q_s))$
14. endif
```

Figure 4.6: Packet cancellation and scheduling.

multiple shadow headers to catch up with the delay.

4.5.2 Shadow Data Packet Cancellation

We now describe how our scheme processes shadow data packets. At the output interface, shadow packets and real packets are separated into two queues, Q_s and Q_r. This also allows the shadow configuration and real configuration to define different queue sizes and queuing disciplines. When it is time to transmit the next packet, the line card applies the algorithm shown in Figure 4.6.

Specifically, if Q_s is empty, send $head(Q_r)$, the head of the real packet queue; otherwise, extract the headers of the shadow packets that should be transmitted and combine them with $head(Q_r)$. We may extract multiple (up to $MAX\_CANCELLABLE$, set to 3 in our implementation) shadow packets to "piggyback" on a real packet due to packet payload sizes and previous delay of shadow packets. To determine whether a shadow packet should be transmitted or piggybacked, the shadow queue maintains a virtual clock. The virtual clock estimates whether the transmission of a shadow

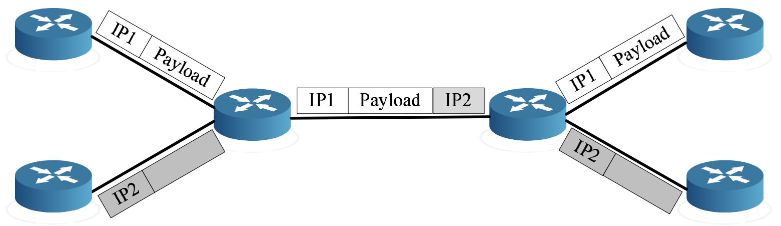

Figure 4.7: Shadow packet header combined with a real packet for transmission on a single link.

packet should be started (*virtual_clock_expired*) if there were only shadow traffic.

Note that it is important that when extracting headers from a shadow packet, we extract all IP headers to allow the scheme to work properly when tunnels or VPNs are configured. If any IP header that must be interpreted is encrypted, the scheme may not work. The TCP/UDP header, if it exists, should also be extracted since it may be required for packet filtering (*e.g.*, in Cisco's policy routing, NetFlow sampling, and firewalls). In a simple IP network without tunnels or VPNs, the extracted headers will consist of a single IP header and a TCP/UDP header, and will typically be 40 or 28 bytes in size.

There is one additional piece in the scheme. It must be possible for the incoming interface at the receiving router to determine whether it is receiving a single packet or combined packet. If the link-layer payload is larger than length indicated by the IP header, the router strips off the appended headers, verifying their IP version, header length, and optionally the checksum.

Figure 4.7 shows how a shadow payload can be canceled with a real packet for transmission over a link. The shadow header is extracted at the receiving interface and forwarded independently.

With packet cancellation, it is possible that the full size of the transmitted frame becomes larger than the next interface's MTU, causing the packets to be silently dropped at the next hop. To handle this, one could simply decrease the MTU to accommodate the additional canceled packets. To avoid additional fragmentation, one could instead increase the MTU, but internally process packets (*i.e.*, handle fragmentation) at the routers according to the original MTU.

Further consideration is required when operating on Ethernet. To provide intuition, the algorithm in Figure 4.6 might fill in all "whitespace" left by real traffic with full shadow packets, causing the link utilization to approach 100% and causing large delay variations. One simple way to solve the problem is to always transmit only the shadow header and set a timer to throttle shadow queue transmission rate when the real queue is empty. In our implementation, we found that the available timers are too inaccurate to retain the appropriate packet delay variations. Thus, we adopt the heuristic that even when the real queue is empty, only the shadow packet header is transmitted if link utilization is above a certain threshold (we use 85%). Since a previous hop may have trimmed a shadow packet, it may be necessary to expand the packet and zero-fill the payload when below the threshold.

4.5.3 Shadow Control Packet Forwarding

We previously considered only shadow data packets. Packet cancellation cannot be applied to shadow control packets, such as route advertisements, SNMP messages, or ICMP packets. For safety and because control packets can originate from many places (routing processes, ARP, ICMP, etc.), we opt to explicitly mark a shadow packet that *can* be canceled (*e.g.*, in generated testing traffic) with a PD bit, indicating that its payload is drop-able. We process shadow control packets using a separate queue.

Fortunately, control packets are a very small percentage of the traffic. It is claimed [68] that OSPF Link State Update packets consumes only 0.23 bps on average, and similarly for IS-IS. Recent measurements [48] indicate that ICMP packets account for only 0.2% of total traffic and 0.02% of total data on a backbone link.

4.5.4 Overhead and Perturbation Analysis

FIB Lookup

One potential bottleneck is FIB lookup instead of bandwidth. Since a combined packet received in packet cancellation mode contains multiple headers that might require separate lookups, it is crucial that the router be able to support this additional processing burden.

Forwarding engines in many routers are designed to handle worst-case scenarios where all incoming packets have the minimum size. In particular, assume that a router can support $\alpha \frac{L}{K_{min}}$ packets per second where L is the link speed in bytes per second; K_{min} (typically, $K_{min} = 40$ bytes) is the minimum packet size; and $\alpha \leq 1$ is the efficiency factor.

Let k_r denote the packet sizes of real traffic and k_s the packet sizes of shadow traffic. Let α_r be the link utilization caused by real traffic and α_s that of shadow traffic. To sustain lookup, we need:

$$\mathbb{E}\left[\frac{\alpha_r L}{k_r}\right] + \mathbb{E}\left[\frac{\alpha_s L}{k_s}\right] < \alpha \frac{L}{K_{min}}.$$

Using the packet size distribution in [48], we can compute α_s given α_r and α. For $\alpha \geq 0.7$ and $\alpha_r \leq 0.8$, we have $\alpha_s \geq 0.75$, meaning the link utilization for shadow traffic can reach up to 75% while still being supported by the forwarding engine.

Performance Measurement Accuracy

Our packet cancellation scheme tries to remain as consistent as possible with the original forwarding behaviors for both shadow and real packets. This is important since the operator must have confidence that the measurements obtained on real and shadow traffic are indicative of the measurements that would be observed if only real or only shadow traffic were present in the system.

To better understand our scheme, consider a basic model: packets have uniform sizes, all packets have space reserved for an additional shadow header, and packets do not arrive in the output queue when a transmission is in progress. Then, we can show that there will be no delay or loss perturbations for either real or shadow packets.

Claim 1. *For any packet p, $d^r(p) = d^s(p) = 0$ where $d^r(p)$ (*resp.*, $d^s(p)$) is the end-to-end packet delay perturbation for a real (*resp.*, shadow) packet.*

Claim 2. *For any packet p, $l^r(p) = l^s(p) = 0$ where $l^r(p)$ (*resp.*, $l^s(p)$) is the packet loss perturbation for a real (*resp.*, shadow) packet.*

4.6 Configuration Commitment

As we discussed in Section 4.3, with a consistent and reachable forwarding state and satisfactory performance, the network operator may then decide to apply the shadow configuration as the network's actual configuration. We define the objective of the commitment process to be swapping the shadow and real configurations at all routers within the srnet. Swapping allows the network to rollback if an error occurs or the operator finds the new configuration unacceptable.

4.6.1 Overview

Although there are several previous studies on updating FIBs across routers (*e.g.*, [32, 33, 103]), our shadow configuration commitment problem is distinct from these previous problems. For example, many types of changes and routing processes may be involved in a configuration change, so routing-protocol specific techniques (*e.g.*, [33]) may not apply.

Our protocol is inspired by the simple and clean map dissemination protocol proposed by Lakshminarayanan *et al.* in [61]. We address additional issues in our specific context including integration with version control of distributed configuration files, rollback of configurations, and simplicity of router maintenance.

To integrate with configuration version control (*e.g.*, CVS), before each commitment, the operator broadcasts two tags to each router: C_{old} identifies the real configuration before swap, and C_{new} the shadow configuration before swap. An additional functionality of the tags is to mark packets to avoid forwarding loops during the swapping period; this is inspired by the map dissemination in [61]. After commitment, the tags should be removed for simplicity of router maintenance.

Consider the scenario when routers always tag packets (*e.g.*, with global map sequence numbers [61]), and the network operator powers on a new router. After reading its local configuration file, a routing process (either shadow or real) must communicate with the corresponding routing processes of its neighboring routers. However, since the router does not know which tag denotes the real configuration and which denotes the shadow, it may not be able to tag routing messages correctly such that they are demultiplexed to the correct routing processes at its neighbors. One could design various ways to work around this problem (*e.g.*, designating globally constant tags or a protocol to allow a router to query tags), but they introduce extra complexity. Our commitment protocol chooses to remove the tags after commitment so that the shadow bit has well-defined semantics (0 indicates current and 1 indicates shadow) during normal operation.

4.6.2 Protocol Operation

The protocol proceeds in four phases. Messages to the routers are sent first using the real configuration, then the shadow configuration in the case where the real configuration is non-operational.

Phase 1: During the first phase, the operator sends a TAG DISTRIBUTION message containing two tags to each router. The two tags are temporary network-wide identifiers for the configurations: C_{old} identifies the real configuration before swap, and C_{new} the shadow configuration before swap. Upon receiving these tags, each router creates a lookup table to remember the mapping. To report its configuration file to version control (`diff` is enough) and to make sure that all routers have received the tags, each router responds to the TAG DISTRIBUTION message with an acknowledgment. The operator waits to receive an acknowledgment from each router.

To prevent links from being oversubscribed while commitment is in process, testing traffic marked with the PD bit (discussed in Section 4.5) is immediately dropped by routers as of this phase. This is done by adding an output filter rule.

Phase 2: During the second phase, every router knows the tags, so the operator sends a TAG PACKET message to all routers causing them to start marking packets with tags. Since routers do not receive the TAG PACKET message simultaneously, some packets are marked with tags and some use the shadow bit during this phase. Packets generated at the router by a configuration are marked with that configuration's tag, and received packets already marked with tags are forwarded according to the appropriate configuration. Tags are added to packets received without tags: if the shadow bit is unset, it uses the tag of current real configuration (currently C_{old}); otherwise, it uses the tag of the current shadow configuration (currently C_{new}). If a router has not received the TAG PACKET message but receives a packet with a tag, it additionally triggers the router to transit to a state as if it had received the TAG PACKET state. This indirect triggering can speed up this phase.

Before moving to phase 3, the network must wait for the following two conditions

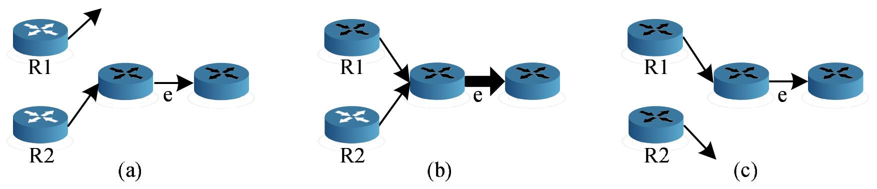

Figure 4.8: Scenario showing how a transient state can cause temporary congestion. White routers have not yet swapped; black routers have swapped.

to become true: (1) no routers are still marking packets using the shadow bit; (2) no packets using the shadow bit are in transit.

At the second half of the Phase 2, the two conditions are satisfied. For the first condition, the operator needs to receive an acknowledgment from each router. After the first condition is true, the operator satisfies the second condition by waiting for a short time (*e.g.*, the estimated upper bound of link latency) until all packets have been processed by the next router in their path.

Phase 3: During the third phase, since no packets will be using the shadow bit, the routers can safely swap the configurations. The operator transmits a SWAP message to the routers. Each router swaps the real and shadow configuration after receiving the message, and sends an acknowledgment back to the operator. Note that the tags associated with each configuration are *not* swapped. Also note that ingress routers that have received the SWAP message now tag unmarked packets with C_{new} instead of C_{old}.

Phase 4: In the last phase, the operator sends a MARK SHADOW BIT message to each router, allowing them to again mark packets using the shadow bit. To report success, each router sends an acknowledgment back to the operator.

4.6.3 Error Handling and Rollback

There are potential error conditions during commitment. Link or router failures cause the routing and forwarding processes (*e.g.*, fast rerouting) to automatically start to react and bypass the failed equipment. The presentation below is focused on error conditions leading to the disruption to our commitment protocol.

Transient States: We define a transient state as a state where some data packets use the old configuration and others use the new configuration. A potential pitfall of a transient state is that the utilization of some links may be higher than it would be under either of the two configurations. Consider an example shown in Figure 4.8. Routers R_1 and R_2 will both change forwarding paths in the new configuration. In Figure 4.8(a), neither has swapped and only R_2 forwards through link e. After R_1 has swapped in Figure 4.8(b), the link is used by both routers, possibly causing temporary congestion. Once R_2 swaps in Figure 4.8(c), the transient state ends and the final router is no longer using link e. Note that such transient states also can happen under some circumstances with other approaches such as the map dissemination approach [61].

Recovery and Rollback: During phase 1, if any one router reports an error or the controller does not receive acknowledgments from all routers, the commitment should abort. As a soft state design, if a router does not receive TAG PACKET before its local timeout, it should change back to the normal state. During phases 2 and 4, if the operation of any router is unsuccessful or times out, the operator will retry the phase. Routers can remain in their current states, as this is not a transient state. It is straightforward if the operator chooses to rollback to the original configuration since the tags are already distributed and only phases 2, 3, and 4 of the protocol need be executed.

The only phase in which a transient state can happen is phase 3. Here, it is important for the state not to be permanent. Consider what can happen during phase 3. If acknowledgments are received from all routers, the transient state has already ended and no rollback is necessary. If at least one acknowledgment is missing, there are two possible reasons: a router did not receive or process the SWAP command, or the SWAP is processed but the acknowledgment was not delivered. We would like to detect the first case. Since an error may have occurred at such routers (*e.g.*, a routing process crashed), it may not be possible to query them directly. Thus, the operator queries the router's neighbors. If the router in question is tagging its forwarded traffic (recall that only real packets are present) with C_{old}, then there exists a router that has not processed the SWAP message, and the srnet should rollback. Note that even if a router crashes during commitment, both the real and shadow configurations of other routers within the srnet reconverge appropriately.

Proposition 1 (Safety)**.** *Packets never alternate back and forth between configurations. Thus, the commitment protocol does not create any additional forwarding loops. Also, control packets such as route advertisements are delivered correctly even while the commitment protocol is executing.*

Proposition 2 (Liveness)**.** *If for every router, commitment control messages are delivered in finite time, and the router either responds to the messages or is recovered offline in finite time, the srnet returns to normal operation, and the transient state is no longer present.*

4.7 Implementation

To demonstrate feasibility, we have implemented a fully operational system supporting shadow configurations. We now discuss in detail Layers 1 and 2 of our system architecture. The related components are shown in Figure 4.1.

4.7.1 Objectives

There are three primary objectives fulfilled by our implementation: (1) identify operating system configuration entities with the shadow and/or real configurations; (2) keep CPU and memory overhead low by merging configuration entities where possible; (3) reduce code changes (*e.g.*, to routing processes and network tools) after introducing shadow configurations.

4.7.2 Supporting Shadow Configurations

A key issue in implementing support for shadow configurations is associating entities maintained within the operating system (*e.g.*, FIB entries, filtering rules, interfaces, neighbor entries, and packets) with the appropriate configuration. To demonstrate that this can be done with minimal effort, we present an implementation consisting of extensions to the Linux Kernel (version 2.6.22.9). Our design is able to support both XORP [44] (version 1.4) and Quagga [77] (versions 0.98.6 and 0.99.9) without any source code changes to either software package. Either can be used interchangeably above our shadow-enabled kernel, which illustrates support in heterogeneous environments.

Figure 4.9 shows the major components of the implementation. We emphasize that different routers may choose different implementation as long as the messaging format (*i.e.*, how shadow data packets and shadow control packets are encoded) is standardized.

Separating Configurations: Each entity is associated with a particular configuration. Entities corresponding to the current real configuration are applied to transit traffic and routing processes that communicate with the outside world, while entities corresponding to the current shadow configuration are being evaluated by the

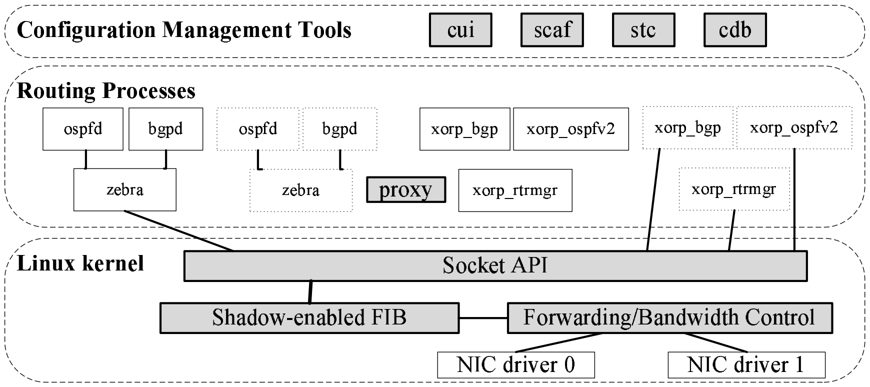

Figure 4.9: Implementation of router supporting shadow configurations: Shaded parts are new or modified.

