The Dissertation Committee for Taylor Louis Riché
certifies that this is the approved version of the following dissertation:

The Lagniappe Programming Environment

Committee:

Harrick M. Vin, Supervisor

Gregory Lavender, Co-Supervisor

Micheal Dahlin

Don Batory

Raj Yavatkar

The Lagniappe Programming Environment

by

Taylor Louis Riché, B.S.E.; M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2008

To Carrie, Robert, and Bobbie Riché

Acknowledgments

First, I want to begin my thanking my wife, Carrie. Her love and support is unbounded. Specifically, I want to thank her for one thing that relates directly to this dissertation. I was working at IBM in 2001 when I was accepted into the doctoral program in the Department of Computer Sciences at the University of Texas at Austin. I was somewhat hesitant to quit my job; I honestly had not expected to get in. I figured I would get into the Masters program and go back part-time.

Everything changed when I got that acceptance letter. I had been given the opportunity to pursue my dream that I had had since I was 13: to get a Ph.D. and become a professor. Still, I was not sure. I remember telling Carrie that I had been accepted. We were not dating at the time, but we were close friends, and she was so happy for me. It was this excitement that helped reignite my own dream and made me realize that we only get to live the dreams we follow. I remember telling her that I had decided to accept the offer, quit my job, and come