operator.

We append data structures for necessary entities with a mask, where each bit position corresponds to a particular configuration. If an entity appears in more than one configurations, multiple bits are set in the mask.

One installed configuration is considered as the real while another is considered the shadow. This mapping is maintained in a simple two-entry translation table, allowing the commitment's swap operation to simply swap the entries in the translation table.

Shadow-enabled FIB: We merge entries in FIB table for both configurations to reduce memory overhead. FIB entries use a mask to indicate the configurations to which the destination subnet belongs.

We extend the FIB lookup, insertion, and deletion algorithms to handle the merged FIB table. If the forwarding behaviors (*e.g.*, next hops) in the two configurations are different, we record the difference inside the entry.

Other similar kernel tables, such as neighbor entries, filtering rules, and interface addresses are handled similarly.

Socket API: Extending the kernel tables is not enough. When a userspace program, such as a routing process or a testing tool communicates with the kernel, it uses the socket API. We extend the kernel's socket data structure to reference the configuration to be used when transmitting packets and demultiplexing incoming packets. Routing processes in different configurations can safely bind to the same IP addresses and ports.

Packets: Our current packet format supports IPv4 and ARP, but the same methodology can be applied to IPv6 or other Layer 3 protocols. During normal operations, each packet needs two bits: a shadow bit S, and a PD bit to indicate whether the payload can be dropped. Both S and PD are always 0 for transit traffic. For IPv4 packets, S uses the low bit of the version field, and the PD uses the unused flag bit. Such a mapping causes shadow packets to be automatically dropped by routers that are not shadow-aware. Two additional bits are needed during commitment: TP indicates whether a tag is present, and TG indicates the tag. We store TP in the highest bit of the TOS field and TG in the next highest bit. We use the highest four bits of the ARP header's operation field to mark ARP packets. Note that it is also possible to encode some or all of this information in a shim header.

Packets received by the kernel are demultiplexed according to the translation table (and the tag assignment during commitment). A reference to the appropriate configuration is stored in the packet's data structure for usage in key parts of the TCP/IP stack such as the routing cache and FIB lookups, ICMP errors, and UDP/TCP demultiplexing.

Shadow-aware Programs: Since we also would like provide support for existing programs, we allow a default configuration to be defined for a process, and the attribute is inherited by child processes. Sockets created by a process initially belong to the process's default configuration. We can then launch any program within the desired configuration.

A shell is started for each configuration to enable an operator to apply changes to a particular configuration. The shell indicates whether its configuration is currently defined as the real or shadow.

Routing Processes and Tools: In most implementations, routing processes are normal user processes. Changing networking configurations in the Linux kernel is primarily done using `netlink` sockets. By starting a routing process in the appropriate shell, its sockets are associated with that configuration and the kernel interprets the changes to entities as applying to that configuration. We configure Quagga and XORP such that two instances can be running concurrently, allowing both a shadow and real configuration to be deployed.

The same technique is applied to common network testing tools such as `ping`, `traceroute`, and homegrown scripts, allowing them to operate without modification. We use this approach with our custom traffic generation program and measurement program used in our evaluations.

It is possible that some vendors add shadow-awareness directly to userspace processes (*e.g.*, to use a shared RIB to further reduce memory overhead or supporting additional features in traceroute), while others may want to reduce code changes.

Connection to Outside: Our implementation uses proxies to handle control plane connectivity to outside of a srnet. Such connectivity is necessary to support incremental deployment and interdomain scenarios. These simple proxies can handle not only normal operations but also shadow commitment.

Consider the example of eBGP. Suppose without shadow configurations a BGP routing process b has a BGP peer e in another domain; that is, b has a TCP connection at port 179. With shadow configurations, corresponding to b, there may be two BGP processes b^r and b^s for the real and shadow configurations. We introduce a proxy b^p for b. Then b^p peers with the external BGP peer e (by listening at the IP address and BGP port 179). The process b^p forwards each incoming BGP message from e to both b^r and b^s, which can then apply its ingress filtering policies. Whenever b^r sends a BGP message to e, it is forwarded to b^p which forwards to e.

We use a novel transaction rollback technique to handle commitment with visible external effects. Specifically, the proxy keeps a log of forwarded messages. Whenever b^s sends a BGP message to e, it is stored locally by b^p. If the network swaps the real and shadow configurations, b^p computes the differences of the messages of b^r and b^s, rolls back the unnecessary impacts of b^r (*i.e.*, withdraw different routes), and then installs the effects of b^s without disconnecting the external BGP connection.

Shadow-aware Interfaces: It is necessary for routers to drop shadow packets and remove tags from transit packets (in the case of commitment) before exiting an srnet. We enable a *shadow-aware* attribute on each interface that participates in the srnet.

Since our evaluation environment utilizes ARP, there is one additional complexity during commitment. Egress traffic should not be delayed or possibly dropped while it waits for the new configuration to query for the MAC address of the peering router outside of the srnet. Thus, we configure the kernel to accept unsolicited ARP replies and duplicate any received ARP reply to the shadow configuration for interfaces with the *shadow-aware* attribute disabled.

4.8 Evaluations

We first present our methodology, then present our results in two parts. In the first part, we present results that show that the overhead of supporting shadow configuration is very small. In the second part, we demonstrate the effectiveness of shadow configurations in three usage scenarios.

4.8.1 Methodology

Implementation: We use our implementation as described in Section 4.7.

Configurations: We use the configuration files of the two operating networks in Table 4.1. US-ISP is a large US tier-1 ISP.

Network	#Nodes	#Directed Links	Syntax
Abilene	9	26	Juniper
US-ISP	-	-	Cisco

Table 4.1: Network configurations used.

We use the configurations of US-ISP only for evaluation of FIB size overhead. The rest of our experiments use a small illustrative topology and an emulation of the Abilene backbone. We use Emulab's [97] 3 Ghz PCs with 1 Gbps and 100 Mbps Ethernet links. We take additional steps to load configuration data into our emulation of the Abilene backbone. Configuration commands are translated to both XORP and Quagga syntax. Then BGP routes from Abilene's July 2007 BGP RIB dumps are injected as static routes at virtual egress points, `dummy0` interfaces, at the appropriate routers. Routes for the University of Utah are removed so as not to interfere with the Emulab addresses configured on the routers. Since the versions of XORP and Quagga used did not support IS-IS, we translated Abilene's configurations to use OSPF.

Data Traffic: We use CAIDA [15] packet traces in our evaluations. When using these traces on our emulation of the Abilene backbone, we remove packets for destination addresses not appearing in the BGP routes accepted by Abilene.

Performance Measurements: To obtain performance measurements under packet cancellation, we use a custom utility similar to `iperf` that timestamps generated packets just following the IP header and sends using raw sockets. The timestamp is not lost during packet cancellation. We modify the kernel to deliver canceled packets to raw sockets. The server computes delay between sending and receiving time, and uses linear regression to subtract off mean delay and account for clock drift.

4.8.2 Overhead

Since we intend that shadow configuration be used in production networks, the overhead of supporting it should be small. One reason we chose Linux is to see the overhead in a general platform. We consider (a) data path forwarding overhead due to additional complexity to support a shadow configuration; (b) FIB storage overhead due to addition of a shadow configuration; and (c) FIB update overhead due to addition of a shadow configuration.

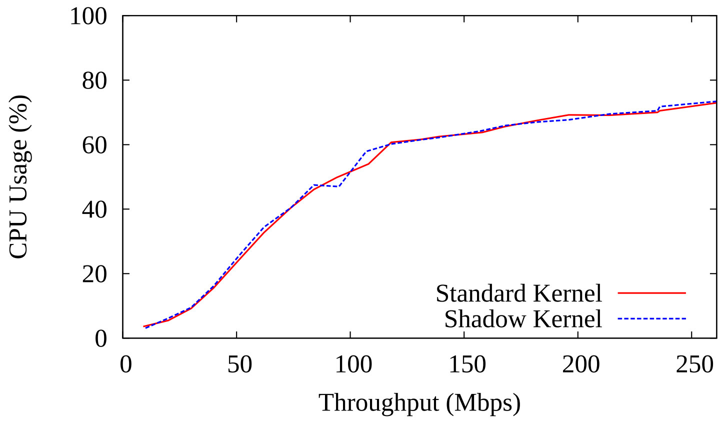

Figure 4.10: System CPU utilization for varying traffic rates (300-byte packets).

Data Path Forwarding Overhead: Our results show that there is truly a negligible overhead on the data forwarding path due to the additional complexity of supporting a shadow configuration. For this test, we use a particular traffic load both with the standard Linux kernel, and then again with our shadow-enabled kernel. When employing our shadow kernel, we load a shadow configuration but do not generate shadow traffic.

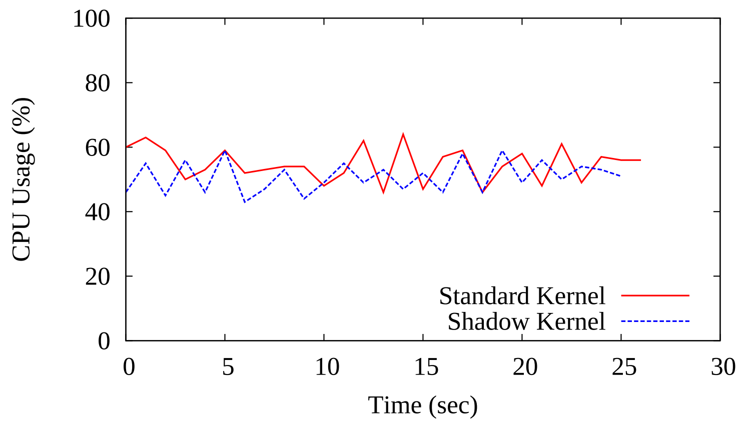

Figure 4.11: System CPU utilization for FIB updates (100 Mbps, 300-byte packets).

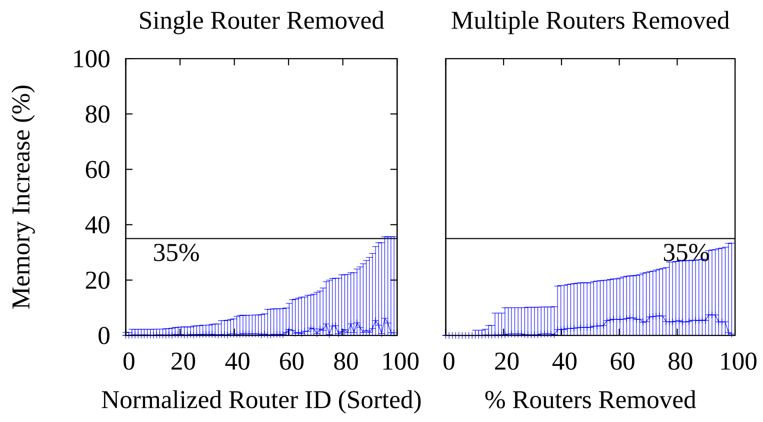

Figure 4.12: FIB storage overhead for topology changes in shadow configuration (US-ISP).

We use a topology with 3 routers with 1 Gbps Ethernet links; there is a sending, intermediate, and receiving router. The sending router uses the Linux kernel's `pktgen` module to generate 300-byte packets so we can stress-test the intermediate router's forwarding path. Our implementation doesn't use any additional memory copies for real packets, so larger packet sizes do not add overhead in our shadow kernel.

The sending router transmits packets for 30 seconds with randomly generated destination IP addresses in the range 10.0.0.4-10.255.255.255 to ensure that FIB lookups (on the intermediate router) are rarely handled by the routing cache. The intermediate router configures one default route for 10.0.0.0/8 to route to the receiving router, and also adds additional 9306 randomly generated entries from 10.0.0.0/8 with a prefix length distribution matching the global BGP tables published by the Route Views Project on January 18, 2008 [89]. Note that there are no prefix lengths shorter than 8. Also, 9293 routes are added in shadow configuration, with 60% of the prefixes shared with the real configuration.

The comparison in CPU utilization between our shadow kernel and the standard kernel are shown in Figure 4.10. The machines are hyperthreaded, so we increase the data rate until the CPU handling the input interface interrupts reaches 100% utilization. The reported value is the overall CPU utilization including both CPUs. Our implementation does not noticeably increase CPU utilization as compared to the standard kernel. Note that the full 1 Gbps capacity is not reached due to the smaller packet sizes. Our implementation can achieve the same rates cited by [10] for 1430-byte packets.

FIB Storage Overhead: One concern is that the number of FIB entries will be increased. However, for most networks, the network prefixes are relatively fixed, and thus should appear in both real and shadow configurations. Accordingly, the number of entries and IP prefix lookup costs do not increase significantly. We incur a storage overhead only if the shadow and real configurations specify different next-hop behaviors, since otherwise only a single FIB entry is required.

Scenario	Changed FIB Entries	Memory Overhead
Remove NEWY↔WASH	13074	4.7%
Remove LOSA	4467	1.6%
Remove KANS	19874	7.2%

Table 4.2: FIB storage overhead (Abilene).

Table 4.2 shows the increase in FIB size across all routers for the Abilene network due to configuration changes made in the shadow configuration. Both the real and the shadow configurations have more than $90,000$ FIB entries. We observe these topology changes lead to a small overall storage increase. Although in the theoretical worst case the storage may double, in our real implementation the increase is less than 8% due to the sharing between real and shadow next-hops. We anticipate that this sharing is common.

To evaluate the scenarios for a larger network, we use the configuration of US-ISP, a large tier-1 ISP. We use its backbone topology, OSPF link weight configuration, and external routes to compute the FIB size at each router. Each router has a few hundreds of thousands of FIB entries. The presented memory overhead is based on data structure sizes in the Linux kernel implementation.

Figure 4.12 shows the results for two scenarios. The vertical bar denotes the maximum and minimum per-router memory overhead, and the dark points denote the average memory overhead over all routers. In the first scenario, we show the memory overhead when only a single router at a time is removed from the network in the shadow configuration. We observe that in the worst cases, the routers with the worst FIB overhead have their FIB storage increased by no more than 35%.

These “worst” routers are often stub routers with low connectivity in the topology. Thus, one way to reduce their storage, if necessary, is forwarding entry aggregation or virtual address mapping. The average is much lower, under 5% in most cases. Next, we show the FIB memory overhead as routers are removed one-by-one in the shadow configuration. There is no case in which the router with the worst FIB overhead has its FIB storage increased by 35%. The average overhead is much lower than the worst case.

FIB Update Overhead: Since we also extend the FIB insertion and deletion routines to handle shadow configuration, we also evaluate the performance when the FIB is being frequently updated. We use the same setup as the prior experiment on FIB data forwarding processing overhead, but we also randomly add and delete between 1 and 100 routes in the real configuration in 10.0.0.0/8 each second at the intermediate router as it is forwarding traffic.

Figure 4.11 shows the results. Again, there is no noticeable difference between supporting shadow configuration or not. Note that when running this experiment without the FIB updates, the CPU utilization for both our shadow kernel and the standard kernel fluctuates much less, but both remain nearly identical for the duration of the experiment.

4.8.3 Usage of Shadow Configurations

We now demonstrate the effectiveness of shadow configuration in three usage scenarios.

Equipment Maintenance: A usage scenario of shadow configuration is equipment maintenance. We use this scenario to demonstrate the performance of our commitment protocol.

In this experiment, we use the Abilene topology and configurations, and generate transit traffic according to the CAIDA traces from peering routers configured at New York, Seattle, and Atlanta. Emulab's delay nodes are used to model propagation delays.

In this scenario, we bring the Kansas router down for maintenance and return it to service when finished. The real configuration is initially cloned to the shadow configuration. Next, we disable OSPF in the shadow configuration on the Kansas router, wait 10 seconds, then commit at time 48. The network operator may then safely perform upgrades, and restart it when finished. Once the shadow configuration with Kansas enabled converges, the configurations are again swapped, causing the Kansas router to again forward transit traffic.

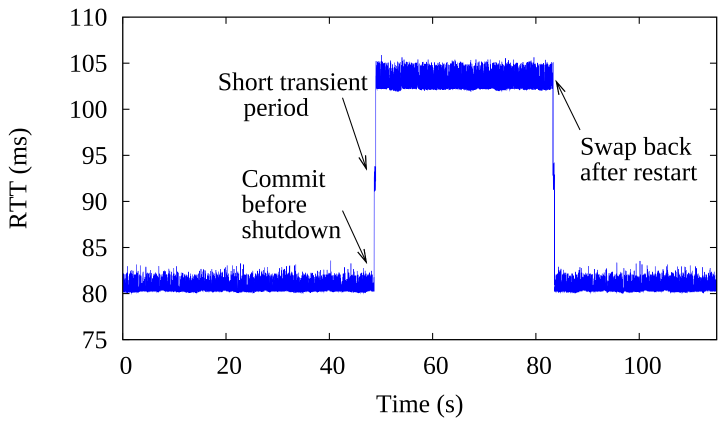

Figure 4.13: RTT between peers at New York and Seattle during commitment and rollback.

Figure 4.13 shows the round-trip time between the peering routers at New York and Seattle. Note that there are three modes of operation at 82 ms, 92 ms, and 102 ms due to the Abilene routers asynchronously executing the swap in phase 3 of the commitment protocol. This arises because the ICMP echo request follows a different path than the reply, due to tagging at the ingress routers. The intermediate transition phase lasts for a short time, but the packet forwarding behavior during this transition phase is clean and controlled. Importantly, there are no packet losses.

Our commitment protocol is executed over serial consoles to each router. We are currently developing a protocol to access the routers' configuration terminals using both the shadow and real configurations such that the protocol is resistant to misconfiguration in one of the two configurations.

Parameter Tuning: The next usage scenario for shadow configurations we evaluate is parameter tuning. We update a set of OSPF link weights simultaneously. The real configuration uses Abilene's normal link weights, and we then change *all* of the link weights in the network to be the inverse of the bandwidth (*i.e.*, all equal in the Abilene case) using two methods: (1) manual configuration and (2) shadow configurations.

To perform the manual configuration, we update the link weights using parallel Telnet sessions, which takes about 4 seconds. With shadow configurations, we update the link weights in the shadow configuration, wait 20 seconds for convergence, and then execute the commitment protocol.

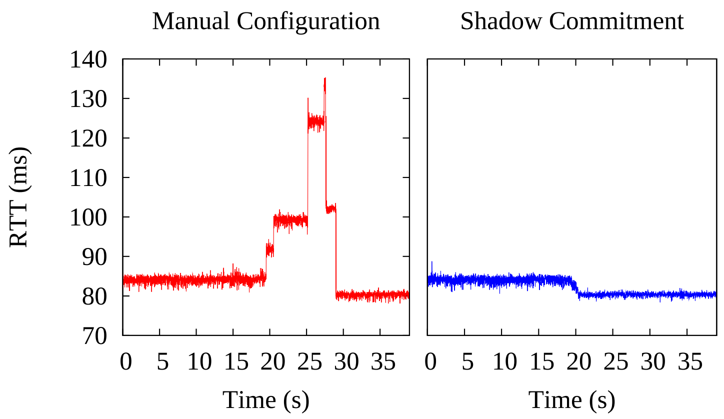

Figure 4.14: RTT between peers at New York and Seattle during OSPF link weight change.

We immediately notice in Figure 4.14 that using the shadow configuration avoids the reconvergence process. Under manual configuration, the round-trip time between the peer routers at Seattle and New York fluctuate between 83 ms and 135 ms before settling on the converged value of 80 ms. Using shadow configurations provides a quick and smooth transition since convergence takes place in the shadow configuration prior to commitment.

New Service Testing: The last usage scenario we evaluate is testing of new services. We use this scenario to demonstrate our packet cancellation technique and show that (1) there is little effect on transit traffic and (2) performance measurements on shadow traffic are indicative of its true performance.

	Real	Shadow
Config	Abilene configuration	Abiliene configuration with 4 link weights adjusted for load balancing
Traffic	Transit traffic generated from CAIDA traces with 30% utilization on bottleneck link	Duplicated real traffic and UDP streaming video with 6 servers and 12 clients

Table 4.3: New service testing experiment setup.

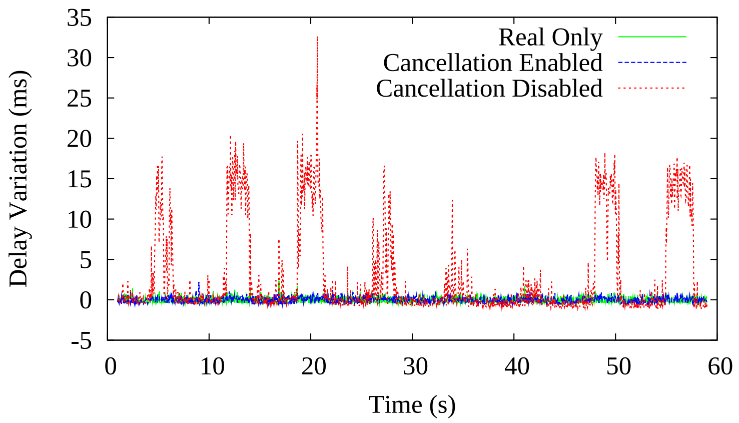

Figure 4.15: Delay variation for real transit traffic (Seattle→Chicago).

Setup: In this scenario, a network operator is testing a streaming video application under a new set of OSPF link weights. Our setup is shown in Table 4.3.

UDP packet traces are constructed using a high-definition movie trailer and the VideoLAN [91] VLC software. The movie trailer alternates between complex scenes (using up to 22 Mbps) and a black background with text (using 450 Kbps).

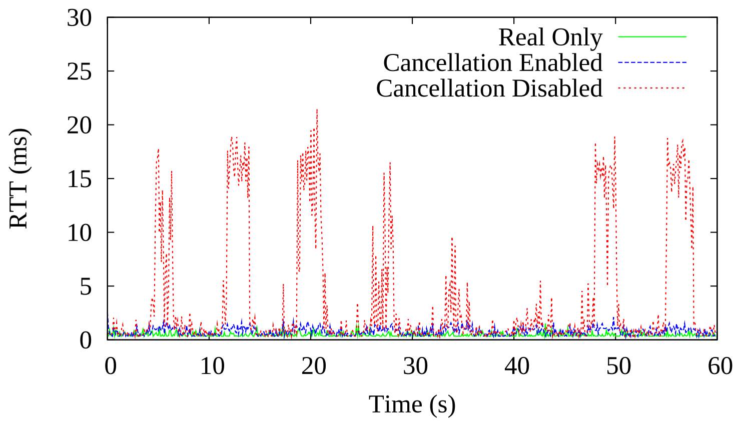

Figure 4.16: RTT in real configuration (Salt Lake City→Atlanta).

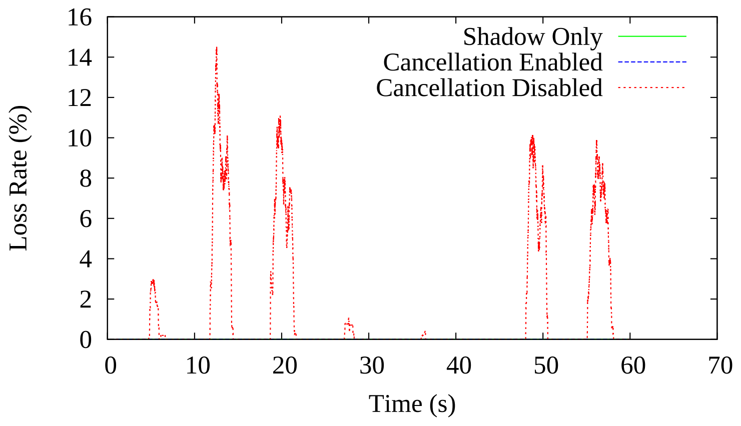

Figure 4.17: Loss rate for streams (Salt Lake City server).

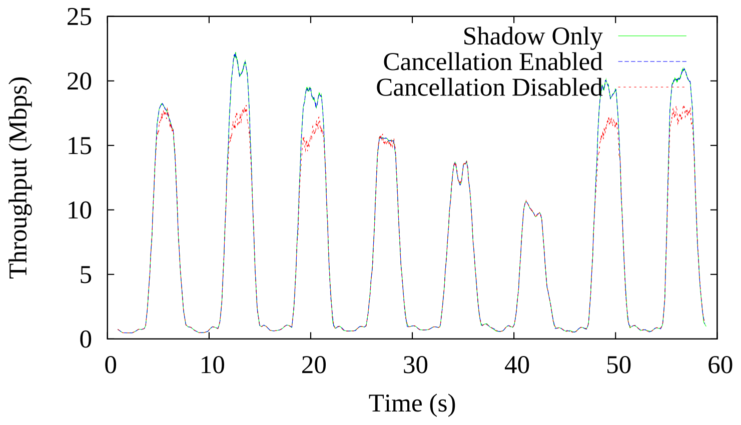

Figure 4.18: Stream throughput (Houston→Chicago).

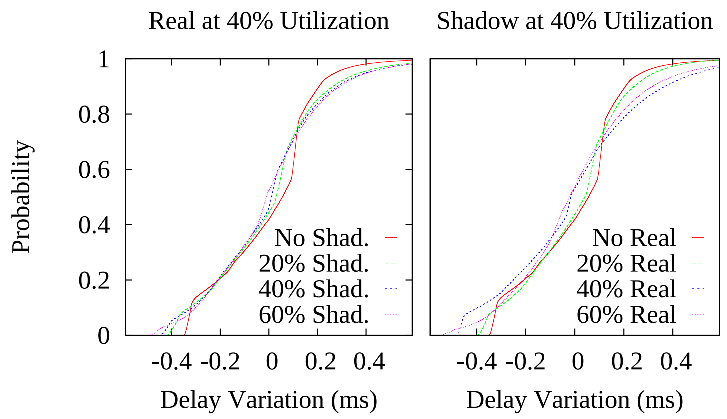

Figure 4.19: Delay variation CDF (illustrative topology).

With this setup, there exist time intervals when the combined (raw) real and shadow traffic intensity exceeds link capacity on some links, meaning bandwidth partitioning is not effective for obtaining accurate performance results.

Delay nodes are removed from the Emulab experiment since we want to observe small-scale variations over multi-hop flows. We also use 100 Mbps links to more easily observe delay variation given the resolution of our measurement tools.

Safety for Transit Traffic: Our experiments show that the shadow traffic has little effect on the real traffic when packet cancellation is enabled. We show the measured performance for two paths. Figure 4.15 shows the delay variation for traffic from Seattle to Chicago. The real traffic performance with packet cancellation enabled overlaps the performance when only real transit traffic is present, while the delay variation rises sharply up to about 15 ms without packet cancellation. Similar behavior is observed between Salt Lake City and Atlanta (Figure 4.16), where the round-trip time increases from under 1 ms up to 20 ms without packet cancellation. Round-trip time is largely unaffected with packet cancellation enabled.

Shadow Performance Accuracy: We next show that packet cancellation provides accurate performance measurements despite the presence of real transit traffic. In our experiment, there are multiple streaming sessions that have incorrect measurements when packet cancellation is not enabled. For example, the throughput measurement for the video stream from Houston to Chicago (Figure 4.18) shows the correct value of 22 Mbps. Without packet cancellation, the measurements incorrectly show that only 18 Mbps is supported.

Multiple video streams in our experiments also show that loss rates with packet cancellation are indicative of the true value. Figure 4.17 shows the loss rate of streams served by Salt Lake City. Without packet cancellation, it is erroneously reported to be up to 14%, while packet cancellation correctly has no losses.

Fine-grained Accuracy: Finally, we show in more detail how real traffic is protected and performance characteristics of shadow traffic are preserved under packet cancellation. We use a simple illustrative topology shown in Figure 4.7 and the CAIDA traces. Figure 4.19 shows CDFs of delay variation for both real and shadow traffic. The observed performance for real traffic is largely unchanged as we increase shadow traffic until raw total traffic intensity reaches link capacity (100%). Similarly, delay variations for shadow traffic closely approximate its actual behavior.

4.9 Summary and Future Directions

In this chapter, we presented the novel idea of shadow configurations. We developed novel techniques such as packet cancellation and shadow commitment to substantially improve the capability of configuration evaluation and management. There are many avenues for future exploration. In particular, a future direction is to more fully integrate shadow configurations with automated configuration debugging and network diagnosis.

Chapter 5

PEAC: Dual System for Internet Live Streaming

5.1 Background and Motivation

Live streaming distribution is an important class of Internet applications. These applications have been used to carry not only many daily events but also major events such as the Obama inauguration address, the 2010 Winter Olympics, and the 2010 World Cup.

5.1.1 Complexity of P2P Live Streaming

To satisfy increasing live streaming scale as well as user requirements on new features, many live streaming distribution systems have become increasingly complex. As an example, many modern live streaming systems are either P2P based or add P2P as a key feature to improve scalability and fault tolerance (*e.g.*, PPLive, CNTV, ChinaCache LiveSky, Zattoo). The latest release of Adobe Flash [1], a major platform for streaming distribution on the Internet, has introduced Stratus, a P2P data exchange mode. However, a P2P based system consists of a large number of algorithmic components and configuration parameters that interact in complex ways.

As a result of increased complexity, many live streaming distribution systems have become increasingly difficult to understand, often operate at sub-optimal states, and exhibit undesirable, buggy behaviors in real settings. Many live streaming distribution systems release new versions that do not reach anticipated the performance level or scale. For example, PPLive, a major live streaming distribution system, encountered major performance issues when moved from initial deployment in university networks to general network settings. As another example, a major category of user complaints in the user forum of a major live streaming system [76] is poor performance after a major product release.

5.1.2 State of the Art

Realistic evaluation is essential to the understanding and improvement of Internet live streaming distribution systems. During our survey, a major streaming software developer commented that "a major factor delaying our product release is concerns on lacking of realistic testing capabilities." However, existing testing techniques such as theoretical modeling, simulations, and lab testing (*e.g.*, [12,13,60,63,74,102,104]) all have limitations for conducting realistic performance evaluation of an Internet-scale, distributed streaming system. Specifically, lab testing is often severely limited in scale. No testbed can easily scale above thousands of clients. Theoretical modeling, lab testing, and simulation may deviate from reality, for example, in terms of user distributions, capabilities, and behaviors. Furthermore, these techniques lack the ability to provide a platform to discover previously-unknown but important, real problems. For example, network management policies implemented by some ISPs (*e.g.*, Comcast PowerBoost [20] and congestion management [9]), large hidden buffers (*e.g.*, large ADSL buffers [24]) at last hop of some real networks, and poor peering between some ISPs [58] may all have large impacts on performance. On the other hand, the theoretical model, simulation model, or the lab setting may not include such factors until they are discovered, much later, in real tests.

In current practice, some live streaming systems (*e.g.*, PPLive, UUSee) use testing channels to achieve scale (to a setting larger than their internal small lab) and realism. Real users joining a testing channel use a different set of experimental algorithms than the stable algorithms running on other channels. However, the testing channel approach encounters two important, practical problems. First, passively using real users may not provide effective performance evaluation settings. For example, a developer may want to evaluate the effectiveness of a set of experimental algorithms to handle a flash crowd on the order of 1000 users, but the current testing channel, which may have a higher number of real users, say 10,000 users, does not give a flash-crowd arrival pattern. Second, a major concern of utilizing testing channels is that if the experimental algorithms do not perform well, users will experience poor quality. This concern leads to conservative usage of testing channels, thus limiting tests to a small number of real users, reducing the effectiveness.

5.1.3 PEAC Overview

In this chapter, we present PEAC a novel Internet live streaming distribution system that integrates performance evaluation as an intrinsic capability, during production live streaming, to take advantage of the availability of a large number of real user clients located at real network settings. It integrates realism and experiment control by creating experimental scenarios using real user clients. It also provides highly scalable disruption protection for real users' viewing experience if the experimental algorithms do not perform well. PEAC complements analysis, simulation, and lab testing to provide a more complete experimentation framework.

PEAC achieves the aforementioned objectives by utilizing the Dual-System Architecture. In particular, the production system is the *stable* system, which is the existing system with reasonable performance and typically exists in a continuously evolving system, and test systems called *experimental* systems to be evaluated.

Specifically, real users are first served by the stable system during the *staging phase*, during which real users begin playing while the experiment control system waits on the triggering condition to start developer-specified testing scenarios (*i.e.*, targeted arrival and departure user behaviors and the experimental algorithms). After the triggering condition is met, the experiment control system starts the *testing phase* by *transparently* moving a set of clients already in the stable system to construct experimental scenarios of clients to be served by the experimental algorithms. The experimental control system creates experimental scenarios using distributed algorithms, and requires only light-weight soft-state information at trackers. The experiment control system handles peer dynamics and uses the stable system as a scalable control channel with fast feedback, to detect user-initiated departures and peer failures.

A key challenge in the testing phase is how to protect the experiences of a large number of real users running experimental algorithms. A possible scheme to protect user experiences is to use CDN or supernodes, when the experimental algorithms in a new version do not perform well (*e.g.*, [42]). However, without considering constraints such as the data flow constraints (Section 5.5.3), existing systems provide only protection, not experimental accuracy. While PEAC supports protection using the CDN with proper consideration for these constraints, a shortcoming of using CDN protection, is that such sources may not always be available, be costly, or not provide enough scalability during a large-scale experiment. For example, if using only a CDN, a trial involving 50,000 real users in a channel at 400 Kbps will need up to 20 Gbps bandwidth. If the modified feature is shared by multiple channels in a streaming system with 2 million concurrent users, CDN load can be as high as 800 Gbps. PEAC again utilizes the dual system design. Instead of relying on a CDN or supernodes, PEAC triggers the stable system as the first line of rescue, substantially improving scalability.

Implementing a dual system during the testing phase, however, introduces technical challenges. Specifically, with both stable and experimental systems running, a dual system is faced with the *dual-system resource and task allocation problem.* If all resources (*i.e.*, bandwidth) and tasks (pieces to be downloaded) are assigned to the stable system, the dual system is guaranteed to achieve the performance of the stable system. However, there will be no performance evaluation results on the experimental system. On the other hand, if all resources (*i.e.*, bandwidth) and tasks (pieces to be downloaded) are assigned to the experimental system but the experimental system performs poorly, the stable system cannot provide effective rescue as it does not have resources and assigned tasks. To address this challenge, PEAC introduces a novel scheme named *Adaptive Task Reassignment* to implement resource and task assignments, achieving robust protection and experimental accuracy at the same time.

We completely implement PEAC and demonstrate its benefits. Our implementation addresses an important practical concern of software architecture supporting flexible experiments with real users. PEAC introduces a Compositional Runtime to achieve efficient code and data sharing and provide online client reconfiguration at end-user machines.

5.2 Motivating Use Cases

PEAC is a streaming distribution system supporting live streaming. A developer of PEAC can use the built-in experimentation capability to conduct many experiments in the production system. We present three use cases spanning software testing process, scientific evaluation and parameter tuning, and new algorithm design of large-scale network environments.

Regression Test Suite for User Performance: The developer of a live streaming system defines a testing suite of user behavior scenarios such as flash crowd, gradual arrivals, and steady state, at different scales (say, 100s, 1,000s, 10,000s users). After developing a new version of the software, the developer should conduct regression testing, based on the testing suite, under real settings, to ensure that performance issues are not introduced even by seemingly-unrelated changes. For example, even changes to NAT (Network Address Translation) traversal may have impacts on streaming performance. PEAC creates the scenarios defined in the testing suite, using user clients in the production environment, along with thresholds for acceptable performance. These experiments may be enqueued and executed by PEAC to ensure that the newer version has expected performance in a wide range of large-scale, real environments before being officially released.

Parameter Tuning: A live streaming system can include many parameters (*e.g.*, timeouts, upper/lower bounds for rate control, and maximum number of peers). PEAC allows the designer to explore the parameter space in real settings by defining an experiment with multiple scenarios to be executed in parallel for comparison (*e.g.*, [40]), and/or perform factor analysis to study the effects of different parameters.

Algorithm/Feature Testing: A live streaming system can include many algorithmic modules such as the rate allocation algorithm (*i.e.*, how a peer allocates its bandwidth to neighbors), the piece selection algorithm (*i.e.*, which piece to download first), and the topology management algorithm (*i.e.*, which peers connect to which peers). It may also include other algorithmic modules to integrate with other techniques such as network-friendliness improvements (*e.g.*, [5, 99]), shared bottleneck usage (*e.g.*, [54]), or flash-crowd admission control. PEAC enables one to test, in real settings, the performance of different algorithms and new features across a general population or in specific settings.

5.3 Overview

We now give an overview of PEAC. We start with the scope and basic workflow of PEAC. Then, we introduce the key components of PEAC.

5.3.1 Setting and Scope

PEAC is a hybrid live streaming system using both P2P and CDN. In particular, PEAC is a piece-based system, in which the source divides streaming contents into pieces to distribute to clients. A client downloads pieces predominantly using the peer-to-peer mode.

PEAC focuses on performance evaluations in real settings with real users and complements existing testing methods. We assume that the experimental subsystem is tested in lab settings such as unit tests to fix correctness errors and program crashes, and provide repeatability.

5.3.2 Experiment Workflow

A single PEAC experiment consists of one or multiple *scenarios*, where each scenario is defined by the *peer behavior configuration* and the *code* that will be executed by the peer running the scenario. Multiple scenarios in a single experiment are executed in parallel. This is typical when developers want to compare the relative performance of different algorithms or user behaviors.

A PEAC experiment proceeds in three phases:

- *Definition:* In this phase, the developer defines the experiment *scenarios* to be executed, and places the experiment in the *experiment queue*.
- *Staging:* The set of available peers in the target channel is monitored for the feasibility of generating the pending experiment scenarios. During this period, the code and peer behavior configurations supporting the experiment are distributed to peers.
- *Testing:* The feasibility condition is triggered, and a set of peers (transparently) transition to use the experimental algorithm(s) at times according to the peer behavior configuration. Note that during testing, the experimental subsystem is treated as a black box.

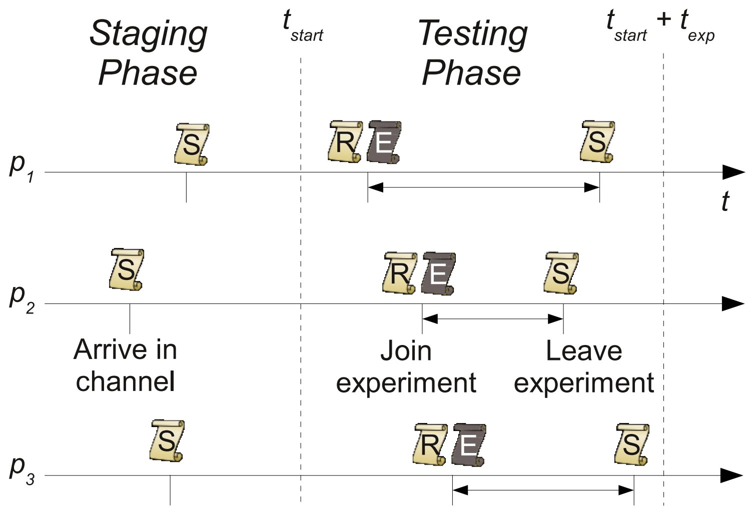

Figure 5.1: PEAC experiment scenario timeline.

Figure 5.1 shows an example timeline for a single experiment scenario with three clients. Clients begin by using only the stable system (labeled by S), but transition to using the rescue and experimental subsystems (labeled by R and E, respectively) upon joining the experiment. If the stable system is used as the rescue, it simply continues to run and assumes the role of the rescue subsystem.

5.3.3 Components

PEAC experiment workflow is implemented by components shown in Figure 5.2:

- Experiment Definition and Control: This component manages and monitors the experiment lifecycle to initiate and gather results from experiments. Experiment scenarios, including peer behavior configurations and any necessary code, are distributed to peers with the help of trackers, a CDN, or the P2P overlay itself.
- Compositional Runtime: The Compositional Runtime enables easy, on-the-fly deployment and reconfiguration of the algorithmic components in each client participating in an experiment.
- Resource/Task Scheduler: As we discussed in Section 5.1, PEAC uses a dual system to provide scalable protection and experimental accuracy at the same time. The scheduler controls task and resource allocation among a rescue and experimental subsystems when they execute concurrently.
- Media Player: The media player combines the video stream from multiple running subsystems and displays it to the user.

5.3.4 Experiment Definition and Control

A key component of PEAC is its Experiment Definition and Control (EDC) component. Although real user clients located at real networks provide a high degree of realism, passively using real users may not provide key desired performance eval-

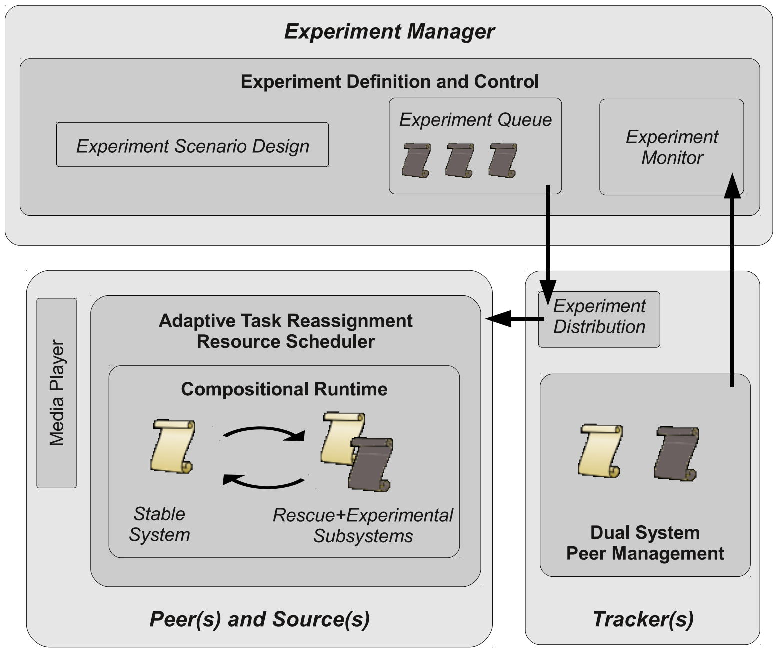
Experiment Manager
Experiment Definition and Control
Experiment Scenario Design
Experiment Queue
Experiment Monitor
Experiment Distribution
Media Player
Adaptive Task Reassignment Resource Scheduler
Compositional Runtime
Stable System
Rescue+Experimental Subsystems
Dual System Peer Management
Peer(s) and Source(s)
Tracker(s)

Figure 5.2: PEAC system architecture.

uation settings such as a flash crowd. Thus, EDC allows a developer to specify a peer behavior configuration defining the desired user behaviors: the peers selected to be included in the scenario; the selected peers' arrival behavior; the selected peers' lifetimes, and user behavior in response to video quality.

A key challenge of implementing EDC is scalability; we would like to support channels and experiments with a large number of users. One way to implement EDC is direct tracker control. This is the approach taken by some experimental platforms (*e.g.*, [62, 87, 97]), in which one or more controllers issue commands at appropriate times to each peer to let it join or leave the test. While this approach may work in certain settings, operating with real P2P peers is more challenging due to increased controller load and complexity. For example, since between 60-80% [47] of users are behind gateways using NAT, the controller must implement frequent keep-alive messages between the controller and peers to keep the connection open only in preparation for commands issued in the future.

In addition to supporting the direct server control mode, PEAC also introduces a second mode called *Distributed Scenario Control.* This mode decouples the scenario parameters from their execution, and thus relaxes the requirement that control messages are delivered to each peer within a small delay, in contrast with direct control approaches. In particular, the scenario parameters can simply be broadcast to each peer via existing mechanisms (*e.g.*, piggybacked on existing messages, redistributed using the P2P overlay or a CDN). Then, each peer *locally* decides and controls its own arrival and departure times within a scenario. More details on Distributed Scenario Control will be given in Section 5.4.

5.3.5 Resource/Task Scheduler

A key novelty of PEAC is that it uses a dual system in which both experimental and rescue subsystem run parallel during an experimental trial. The clear semantics of Internet live streaming makes it possible to merge the outputs of these two subsystems before presenting to the users.

A key challenge in supporting large-scale live streaming evaluation with dual systems, as we discussed in Section 5.1, is the resource and task assignment problem. The solution to this problem can have a major impact on (1) the robustness to protect user's experiences; and (2) experimental accuracy.

Problem Formulation: To precisely state the requirements of the design of the resource and task scheduler, we introduce some notations. Let S denote the existing stable system, E an experimental system being evaluated, and C the algorithms for fetching from a CDN or supernode.

The resource/task scheduler controls the allocation A of tasks and resources to the subsystems. In particular, the behavior of an experimentation subsystem over time $T = [t_1, t_2]$ is dependent on both the allocated download tasks and resources.

Let D=[$D_S(t)$, $D_E(t)$, $D_C(t)$] denote the download tasks (pieces) assigned at time t to the stable, experimental, and CDN algorithms, respectively. Likewise, let R=[$R_S(t)$, $R_E(t)$, $R_C(t)$] denote the resources (*e.g.*, bandwidth, connections, CPU, and memory) allocated to each at time t. Then $A = [D, R]$ denotes the combined experimentation control behavior.

The performance of resource/task allocation A is compared with two benchmarks. Let A^S denote the allocation that all tasks and resources are allocated to the stable system; A^E the allocation that all tasks and resources are allocated to the experimental subsystem.

Requirements: Then the objective of PEAC is to design an allocation A satisfying two requirements. The first requirement (R1) is to maintain user experience. Given a performance metric $Perf()$ defined such that higher values indicate better performance, PEAC requires:

$$Perf(A) \geq Perf(A^S), \tag{R1}$$

for performance metrics affecting the streaming quality visible to users. In particular, the overall system should perform at least as well as (or possibly better than) the stable system alone.

The second requirement (R2) is to obtain accurate performance measurements from the experimental subsystem. In particular, this requirement can be stated as:

$$\text{obtain } Perf(A^E) \text{ from } Perf(A). \tag{R2}$$

It is important to note that real live streaming systems are not deterministic systems in that the performance metrics are random variables. Thus, the objective of

PEAC is to collect samples of performance metrics and evaluate on the distributions.

A major challenge in satisfying (R2) is that the rescue subsystem, in particular the stable system as a rescue subsystem, may cause interference to the experimental subsystem. Consider a simple example that when the experimental subsystem has all of the clients' uploading capacities, it can perform well. However, the resource/task scheduler may trigger the rescue subsystem as rescue too early and allocate resources to it. Then the accuracy of the experimental trial is lost.

Issues with Fixed Allocation: One might think that a simple, intuitive task and resource allocation scheme is *fixed*, proportional allocation to provide isolation. Specifically, in a simple, homogeneous proportional allocation scheme, the experimental subsystem is allocated α fraction (say, 20%) of the resources and α fraction of the tasks. A major advantage of this scheme is simplicity. Given this scheme, it may be cleaner to design and conduct capacity planning for the rescue subsystem.

A major issue of the proportional allocation scheme, however, is to achieve (R2). Let $Perf(A^{\alpha E})$ denote the performance of the experimental subsystem allocated with α fraction of the tasks as well as α fraction of the resources. A streaming system is said to be a scale-invariant streaming system for metric $Perf$ at α if:

$$Perf(A^{\alpha E}) = Perf(A^{E}), \qquad \text{(Scale-Invariant)}$$

for the given α.

Although one can derive sufficient conditions for a system to be scale-invariant, not all systems are scale-invariant. One example setting that is not scale-invariant is that there are bottlenecks in the internal networks. In addition, algorithms such as slow-start may need modifications to support fixed allocations. To allow PEAC to support more general experimental algorithms, we propose a more adaptive allocation scheme named Adaptive Task Reassignment to satisfy both (R1) and (R2). Adaptive Task Reassignment is presented in detail in Section 5.5.

Rescue Source Considerations: There are systems that use CDN as a rescue source in a test channel (*e.g.*, [42]). However, CDN may not be scalable or be more costly. Also, without considering constraints such as the data flow constraints (Section 5.5.3), such systems provide only protection, not experimental accuracy.

5.4 Distributed Scenario Control

We now give more details on Distributed Scenario Control. Recall from Section 5.3.4 that the objective of Distributed Scenario Control is to create an experiment where peers decide and control their arrival and departure locally. It may first appear that achieving the defined behavior requires global coordination. However, Distributed Scenario Control's design allows each peer to *independently* decide if it should participate in the experiment scenario, and if so, at which time it should become active (begin running). This improves scalability of experiment control with a large number of peers.

Below we present the distributed algorithms. We focus on the peer behavior configuration for a single scenario in an experiment; the algorithms are easily extended to support multiple scenarios in executing in parallel. We first focus on peer arrivals and departures. We then present the triggering condition to start the arrival process. We present the algorithm assuming that users do not leave while the experiment scenario is in progress. Extensions to handle early, user-initiated departures are in Section 5.4.3.

5.4.1 Controlled Arrivals and Departures

Distributed Scenario Control implements a flexible peer arrival and departure scheme. Peer starts to arrive (*i.e.*, transition) into the experimental systems after triggering conditions are met. We first discuss the arrival/departure generation algorithm, and in the next subsection discuss the triggering conditions.

Controlling Arrivals: Since real-world peer arrivals may be time varying due to time of day or event schedule effects, the peer arrival rate (peers/second) will be a function of time.

Problem definition: Given a global arrival rate function $\lambda(t)$ on the interval $[0, t_{\text{exp}}]$ (the duration of the scenario), we devise an algorithm such that each peer i independently computes its own arrival time $a_{\text{e},i}$, given the scenario start time t_{start}, $\lambda(t)$, and $[0, t_{\text{exp}}]$. Note that the arrival time here is the time that the peer switches to the experiment system. Also note that the total number of peers to participate in the scenario is implicitly included in this definition of the arrival behavior.

To generate non-homogeneous arrivals specified by rate function $\lambda(t)$, we apply the following theorem [21, 75]:

Theorem 1. *Let $T_1, T_2, \ldots$ be random variables representing the event times of a non-homogeneous Poisson process with continuous expectation function $\Lambda(t) = \int_0^t \lambda(x)\,dx$, and let N_t represent the total number of events occurring before time t in the process. Then, conditional on the number of events $N_{t_{\text{exp}}} = n$, the event times $T_1, T_2, \ldots, T_n$ are distributed as order statistics from a sample with distribution function $F(t) = \frac{\Lambda(t)}{\Lambda(t_{\text{exp}})}$ for $t \in [0, t_{\text{exp}}]$.*

An implication of Theorem 1 is that we can generate arrival times by drawing random numbers, *independently*, according to the same distribution function $F(t)$. Sorting these independent arrival times, we obtain the arrival times of peers following

```
Tracker:
01. Generate $n$ from $N_{t_{\text{exp}}} \sim \text{Poisson}(\Lambda(t_{\text{exp}}))$
02. Send $t_{\text{start}}$, $t_{\text{exp}}$, and $\lambda(t)$ to $n$ chosen peers

Peer $i$, upon receiving $t_{\text{start}}$, $t_{\text{exp}}$, and $\lambda(t)$:
03. Draw waiting time $w_i$ according to $F(t) = \frac{\Lambda(t)}{\Lambda(t_{\text{exp}})}$
04. Compute arrival time: $a_{\text{e},i} = t_{\text{start}} + w_i$
```

Figure 5.3: Algorithm incorporating centralized control for each peer i to choose arrival time $a_{\text{e},i}$.

the desired arrival rate function $\lambda(t)$.

Specifically, Figure 5.3 gives one algorithm derived from the Theorem. In this algorithm, the tracker picks n, chooses n specific peers, and then distributes the rate function to these n peers so that each of them can independently generate its arrival time. One requirement of this algorithm is synchronized clocks across peers for accuracy. We assume that peers either use NTP or a central server (*e.g.*, tracker) as a reference point, and that small inaccuracies on the order a couple seconds are acceptable. The effects of desynchronization are studied in our evaluations.

A problem of selecting specific n peers is that it may induce large overhead on the experiment control system: the tracker becomes hard state and needs to keep track of the specific n peers.

To reduce tracker overhead, Distributed Scenario Control approximates the total number of peers to join the experimental subsystem. This mode trades slight variation in arrival rate for higher scalability.

Specifically, in this mode, Distributed Scenario Control draw the total number of peers from a distribution $\hat{N}_{t_{\text{exp}}}$ with the same mean ($\text{E}[\hat{N}_{t_{\text{exp}}}] = \text{E}[N_{t_{\text{exp}}}]$) but without increasing the variance ($\text{Var}[\hat{N}_{t_{\text{exp}}}] \leq \text{Var}[N_{t_{\text{exp}}}]$). Distributed Scenario Control permits the variance on number of total peers to be reduced in the interest of tighter

```
Tracker:
01. Let $M$ be the total number of available peers
02. Let $p = \frac{\Lambda(t_{\text{exp}})}{M}$
03. Send $t_{\text{start}}$, $t_{\text{exp}}$, $\lambda(t)$, and $p$ to each peer

Peer $i$, upon receiving $t_{\text{start}}$, $t_{\text{exp}}$, $\lambda(t)$, and $p$:
04. if $random() > p$ then return
05. Draw waiting time $w_i$ according to $F(t) = \frac{\Lambda(t)}{\Lambda(t_{\text{exp}})}$
06. Compute arrival time: $a_{\text{e},i} = t_{\text{start}} + w_i$
```

Figure 5.4: Algorithm with decentralized control for each peer i to choose arrival time $a_{\text{e},i}$.

control over the number of peers.

Observe that using $\hat{N}_{t_{\text{exp}}} \sim \text{Binomial}(M, \frac{\Lambda(t_{\text{exp}})}{M})$ (where M is the total number of available peers) not only fits the requirements, but enables a simple distributed solution. The modified algorithm is shown in Figure 5.4. In this approximate algorithm, the tracker computes p as a ratio of the expected value of peers in the scenario ($\Lambda(t_{\text{exp}}) = \text{E}[N_{t_{\text{exp}}}]$) to the total number of available peers M. Each peer i independently, with probability p, participates in the scenario and computes an arrival time at which it will become active. The number of peers to participate in the scenario is distributed according to a Binomial distribution with mean $\Lambda(t_{\text{exp}})$.

Note that the accuracy of the distributed solution is also dependent on peer dynamics. If a significant number of peers join or leave after M is computed (Line 1) but before each peer decides whether to enter the scenario (Line 4), the actual number of peers in the scenario may be significantly different than defined by the peer behavior configuration. If such a change is detected and is undesirable, Distributed Scenario Control aborts the process and retries when the number of available peers stabilizes temporarily.

Controlling Departures: After a peer has determined its arrival time for the scenario, it also needs to determine the time at which it should depart (from the experiments but remain in the stable system). A basic technique is to specify a single probability distribution on peer life time, from which peer departure times are drawn. However, since a user's viewing duration may be dependent on its arrival time or time of day [88], PEAC allows the peer lifetime distribution to be dependent on the arrival time. Specifically, for a peer arriving at time t, the peer's lifetime is drawn from distribution L_t. The set of distributions $\{L_t\}_t$ is a configuration parameter. While this is formally defined as an infinite set, it is approximated as a finite set on non-overlapping time intervals in implementation.

Another factor that contributes to user departure is user behaviors. Specifically, a user may leave a channel sooner if the displayed video has poor quality [64, 65, 88]. PEAC allows the developer to define conditions (*e.g.*, playback freezes 3 times) under which the peer should depart before its lifetime expires.

Extensions with Peer Classes: The preceding controls on arrivals and departures can be customized for each specific *class* of peers, in a setting with heterogeneous peers. For example, the peer behavior configuration may indicate that supernodes should become active earlier than normal peers. In particular, classes are non-overlapping subsets of peers. A class is specified by a set of properties on peers. To determine the class of a peer, our system gathers the properties of the peer when it initially registers with the tracker (*e.g.*, IP address or peer role), from external systems (*e.g.*, P4P [99] or ALTO [5]), or while the peer is running in the stable subsystem (*e.g.*, upload capacity estimate). Denote for peer class j the desired arrival behavior as $\lambda_j(t)$ and the peer lifetime distributions as $\{L_{j,t}\}_t$.

5.4.2 Triggering Conditions

The preceding controlled arrivals and departures start *when* an experiment can be triggered; that is, when the system transits from the *staging* phase to the *testing* phase. A remaining question is at what time the transition can happen.

Distributed Scenario Control triggers an experiment when certain conditions are met. Note that since actual user behavior is not known *a priori*, satisfying the triggering conditions does not guarantee that a sufficient number of peers are available for the duration. The conditions presented here are extended to include a safety margin $\alpha > 1$ to account for prediction variations.

Denote the number of available peers in class j at time t as $\text{avail}_j(t)$. For times in the future, this function may be a conservative estimate and/or based on historical behavior.

Arrived Peers Condition: At time t_{start}, there must be a sufficient number of candidate peers to participate in an experimental scenario. Thus, we have:

$$\Lambda_j(t_{\text{exp}}) \leq \text{avail}_j(t_{\text{start}}). \tag{5.1}$$

Peer Lifetime Condition: A sufficient number of peers only at the beginning is not enough. We also need a sufficient number of peers at any moment during the test.

The expected number of active peers in class j in the scenario at time t can be written as:

$$\text{active}_j(t) \leq \Lambda_j(t) - \int_0^t \lambda_j(x) \Pr\{L_{j,x} < t\}\, dx,$$

where the second term is the cumulative number of departures up until time t. Note that peer departures due to insufficient playback quality are not explicitly included.

Such departures only decrease the number of active peers in the scenario, making the right-hand expression an upper bound.

Thus, we must also have:

$$\text{active}_j(t) \leq \text{avail}_j(t) \, \forall \, t \in [t_{\text{start}}, t_{\text{start}} + t_{\text{exp}}], \tag{5.2}$$

to ensure that there are enough peers in class j throughout the duration of the scenario.

It is important to note that condition (5.2) does not require that any particular peer is active for its entire lifetime. Distributed Scenario Control implements *peer substitution* to handle peers that leave before either the desired departure time is reached or the departure conditions were met. Peer substitution is explained in the next section.

5.4.3 Handling Uncontrolled Early Departures

The preceding algorithms assume that users do not leave while the experiment scenario is in progress. We now handle user-initiated departures, which may occur when users switch channels or terminate the client software while actively participating in a scenario. End-user network and system failures are treated as user-initiated departures. We refer to such peers as *early-departed peers.*

Distributed Scenario Control uses a novel scheme to *substitute* early-departed peers with an available replacement peer in the same channel. When the replacement peer takes over, it “reconstructs” the state of the replaced peer.

Distributed Scenario Control relies on the tracker to choose the replacement peer; the algorithm proceeds in three phases. First, the tracker keeps a summary of the current state for each peer in the scenario. Second, the tracker detects early-departed peers. Finally, the tracker chooses a replacement peer.

Observe that detecting early-departed peers cannot be done within the experimental subsystem since the peer may depart before its scheduled arrival, meaning the experimental subsystem has not yet communicated with the tracker or other peers. Thus, Distributed Scenario Control monitors peer state and detects departed peers within the rescue subsystem.

Capturing Peer State: Each time a peer executes a keep-alive to the tracker, it piggybacks on the message a summary of its current state for the scenario including its scheduled arrival and departure times. If the peer is currently active in the scenario, it also sends its current list of peers and buffer map (list of downloaded pieces). When the peer exits, it also piggybacks its state in the disconnection message to the tracker. To reduce bandwidth, only differences are sent if the tracker acknowledged the previous state.

Detection: Upon receiving a disconnection message from a peer which is active in the scenario, the tracker immediately detects it as an early-departed peer. To handle cases where a disconnection message is not received, Distributed Scenario Control reuses the existing mechanism in the tracker to detect such cases, typically by declaring a peer as departed after failing to receive keep-alive message for a certain time.

The tracker maintains a FIFO queue D_j of last-known states for early-departed peers in class j. Upon detecting an early-departed peer in class j, the tracker appends the peer's last-known state to D_j.

Substitution: Peers that are not participating (either inactive or active) in the scenario are *candidates* to serve as replacement peers. Upon receiving a keep-alive message from a candidate in class j, the tracker checks if D_j is non-empty. If so, it dequeues the first peer state from the queue and includes it in the keep-alive reply.

If the received state indicates that the replaced peer was active in the scenario, the replacement peer immediately becomes active as well, and downloads the required pieces from a CDN or supernode (to avoid causing excess resource contention). It also initiates connections to the indicated peers. If the received state indicates that the replaced peer was not active in the scenario, then the replacement peer waits until the scheduled arrival time to become active.

5.4.4 Extensions

Re-use Experiment Peers: By default, Distributed Scenario Control allows a peer to choose a single arrival time in the scenario. This makes it easier to satisfy Theorem 1, which requires that each arrival time be chosen independently, whereas selecting a second arrival time for a peer depends on the first arrival and departure times that were selected. To allow such flexibility, Distributed Scenario Control can be extended by dividing the arrival and departure behaviors into independent sub-intervals.

P2P-enabled Dead Peer Detection: Relying on the tracker to declare the peer as departed may cause the delay for substituting the peer to be long if a disconnection message is not received. If the rescue system uses P2P exchanges, it is possible to detect such cases much faster since message exchanges between peers is much more frequent.

5.5 Adaptive Task Reassignment

Now that we have an environment in which we can create experiment scenarios at a large scale with real users, we present our resource/task scheduler. The scheduler controls the resources and tasks allocated to the stable, rescue, and experimental subsystems, and it is crucial towards satisfying requirements for both protection of user experience and experimental accuracy (Section 5.3.5).

During the staging phase, the scheduler treats the stable system as if it were a rescue subsystem running without any corresponding experimental subsystem. Thus, in this section, we focus on the testing phase when there are both a rescue subsystem and experimental subsystem running concurrently.

5.5.1 Overview of Adaptive Allocation

Adaptive Task Reassignment is based on simple insights: it introduces time shifting and conducts adaptive task and resource allocation across time.

To present Adaptive Task Reassignment, first consider the case that the experimental subsystem is running alone at its typical lag behind the source, with all of the download tasks and resources. This is illustrated in Figure 5.5(a). In this figure, the rightmost piece is the most recent piece produced by the source at time t, denoted by $p_{src}(t)$; $e_{play}(t)$ is the next piece to be played. For simplicity, below we assume that the playpoints of clients are synchronized; small synchronization errors may occur in practice and are acceptable.

The basic idea of Adaptive Task Reassignment is to first run the experimental subsystem as if it were running alone (*i.e.*, with all of the resources and tasks) This improves experimental accuracy. However, in case a piece is not received by its playback deadline in the experimental subsystem, responsibility for downloading the piece is shifted to the rescue subsystem. The rescue subsystem may either be the existing stable system, or it may download from a CDN or supernode. To make rescuing possible, Adaptive Task Reassignment allocates a recovery time $T_{recover}$ to the rescue subsystem to download the missed pieces before the true playpoint $r_{play}(t)$, which is the next piece to be displayed to the user. This mode is illustrated in Figure 5.5(b).

The Adaptive Task Reassignment idea is motivated by existing hybrid CDN+P2P systems (*e.g.*, [42]). In a general sense, one may consider the CDN component in a hybrid CDN+P2P system as the rescue to the P2P component. If a piece cannot be fetched by the P2P component T seconds before its time to be displayed to users, the piece will be assigned to the CDN component. In this sense, the existing hybrid CDN+P2P systems already implement adaptive time shifting of the assignments of tasks.

However, Adaptive Task Reassignment generalizes the hybrid CDN+P2P scheme to the setting of performance experimentation to address key architectural and accuracy issues. The first issue of the hybrid systems is that since they do not consider experimental accuracy, they do not consider the data flow constraints (Section 5.5.3). When the data fetched by CDN are injected to the P2P system and redistributed, we can no longer observe the performance of the P2P system alone (*i.e.*, not satisfying the (R2) requirement). Second, in a general setting, it may be essential that a rescue subsystem utilizes P2P. For example, when there is a shared network bottleneck, a P2P based rescue subsystem may retrieve all missing pieces, while a rescue subsystem using only a CDN may not. Thus, Adaptive Task Reassignment is designed for general, modular rescue and experimental subsystems.

To design an extensible framework for the task/resource scheduler of a system composed of a general, modular rescue subsystem and a general, modular experimental system, Adaptive Task Reassignment introduces a novel scheme called Player Buffer Window as a Control API.

Requirements: To illustrate the need for this API, we first present requirements that must be satisfied by the tasks allocated to the rescue and experimental subsystems when they are both playing.

Bound recovery time: There must be at least time $T_{recover}$ between the playpoints $r_{play}(t)$ and $e_{play}(t)$:

$$e_{play}(t) \geq T_{recover}\mu + r_{play}(t), \tag{recovery}$$

where μ is a the number of pieces per second and is a configuration parameter defined by the channel.

Bound experimental subsystem lag: The experimental subsystem's lag from the source, $p_{src}(t) - e_{play}(t)$, can influence its performance (*e.g.*, if too little time is given to distribute pieces amongst clients). Thus, the experimental subsystem is asked to provide a lower bound e_{lag}^{min} on its lag from the source:

$$p_{src}(t) - e_{lag}^{min} \geq e_{play}(t). \tag{exp-lag}$$

Fixed user-visible lag: The user-visible playpoint is used by both the stable system (during the staging phase) and the rescue subsystem (during the testing phase). To avoid a sudden jump to a different point in the video stream, the user-visible playpoint $r_{play}(t)$ must have fixed lag from the source even as the experimental subsystem is started or stopped. The lag should be sufficiently large to accommodate different experimental subsystems. We assume that minimum user-visible lag r_{lag}^{min} is given as a parameter of the channel:

$$p_{src}(t) - r_{lag}^{min} \geq r_{play}(t). \tag{rescue-lag}$$

Assigning Playpoints is Not Enough: One may suggest that a simple mechanism through which the scheduler controls the subsystems is by assigning their playpoints. However, this is not sufficient. First, the playpoint chosen by either the rescue or experimental subsystems may be a result of their particular algorithms, download rate received from its download source (*e.g.*, peers or a CDN), and desired startup delay. Second, after a particular subsystem begins to play, it may be necessary to pause and rebuffer, possibly shifting the playpoint forward or backward in time. For these reasons, the scheduler should neither assign a particular playpoint nor assume that the playpoint advances at a constant rate.

5.5.2 Player Buffer Window as a Control API

To provide each subsystem flexibility in choosing playpoints, the scheduler instead uses the *buffer window* as a control mechanism instead of the playpoint itself. In this scheme, at any time t, each subsystem is assigned a sliding window indicating the range of pieces that the subsystem can download. We call this window the player buffer window of the subsystem. The scheduler controls a subsystem by providing it an initial window and sliding it at rate μ as time progresses. The subsystem may set then its own playpoint. Given this general API, Adaptive Task Reassignment leaves as much flexibility to individual subsystems as possible.

We denote the window position (the leftmost piece) for the rescue and experimental subsystems at time t as $r_{left}(t)$ and $e_{left}(t)$, respectively.

Given this design, a remaining question is *how* the scheduler assigns the initial windows for each subsystem. In particular, the scheduler provides to the rescue subsystem the value $r_{left}(a_{\mathrm{r},i})$ when the rescue subsystem starts at time $a_{\mathrm{r},i}$. Likewise, the scheduler provides the experimental subsystem the value $e_{left}(a_{\mathrm{e},i})$ when it starts at time $a_{\mathrm{r},i}$.

API Definition: The Player Buffer Window API is implemented by two callbacks provided by each subsystem.

setBufferWindowPosition(pos): The scheduler provides as input to the subsystem

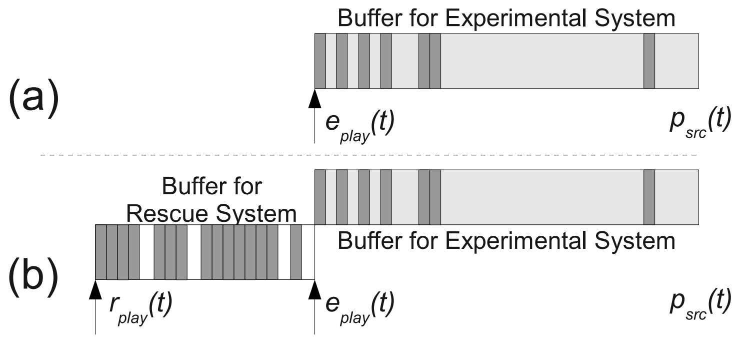

Figure 5.5: Overview of Adaptive Task Reassignment; (a) Experimental subsystem running alone; (b) Adaptive Task Reassignment mode.

the leftmost piece in its buffer window. Specifically, the scheduler provides $e_{left}(a_{\mathrm{e}})$ to the experimental subsystem and $r_{left}(a_{\mathrm{r}})$ to the rescue subsystem.

getPlaypointRange(): To provide flexibility in the actual playpoint position, the scheduler asks a subsystem to provide lower and upper bounds on the offset of the playpoint within the buffer window. Let e_{off}^{min}, e_{off}^{max} denote the bounds provided by the experimental subsystem. Thus, $e_{left}(t) + e_{off}^{max} \geq e_{play}(t) \geq e_{left}(t) + e_{off}^{min}$. Corresponding notations r_{off}^{min} and r_{off}^{max} are defined for the rescue subsystem.

API Usage: The API is used in the following way. First, the scheduler queries a subsystem to determine its playpoint range. Next, the scheduler computes the buffer window position and supplies it to the subsystem. Specifically, when the rescue subsystem starts at time a_{r}, the scheduler computes $r_{left}(a_{\mathrm{r}})$ to instantiate the rescue subsystem. When the experimental subsystem starts at time a_{e}, the scheduler computes $e_{left}(a_{\mathrm{e}})$ to instantiate the experimental subsystem.

After instantiation, the player window of the rescue system is from $[r_{left}(t), e_{left}(t))$, and the experimental subsystem is from $[e_{left}(t), p_{src}(t))$. We next detail the algorithm for computing buffer window positions for both a rescue subsystem and experimental subsystem.

Rescue Player Buffer Window: The rescue player buffer window is computed when the client first joins the channel at time a_{r}.

The scheduler must only consider requirement (rescue-lag) to initialize the rescue subsystem. Thus, it enforces: $p_{src}(t) - r_{lag}^{min} \geq r_{left}(t) + r_{off}^{max}$. Since p_{src} and r_{left} increase at constant rate μ, it suffices to enforce: $p_{src}(a_{\mathrm{r}}) - r_{lag}^{min} \geq r_{left}(a_{\mathrm{r}}) + r_{off}^{max}$.

The scheduler then directly computes $r_{left}(a_{\mathrm{r}})$ as:

$$r_{left}(a_{\mathrm{r}}) = p_{src}(a_{\mathrm{r}}) - r_{off}^{max} - r_{lag}^{min}, \tag{5.3}$$

to minimize the user-visible lag within the requirements.

Experimental Player Buffer Window: The scheduler computes $e_{left}(a_{\mathrm{e}})$ when the experimental subsystem is started. This occurs at the time a_{e} provided by Distributed Scenario Control.

The scheduler must consider requirements (recovery) and (exp-lag) to initialize the experimental subsystem. Combining these requirements, we have $p_{src}(t) - e_{lag}^{min} \geq e_{play}(t) \geq T_{recover}\mu + r_{play}(t)$. To satisfy the combined requirement, the scheduler enforces:

$$\begin{aligned}
&p_{src}(t) - e_{lag}^{min} \\
&\geq e_{left}(t) + e_{off}^{max} \geq e_{left}(t) + e_{off}^{min} \\
&\geq T_{recover}\mu + r_{left}(t) + r_{off}^{max}.
\end{aligned}$$

Since p_{src}, e_{left}, and r_{left} all increase at constant rate μ, it suffices to enforce:

$$\begin{aligned}
&p_{src}(a_{\mathrm{e}}) - e_{lag}^{min} && \text{(exp-max-pp)} \\
&\geq e_{left}(a_{\mathrm{e}}) + e_{off}^{max} \geq e_{left}(a_{\mathrm{e}}) + e_{off}^{min} \\
&\geq T_{recover}\mu + r_{left}(a_{\mathrm{e}}) + r_{off}^{max}. && \text{(exp-min-pp)}
\end{aligned}$$

From these, the scheduler must determine a value for $e_{left}(a_\mathrm{e})$.

However, it is not guaranteed that a feasible assignment exists (*e.g.*, if the experimental subsystem requires too large of a lag). Observe that the quantity (exp-max-pp) is bounded away from (exp-min-pp) by $e_{off}^{max} - e_{off}^{min}$. Thus, a feasible assignment exists iff the condition

$$\begin{aligned} & p_{src}(a_\mathrm{e}) - e_{lag}^{min} - T_{recover}\mu - r_{left}(a_\mathrm{e}) - r_{off}^{max} \\ & \geq e_{off}^{max} - e_{off}^{min}, \end{aligned} \tag{5.4}$$

is satisfied. Note that (5.3) may be shifted to time a_e since r_{left} and p_{src} increase at constant rate μ. After translation and combining with (5.4), we have:

$$r_{lag}^{min} - e_{lag}^{min} - T_{recover}\mu \geq e_{off}^{max} - e_{off}^{min}. \tag{5.5}$$

If no feasible assignment exists to (5.5), the scheduler returns an error indicating that the experiment cannot be executed in the channel. It is important to observe that this condition may be verified when an experiment scenario is defined instead of waiting until subsystem instantiation at each peer.

If the assignment is feasible, the scheduler directly computes $e_{left}(a_\mathrm{e})$ as:

$$e_{left}(a_\mathrm{e}) = p_{src}(a_\mathrm{e}) - e_{off}^{max} - e_{lag}^{min}.$$

5.5.3 Data Flow Constraint

There are additional considerations to account for when implementing Adaptive Task Reassignment, in particular on data flow. The rescue subsystem is restricted to only download pieces in the range $[r_{left}(t), e_{left}(t))]$ to prevent it from downloading duplicate pieces.

It is important that pieces downloaded by the rescue subsystem are *not* made available to the experimental subsystem. Injecting additional pieces into the experimental subsystem can affect experimental accuracy. However, note that it is safe to share pieces received by the experimental subsystem with the rescue subsystem. This local copy does not affect $Perf(A^E)$ and makes it easier to satisfy $Perf(A) \geq Perf(A^S)$.

5.5.4 Rescue Subsystem Window Adjustment

After the experimental subsystem assigns a playpoint, the rescue subsystem's player buffer window may be expanded to be $[r_{left}(t), e_{play}(t))]$ at each time t if the experimental subsystem does not download pieces after the playpoint. This provides additional time for the rescue subsystem to recover missed pieces.

5.5.5 Load on Rescue Subsystem

In order to achieve (R2), it is important to control load on the rescue subsystem when peer resources are used. When a peer initially starts the experimental subsystem under Distributed Scenario Control, there may be missing pieces in the rescue subsystem's window. We trigger CDN protection for these pieces.

In order to achieve (R1), when there is a persistent high load on the rescue subsystem, clients depart the experimental subsystem and return to using the stable system alone.

5.5.6 Accuracy

Adaptive Task Reassignment achieves (R2) or provides a lower bound. In particular, consider the case when the rescue subsystem downloads missing pieces from other peers also running the experimental subsystem. When the experimental subsystem performs well enough that the triggered rescue does not cause substantial interference, Adaptive Task Reassignment provides accurate experimental results. On the other hand, when there are high demands on the rescue subsystem, there can be contention with the experimental subsystem, causing the experimental subsystem to under-perform.

5.6 Compositional Runtime

We implement PEAC using a new architecture called Compositional Runtime. This block-based architecture not only supports experimentation with dual systems and easier distribution of code for an experiment scenario, but also matches well with the setting that large, distributed live streaming systems supporting peer-to-peer mode typically consist of a set of key algorithmic components such as connection management, upload scheduling, admission control, and enterprise coordination.

5.6.1 Overview

The key objective of the Compositional Runtime is that the software structure should allow modular design of algorithmic modules as well as easy composition of a system consisting of both rescue and experimental algorithmic modules for an experimental trial. On the other hand, if we run the multiple subsystems in separate programs, experimentation control (*e.g.*, data flow and buffer coordination in Adaptive Task Reassignment) will be challenging to implement. Specifically, we identify the following specific requirements for the framework:

- *Modularity*: Allow the developer to isolate algorithmic functions into self-contained blocks; blocks in one subsystem should be oblivious to other subsystems.
- *Composition*: Allow blocks to be downloaded and composed at runtime. This enables evaluations on-the-fly without waiting for (or forcing) users to stop and re-start their clients.
- *Isolation*: Protect against poorly-behaving blocks and algorithms. Protection should be provided against crash failures and scheduling should account for blocks consuming too much CPU or memory.
- *Data sharing*: Allow blocks to access shared data structures (*e.g.*, the list of connected peers and the buffer maintaining downloaded pieces). This can provide a more natural interface to developers than an alternative such as an event notification system.

Our software architecture is inspired by prior architectures such as Click [55], GNU Radio [38] and SEDA [95], but is tailored to meet the aforementioned requirements. Algorithmic components are implemented as independent blocks that define a set of input ports and output ports over which messages (called *packets* in our system) are received and emitted. A runtime scheduler is responsible for delivering packets between blocks.

5.6.2 Modularity and Composibility

Block composition for live streaming: One key aspect of our framework is easy composition of algorithmic functions. We use a simple example to illustrate it.

Figure 5.6 illustrates a portion of a live streaming client that is responsible for managing peer connections. A block is responsible for demultiplexing received pack-

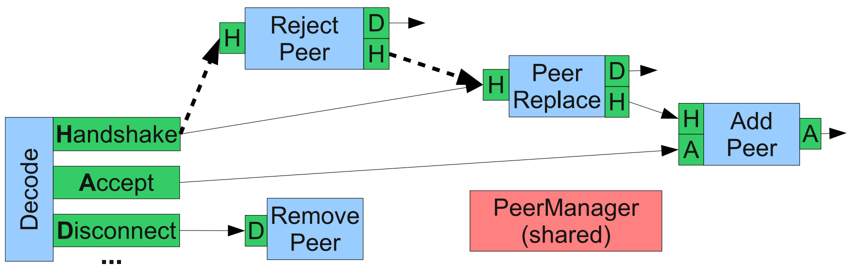

Figure 5.6: Adding an admission control component.

ets and emitting them to connected blocks responsible for handling each type of message.

Now, assume the designer wishes to add an admission control algorithm to avoid reduced performance for existing peers during flash crowds. Admission control may be implemented as an independent block which reads handshake messages for newly-connected peers. The block emits either the handshake message if the new connection should be accepted or a disconnect message to the peer if the connection should be rejected. The designer then composes the block as shown by the dotted line.

This example illustrates how the framework facilitates performing experiments in a variety of scenarios by composing different combinations of algorithms without developing multiple versions of the client software.

Framework Basics: Specifically, PEAC software architecture allows a software designer to write a new block by extending a simple C++ interface. Copy-on-write is used to avoid unnecessary memory copying when delivering packets. Compositions of blocks are defined by a simple JSON configuration file.

To simplify development of new blocks, the runtime also allows blocks to schedule timers for themselves. The designer may also instruct the runtime to execute particular blocks in separate threads, thus enabling the use of blocking operations such `select`.

Blocks are distributed as shared libraries and may be dynamically loaded and composed at runtime. Similarly, blocks may be disconnected, removed, and unloaded at runtime. This feature enables the client to be reconfigured at runtime to transition between the staging and testing phases (*e.g.*, to install the rescue and experimental subsystems).

Callbacks at extension points: Some functionality may not be implementable only by composing blocks. For example, a download scheduler algorithm has two responsibilities: to decide which data to download and from which peer. Formulating this as a matching problem, the result may be suboptimal if both peers and available data from each peer are not considered jointly. To handle cases where algorithms are difficult to decouple, our framework allows blocks to also define callbacks at predefined extension points.

5.6.3 Coexistence of Dual Systems

Our software architecture facilitates designing and running both rescue and experimental systems. In particular, since blocks are independent, the experimental system can be composed from existing algorithms in the stable system by adding, removing, or replacing functionality.

Scheduling: To achieve the requirement of $Perf(A) \geq Perf(A_S)$, the software framework monitors and controls resources used by the experimental system to avoid reduced performance to the overall system. OS-level mechanisms (*e.g.*, similar to [50]) and a "split-process" model [39, 94] can be used to control CPU and protect against crashes.

Data flow and scope: Another aspect of our software system supporting dual systems is to ensure that the rescue and experimental systems composed in the same client share data only according to certain policies. Such policies are attached as a *data scope* to each block.

Data scopes are hierarchical to allow both data sharing and isolation. In our Adaptive Task Reassignment implementation, a *root* scope contains a block that copies pieces from the *experimental* child scope to the *rescue* child scope (see Section 5.5.3). The rescue and experimental systems run in different data scopes so that data structures are isolated by default.

5.7 Evaluations

In this section, we evaluate multiple aspects of PEAC. First, we evaluate the software architecture in terms of code size and implementation experience. Then, we evaluate Distributed Scenario Control with focus on how its scalability and accuracy with a large number of clients. Finally, we evaluate Adaptive Task Reassignment.

5.7.1 Methodology

We have implemented a complete live streaming system with clients, sources, trackers, and an experiment manager. Clients and sources are implemented using the software framework presented in Section 5.6. Sources contain a custom block that downloads pieces from a CDN.

To understand behavior for large-scale channels for Distributed Scenario Control, we use emulated clients using the same client-side implementation of Distributed Scenario Control used by our real clients. We use logs from PPLive channels and the data presented in [2], a measurement of a live baseball game broadcast with nearly 60,000 concurrent users at its peak, to drive our emulations.

To evaluate Adaptive Task Reassignment, we deploy our tracker and clients on

Emulab [97] using Modelnet [90] to control delays and upload capacities. The RTT between clients is 100 ms. A standard HTTP server serves as the CDN.

We focus on two metrics to quantify viewing quality for clients: (1) piece miss rate, defined as fraction of pieces that are not downloaded before their playback deadline, and (2) average buffer ratio, defined as the percentage of pieces in clients' buffers, averaged across all clients. Note that buffer ratio is one of the primary metrics that PPLive uses to determine whether the test of a new client in a testing channel is successful or not.

5.7.2 Software Framework

We first evaluate the software framework. Since we have implemented the full system, we present statistics on the size of the code to illustrate that the framework is simple yet powerful.

The framework and live streaming client are implemented in C++. The full system is divided into multiple components:

- Compositional Runtime (including scheduler, dynamic loading of blocks, etc): 3400 lines of code;
- Pre-packaged blocks (HTTP integration, UDP sockets and debugging): 500 lines of code;
- Live streaming client: 4200 lines of code.

In our implementation experience, using this software architecture has enabled us to incrementally develop and test individual components of the live streaming client. With the Compositional Runtime, members of our research group have implemented application-layer rate limiting, modified download schedulers, and even push-based live streaming by simply writing new blocks and updating a configuration file.

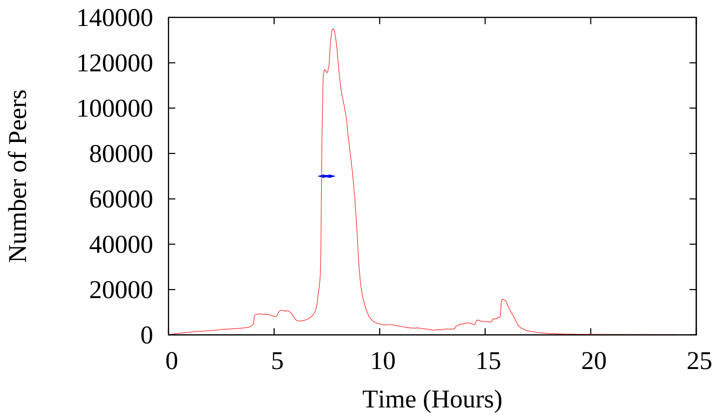

Figure 5.7: Opportunities to trigger 70,000-client 1-hour experiment in SH Sports channel.

5.7.3 Distributed Scenario Control

We now evaluate Distributed Scenario Control by considering the full experiment lifecycle.

Triggering Opportunities: First, we evaluate opportunities to trigger experiments in real P2P live streaming channels. Using logs from popular PPLive channels, we determine the size and duration of experiments that could be triggered by Distributed Scenario Control.

Figure 5.7 shows the results for a 1-hour experiment with 70,000 peers within the SH Sports channel on September 8, 2010. For simplicity, we assume that all peers participate for the full duration of the experiment. The curve indicates the total number of peers in the channel. The horizontal line indicates the times, a 30-minute window in this case, at which the experiment could be triggered.

Figure 5.8 shows the results for a 4-hour experiment with 20,000 peers the HN Satellite channel on September 10, 2010. There is a 45-minute window in which the experiment could be triggered. We observe that this channel permits longer experiments (still with a large number of peers), in contrast to the SH Sports channel

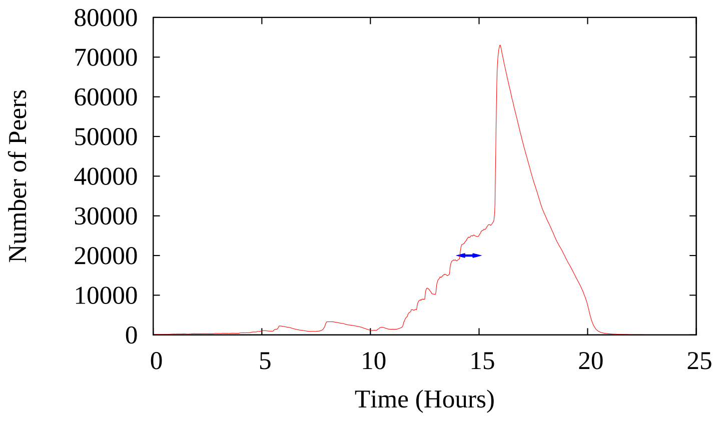

Figure 5.8: Opportunities to trigger 20,000-client 4-hour experiment in HN Satellite channel.

which supports even large experiments but for a shorter duration.

Scenario Parameter Distribution Overhead: Next, we show that Distributed Scenario Control is scalable with respect to control overhead incurred by the tracker to distribute the peer behavior configuration to each client. In our implementation, the tracker includes these parameters in the keep-alive reply. To save bandwidth, the tracker first determines the client's class with a local lookup, then sends only the peer behavior configuration for the client's class. The size of the peer behavior configuration depends on how the arrival rate parameter $\lambda(t)$ is encoded. For flexibility, we use a piecewise-linear definition, causing the overhead to scale with the number of defined sections. Other families of functions could be encoded in a fixed number of bytes.

We compute the control overhead as it scales with the number of peers in the channel for a 1-hour experiment. Figure 5.9 shows the results. For comparison, we also compute the total volume of traffic used by keep-alive messages sent by each peer to the tracker every 30 seconds during the experiment. We observe that the overhead is extremely small, less than 70MB, for a simple arrival behavior definition with 1 million peers. For more complicated arrival behavior definitions with 50 sections in

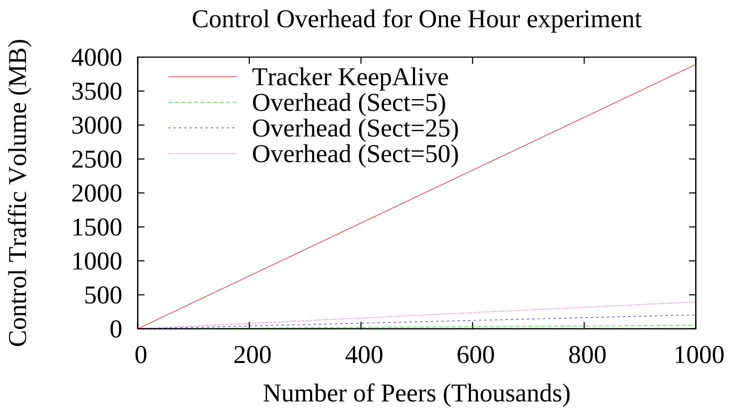

Figure 5.9: Distributed Scenario Control requires little additional traffic.

the piecewise linear function, the overhead remains under 400MB.

Clients must also download the code for an experiment scenario, which includes the algorithmic blocks used. We do not include these overheads in the preceding overhead evaluation since they are inherent to software distribution and existing upgrade mechanisms as well.

Achieved Arrival Behavior: Next, we evaluate the achieved arrival behavior. Though Theorem 1 shows that the resulting arrival behavior should be a Poisson process, it is possible for the achieved behavior to be impacted by (1) the use of a Binomial for distributed control of number of clients, and (2) inaccurate time synchronization between clients.

We evaluate the arrival behavior drawing from real arrival behaviors. In particular, we use the join rate captured from Figure 1 of [2] to instantiate the rate parameter $\lambda(t)$.

To test the achieved arrival behavior with a large number of peers, we emulate Distributed Scenario Control with 100,000 peers. Each client's arrival time is perturbed by a uniform random value selected from the interval $[-\frac{s}{2}, \frac{s}{2}]$ to test the impact of time desynchronization.

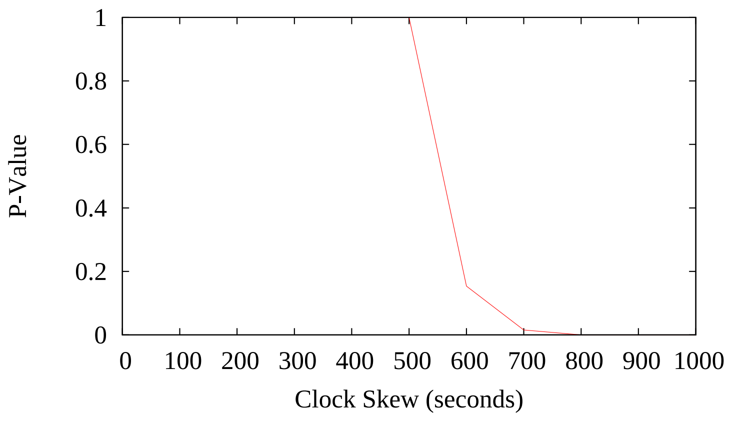

Figure 5.10: Distributed Scenario Control generates accurate arrival behaviors.

We test the hypothesis that the achieved arrival behavior could have been generated by rate parameter $\lambda(t)$ using a Chi-Square test with the null hypothesis that the arrivals in interval $[t_1, t_2)$ are distributed according to $\Lambda(t_2) - \Lambda(t_1)$. We place arrival times in intervals of width 3 seconds, and then merge adjacent intervals until both the expected number of arrivals and the actual number of arrivals in each is at least 5. We execute the Distributed Scenario Control algorithm 30 times for each value of s, compute the p-value for each, then combine the p-values using Fisher's Method and report an overall p-value.

Figure 5.10 shows the results. We observe that there is insufficient evidence to reject the null hypothesis until after $s > 600$ seconds, where the p-value falls below 0.1. This appears to be a surprisingly-high tolerance, but closer inspection of the join rate reveals that the rate changes only gradually. In particular, the slope of the join rate curve at the start of the event ($t = 4$ hours) is 0.93 peers/minute. Next, we consider Distributed Scenario Control with join rates that change more sharply.

To evaluate the arrival behavior in a more severe flash crowd, we perform the same experiment with arrival rate function $\lambda(t) = 2t$ for $t \in [0, 160]$ seconds and a channel size of 3000 peers. The p-value was greater than 0.9 for clock skew $s \leq 3$

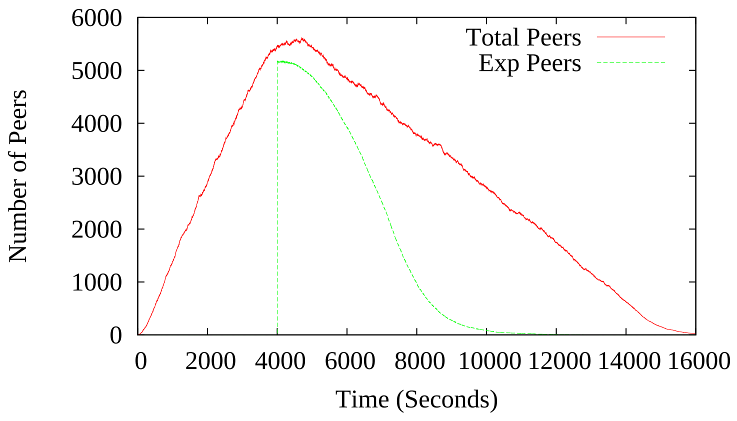

Figure 5.11: Total peers and experimental peers in channel used to evaluate peer dynamics.

seconds and dropped below 0.1 for $s \geq 5$ seconds. Thus, after $s = 5$, we can reject the null hypothesis.

Peer Dynamics: Finally, we evaluate the peer substitution. In particular, we evaluate the delay required to substitute peers, with the intuition that departed peers should be substituted within a short amount of time in order to avoid disrupting experimental accuracy.

To evaluate the peer substitution delay with a substantial channel size, we emulate for the peer substitution algorithm in Section 5.4.3. The emulator captures peer arrival and departure behaviors, keep-alive messages, and re-uses the same experimental control implementation in our real clients.

Figure 5.11 shows the total number of peers in the channel and the number of peers included in the configured scenario. The sharp increase at $t = 4000$ seconds is the time the experiment is triggered, when 5187 peers begin to participate in the scenario. Note that these peers do not become active in the scenario until their chosen arrival time, but are substituted even if they depart before becoming active. The peer substitution algorithm is utilized as the total number of peers declines while there are still peers in the scenario.

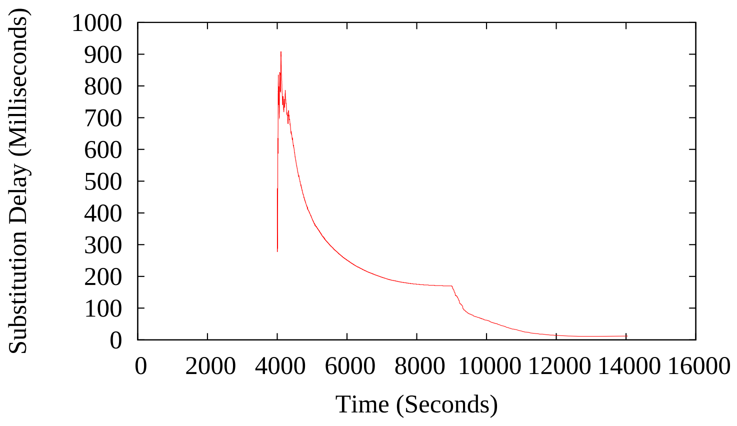

Figure 5.12: Peer substitution delay with peer dynamics.

Figure 5.12 shows the peer substitution delay averaged over 5-second windows. We observe that the substitution delay is larger (nearly 1 second on average) at the start of the experiment. This is because most peers in the channel are selected to join the experiment, leaving only 276 replacement peers. As more replacement peers become available, the substitution delay decreases. For this setting, the maximum peer substitution delay is 2.8 seconds while the average is much lower at 0.168 seconds.

5.7.4 Adaptive Task Reassignment

We now evaluate Adaptive Task Reassignment. We use an experimental algorithm in which a bug is introduced into the peer management component; the timeout for disconnecting unresponsive peers is set to 1 second (the default value is 5 seconds).

We also consider multiple rescue subsystems:

- *P2P Rescue*: The rescue subsystem is our stable system, and
- *CDN Rescue*: The rescue subsystem downloads missing pieces from a CDN.

The following settings are used for our experiments. 120 clients join the channel with an inter-arrival time of 15 seconds. After all clients have joined, they play for 5 minutes. The channel rate is 40 KBps. $T_{recover}$ is 120 seconds.

Experiment accuracy: We first consider experimental accuracy. Table 5.1 shows the results. The buggy algorithm has introduced poor performance, confirmed by the increased piece miss rate of 4.37% (our default P2P algorithms have a 0.46% piece miss rate in this setting). When either the CDN or P2P rescue subsystems are enabled, the measured piece miss rate and buffer ratio is accurate, with a 2.7% error at most. The piece miss ratio when using P2P Rescue is slightly higher since it shares bandwidth resources with the experimental subsystem.

	Buggy	CDN Rescue (Err)	P2P Rescue (Err)
Pieces Missed	4.37%	4.37% (0%)	4.48% (2.5%)
Buffer Ratio	72.71%	70.72% (2.7%)	72.98% (0.4%)

Table 5.1: Adaptive Task Reassignment can achieve experiment accuracy.

User-observable performance: We next investigate user-observable performance. The results are shown in Table 5.2. As expected, the user observes no missed pieces when using CDN Rescue. Using P2P Rescue also shows very good performance, with only 0.04% of pieces missed. This is fewer than the 0.46% missed pieces when the P2P Rescue runs alone, because fewer pieces need to be downloaded by P2P Rescue when the experimental subsystem has already downloaded most of them.

	CDN Rescue	P2P Rescue	P2P Rescue Alone
Pieces Missed	0.0%	0.04%	0.46%
Buffer Ratio	96.66%	97.47%	84.78%

Table 5.2: Adaptive Task Reassignment can protect user experience.

5.8 Discussions and Future Directions

We have illustrated that how PEAC enables scalable evaluation with real user clients. Distributed Scenario Control allows scalable creation of user behaviors; Adaptive

Task Reassignment achieves scalable protection of user experience and experimental accuracy.

5.8.1 Discussion

Q: [Intended Scope]: Is PEAC intended to create a new testbed for live streaming?

A: PEAC is intended to be complementary to existing testbeds (*e.g.*, Emulab and PlanetLab). PEAC has the ability to create and execute large-scale experiments with real end-users and network environments that would be encountered if deployed to end-users. Testbeds can be useful if more controlled environments or conditions are needed for certain tests or evaluations.

Q: [Human Behavior]: Can PEAC evaluate the effects of human behaviors on performance?

A: Human behavior with respect to video quality is not well understood and may even vary between users, making it difficult to model. Since human behaviors may have effects on the performance (*e.g.*, depart sooner if there is poor video quality), it can be beneficial to evaluate such behaviors within PEAC.

PEAC could be extended to evaluate human behaviors by displaying the experimental subsystem's video stream instead of the rescue subsystem's stream, either by default or after a user "opts-in". Instead of departing if poor video quality is observed, users may have the option to view the rescued video stream.

5.8.2 Future Directions

There are also avenues for future work. In particular, given the capabilities of PEAC to run multiple experiment scenarios, it would be useful to have an automated debugging framework for live streaming to identify performance issues in an experimental system. Another future direction is to extend PEAC to other streaming systems, in particular video-on-demand and CDN-based live streaming.

Chapter 6

Conclusions and Future Directions

In this dissertation, we present a novel Dual-System Architecture for supporting large-scale, realistic evaluations for networked systems. Dual systems provide unique benefits beyond existing techniques for testing and evaluating networked systems. In particular, running the test system on the same infrastructure as the production system, evaluations can be performed at the scale of the production infrastructure. By exploiting application-specific semantics, we have shown that dual systems can simultaneously provide accurate evaluations and avoid disruption to users. We have illustrated how the Dual-System Architecture applies to real-world systems through ShadowNet, a dual system for network configuration management, and PEAC, a dual system for Internet-scale evaluation of a P2P streaming environment.

There are multiple avenues for future work. First, it can be possible to better support incremental deployment. In particular, it may not be possible to simultaneously upgrade all components of a production infrastructure simultaneously to support dual systems. In such a case, the components not supporting dual systems are considered outside of the Dual-System Boundary. Though ShadowNet supports certain cases (*i.e.*, if the Dual-System Boundary is between BGP peers), a future direction is to develop techniques to ease incremental deployment for other settings as well.

Second, dual systems may integrate well with online debugging tools. For example, [35] provides an debugging environment for networked applications. When both the test and production system are running concurrently, it may be possible to insert breakpoints at desired points in the execution to inspect the environment while the other system is able to continue normal operation.

Finally, dual systems may be applicable to other usage scenarios as well. Though network infrastructure and P2P live streaming are important applications, a future direction is to apply dual systems to other types of content distribution such as CDNs and video-on-demand. In particular, it may be possible to apply similar techniques to resolve the conflict between accuracy and protection against disruption. One can view the Packet Cancellation algorithm in ShadowNet as a lossy compression that takes advantage of particular properties of traffic delivered by the test system. Similar structure or properties may exist in other systems as well.

Bibliography

[1] Adobe. Adobe Flash Player 10.1. http://labs.adobe.com/technologies/flashplayer10/.

[2] S. Agarwal, J. P. Singh, A. Mavlankar, P. Baccichet, and B. Girod. Performance and Quality-of-Service Analysis of a Live P2P Video Multicast Session on the Internet. In *16th International Workshop on Quality of Service, IWQoS 2008*. Springer, June 2008.

[3] M. Agrawal, S. R. Bailey, A. Greenberg, J. Pastor, P. Sebos, S. Seshan, K. van der Merwe, and J. Yates. RouterFarm: Towards a Dynamic, Manageable Network Edge. In *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management*. ACM, Sept. 2006.

[4] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009*. ACM, Oct. 2009.

[5] IETF ALTO Working Group. `https://datatracker.ietf.org/wg/alto/charter/`.

[6] P. Anderson and A. Scobie. Large scale Linux configuration with LCFG. In *Proceedings of the Atlanta Linux Showcase*, Berkeley, CA, USA, 2000.

[7] D. F. Bacon and S. C. Goldstein. Hardware-assisted Replay of Multiprocessor Programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM, May 1991.

[8] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Kyoto, Japan, Aug. 2007. ACM.

[9] C. Bastian, T. Klieber, J. Livingood, J. Mills, and R. Woundy. Comcast's Protocol-Agnostic Congestion Management System. `http://tools.ietf.org/html/draft-livingood-woundy-congestion-mgmt-09`, Sept. 2010.

[10] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pisa, Italy, Sept. 2006. ACM.

[11] BitTorrent User Forums. `http://forum.utorrent.com/viewtopic.php?id=82450`.

[12] T. Bonald, L. Massoulie, F. Mathieu, D. Perino, and A. Twigg. Epidemic Live Streaming: Optimal Performance Trade-offs. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Annapolis, MD, USA, June 2008. ACM.

[13] L. Bracciale, F. L. Piccolo, S. Salsano, and D. Luzzi. Simulation of Peer-to-Peer Streaming Over Large-scale Networks using OPSS. In *Proceedings of the*

2nd International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2007. ACM, Oct. 2007.

[14] M. Caesar, L. Subramanian, and R. Katz. A Case for an Internet Health Monitoring System. In *Proceedings of Hot Topics in System Dependability (HotDep)*. USENIX, June 2005.

[15] CAIDA. CAIDA: Cooporative Association for Internet Data Analysis. `http://www.caida.org/`.

[16] B.-Y. Choi1, S. Moon, Z.-L. Zhang, K. Papagiannaki, and C. Diot. Analysis of Point-To-Point Packet Delay In an Operational Network. In *Proceedings IEEE INFOCOM 2004, The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, Mar. 2004.

[17] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[18] Cisco Systems. Common Routing Problem with OSPF Forwarding Address. `http://www.cisco.com/warp/public/104/10.pdf`, Dec. 2005.

[19] Cisco Systems. Network Solutions Integrated Test Environment: Delivering on the Promise of Innovation. `http://www.cisco.com/en/US/solutions/ns341/ns522/networking_solutions_products_genericcontent0900aecd80458f98.pdf`, 2006.

[20] Comcast. Comcast PowerBoost. `http://customer.comcast.com/Pages/FAQListViewer.aspx?topic=Internet&folder=8b2fc392-4cde-4750-ba34-051cd5feacf0`.

[21] D. R. Cox and P. A. W. Lewis. *The Statistical Analysis of Series of Events*. Methuen, London, England, 1966.

[22] M. Crovella, C. Lindemann, and M. Reiser. Internet Performance Modeling: The State of the Art at the Turn of the Century. *Performance Evaluation*, 42(2–3):91–108, 2000.

[23] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. NetDiagnoser: Troubleshooting Network Unreachabilities using End-to-end Probes and Routing Data. In *Proceedings of the 2007 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2007*. ACM, Dec. 2007.

[24] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement 2007*. ACM, Oct. 2007.

[25] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments*. ACM, Mar. 2008.

[26] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*. USENIX Association, May 2005.

[27] N. Feamster, L. Gao, and J. Rexford. How to Lease the Internet in your Spare Time. *ACM SIGCOMM Computer Communication Review*, 37(1):61–64, Jan. 2007.

[28] A. Feldmann. NetDB: IP Network Configuration Debugger/Database. Technical report, AT&T Research, July 1999.

[29] A. Feldmann and J. Rexford. IP Network Configuration for Intradomain Traffic Engineering. *IEEE Network Magazine*, 15(5):46–57, September/October 2001.

[30] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007)*. USENIX Association, Apr. 2007.

[31] B. Fortz, J. Rexford, and M. Thorup. Traffic Engineering with Traditional IP Routing Protocols. *IEEE Communications Magazine*, 40(10):118 – 124, Oct. 2002.

[32] P. Francois and O. Bonaventure. Avoiding Transient Loops During IGP Convergence in IP Networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, Mar. 2005.

[33] P. Francois, M. Shand, and O. Bonaventure. Disruption Free Topology Reconfiguration in OSPF Networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications*. IEEE, May 2007.

[34] J. Fu and J. Rexford. Efficient IP-address Lookup with a Shared Forwarding Table for Multiple Virtual Routers. In *Proceedings of the 2008 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2008*. ACM, Dec. 2008.

[35] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007)*. USENIX Association, Apr. 2007.

[36] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*. USENIX, May 2006.

[37] GNS3. `http://www.gns3.net/`.

[38] GNU Radio: The GNU Software Radio. `http://www.gnu.org/software/gnuradio/`.

[39] Google. Google Chrome. http://www.google.com/chrome.

[40] C. Griffiths, J. Livingood, L. Popkin, R. Woundy, and R. Yang. Comcast's ISP Experiences in a Proactive Network Provider Participation for P2P (P4P) Technical Trial. RFC 5632 (Informational), Sept. 2009.

[41] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *5th USENIX Symposium on Networked Systems Design & Implementation*. USENIX Association, Apr. 2008.

[42] D. H. HA, T. Silverton, and O. Fourmaux. A Novel Hybrid CDN-P2P Mechanism for Effective Real-time Media Streaming. Master's thesis, Universite Pierre et Marie Curie, Sept. 2008.

[43] H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and Verification of IPSec and VPN Security Policies. In *13th IEEE International Conference on Network Protocols (ICNP 2005)*. IEEE Computer Society, Nov. 2005.

[44] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing Extensible IP Router Software. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*. USENIX Association, May 2005.

[45] U. Hengartner, S. Moon, R. Mortier, and C. Diot. Detection and Analysis of Routing Loops in Packet Traces. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement 2002*. ACM, Nov. 2002.

[46] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, M. Jordan, A. D. Joseph, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications*. IEEE, May 2007.

[47] Y. Huang, T. Z. J. Fu, D.-M. Chiu, J. C. S. Lui, and C. Huang. Challenges, Design and Analysis of a Large-scale P2P-VoD System. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Seattle, WA, USA, Aug. 2008. ACM.

[48] W. John and S. Tafvelin. Analysis of Internet Backbone Traffic and Header Anomalies Observed. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement 2007*. ACM, Oct. 2007.

[49] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. In *Proceedings of the 1st Annual ACM Workshop on Mining Network Data, MineNet 2005*. ACM, Aug. 2005.

[50] E. Keller and E. Green. Virtualizing the Data Plane Through Source Code Merging. In *Proceedings of the ACM SIGCOMM 2008 Workshop on Programmable Routers for Extensible Services of Tomorrow*. ACM, Aug. 2008.

[51] Z. Kerravala. Configuration Management Delivers Business Resiliency, Nov. 2002.

[52] Z. Kerravala. As the Value of Enterprise Networks Escalates, So Does the Need for Configuration Management, Jan. 2004.

[53] E. Kiciman and L. Subramanian. A Root-cause Localization Model for Large-scale Systems. In *Proceedings of Hot Topics in System Dependability (HotDep)*. USENIX, June 2005.

[54] M. S. Kim, T. Kim, Y.-J. Shin, S. S. Lam, and E. J. Powers. A Wavelet-Based Approach to Detect Shared Congestion. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Portland, Oregon, USA, Aug. 2004. ACM.

[55] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[56] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization Via Risk Modeling. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*. USENIX Association, May 2005.

[57] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and Localization of Network Black Holes. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications*. IEEE, May 2007.

[58] R. Krishnan, H. V. Madhyastha, S. Jain, S. Srinivasan, A. Krishnamurthy, T. Anderson, and J. Gao. Moving Beyond End-to-End Path Information to Optimize CDN Performance. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement 2007*. ACM, Nov. 2009.

[59] Kernel Samepage Merging. `http://www.linux-kvm.org/page/KSM`.

[60] R. LaFortune, C. D. Carothers, W. D. Smith, J. Czechowski, and X. Wang. Simulating Large-Scale P2P Assisted Video Streaming. In *42nd Hawaii International International Conference on Systems Science (HICSS-42 2009), Proceedings*. IEEE Computer Society, Jan. 2009.

[61] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving Convergence-Free Routing using Failure-Carrying Packets. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Kyoto, Japan, Aug. 2007. ACM.

[62] L. Leonini, E. Riviere, and P. Felber. SPLAY: Distributed Systems Evaluation Made Simple. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Apr. 2009.

[63] S. Liu, R. Zhang-Shen, W. Jiang, J. Rexford, and M. Chiang. Performance Bounds for Peer-assisted Live Streaming. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Annapolis, MD, USA, June 2008. ACM.

[64] Z. Liu, C. Wu, B. Li, and S. Zhao. Distilling Superior Peers in Large-Scale P2P Streaming Systems. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications*. IEEE, Apr. 2009.

[65] Z. Liu, C. Wu, B. Li, and S. Zhao. Why Are Peers Less Stable in Unpopular P2P Streaming Channels? In *Proceedings of the 8th International IFIP-TC 6 Networking Conference*, volume 5550 of *Lecture Notes In Computer Science*, Aachen, Germany, 2009. Springer.

[66] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet Path Diagnosis. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003*. ACM, Oct. 2003.

[67] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjalmtysson, and A. Greenberg. Routing Design in Operational Networks: A Look from the Inside. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Portland, Oregon, USA, Aug. 2004. ACM.

[68] J. Moy. OSPF Protocol Analysis. RFC 1245 (Informational), July 1991.

[69] S. Narain. Network configuration management via model finding. In *Proceedings of the 19th Conference on Systems Administration (LISA 2005)*. USENIX, Dec. 2005.

[70] S. Narain. Overview of Configuration Validation. Presentation at LISA 2006 Configuration Workshop, Dec. 2006.

[71] ns-3. `http://www.nsnam.org/`.

[72] A. Nucci, S. Bhattacharyya, N. Taft, and C. Diot. IGP Link Weight Assignment for Operational Tier-1 Backbones. *IEEE/ACM Transactions on Networking*, 15(4), Aug. 2007.

[73] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why Internet Services Fail and What Can Be Done About These? In *4th USENIX Symposium on Internet Technologies and Systems*. USENIX, Mar. 2003.

[74] N. Parvez, C. Williamson, A. Mahanti, and N. Carlsson. Analysis of BitTorrent-like Protocols for On-demand Stored Media Streaming. In *Proceed-*

ings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Annapolis, MD, USA, June 2008. ACM.

[75] R. Pasupathy. Generating Nonhomogeneous Poisson Processes. `http://filebox.vt.edu/users/pasupath/papers/nonhompoisson_streams.pdf`.

[76] PPLive User Comments. `http://tieba.baidu.com/f?kz=700224794`.

[77] Quagga Software Routing Suite. `http://www.quagga.net/`.

[78] B. Quoitin, S. Uhlig, and O. Bonaventure. Using Redistribution Communities for Interdomain Traffic Engineering. In *From QoS Provisioning to QoS Charging, Third COST 263 International Workshop on Quality of Future Internet Services, QofIS 2002*. Springer, Oct. 2002.

[79] M. Roughan, T. Griffin, M. Mao, A. Greenberg, and B. Freeman. Combining Routing and Traffic Data for Detection of IP Forwarding Anomalies. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, Poster Session*, New York, NY, USA, June 2004. ACM.

[80] Scalable Network Technologies. QualNet Developer. `http://www.scalable-networks.com/products/qualnet/`.

[81] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report dapper-2010-1, Google, Apr. 2010.

[82] Skype User Forums. `http://forum.skype.com/index.php?showtopic=95182`.

[83] A. Soule, K. Salamatian, and N. Taft. Combining Filtering and Statistical Methods for Anomaly Detection. In *Proceedings of the 5th Conference on Internet Measurement 2005*. ACM, Oct. 2005.

[84] M. Steinder and A. S. Sethi. A Survey of Fault Localization Techniques in Computer Networks. *Science of Computer Programming*, 53(2):165–194, Nov. 2004.

[85] D. Tang, A. Agarwal, D. O'Brien, and M. Meyer. Overlapping Experiment Infrastructure: More, Better, Faster Experimentation. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, July 2010.

[86] L. Tatman. Incorporating Routing Analysis into IP Network Management. `http://www.agilent.com/labs/features/2003_wp_roca.pdf`, May 2003.

[87] C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *Operating Systems Review*, 40(1), Nov. 2006.

[88] I. Ullah, G. Bonnet, G. Doyen, and D. Gati. Modeling User Behavior in P2P Live Video Streaming Systems through a Bayesian Network. In *Mechanisms for Autonomous Management of Networks and Services*, volume 6155 of *Lecture Notes In Computer Science*. Springer, 2010.

[89] University of Oregon. University of Oregon Route Views Project. `http://www.routeviews.org/`.

[90] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kosti, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *5th Symposium on Operating System Design and Implementation (OSDI 2002)*. USENIX Association, Dec. 2002.

[91] VideoLAN - VLC media player. `http://www.videolan.org/`.

[92] K. V. Vishwanath and A. Vahdat. Evaluating Distributed Systems: Does Background Traffic Matter? In *Proceedings of the 2008 USENIX Annual Technical Conference*. USENIX Association, June 2008.

[93] F. Wang, Z. M. Mao, J. Wang, L. Gao, and R. Bush. A Measurement Study on the Impact of Routing Events on End-to-End Internet Path Performance. *ACM SIGCOMM Computer Communication Review*, 36(4), Oct. 2006.

[94] WebKit2. http://trac.webkit.org/wiki/WebKit2.

[95] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating System Principles*. ACM, Oct. 2001.

[96] A. Whitaker and D. Wetherall. Forwarding Without Loops in Icarus. In *Proceedings of Open Architectures and Network Programming*. IEEE, June 2002.

[97] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *5th Symposium on Operating System Design and Implementation (OSDI 2002)*. USENIX Association, Dec. 2002.

[98] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, Mar. 2005.

[99] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider Portal for Applications. In *Proceedings of the ACM SIGCOMM*

2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, Aug. 2008. ACM.

[100] A. Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? *ACM SIGSOFT Software Engineering Notes*, 24(6):253–267, Nov. 1999.

[101] B. Zhang, T. S. E. Ng, A. Nandi, R. Riedi, P. Druschel, and G. Wang. Measurement-Based Analysis, Modeling, and Synthesis of the Internet Delay Space. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement 2006*. ACM, Oct. 2006.

[102] B. Q. Zhao, J. C. Lui, and D.-M. Chiu. Exploring the Optimal Chunk Selection Selection Policy for Data-Driven P2P Streaming Systems. In *Proceedings P2P 2009, Ninth International Conference on Peer-to-Peer Computing*. IEEE, Sept. 2009.

[103] Z. Zhong, R. Keralapura, S. Nelakuditi, Y. Yu, J. Wang, C. nee Chuah, and S. Lee. Avoiding Transient Loops through Interface-Specific Forwarding. In *Quality of Service - IWQoS 2005: 13th International Workshop*. Springer, June 2005.

[104] Y. Zhou, D.-M. Chiu, and J. C. Lui. A Simple Model for Analyzing P2P Streaming Protocols. In *Proceedings of the IEEE International Conference on Network Protocols, ICNP 2007*. IEEE, Oct. 2007.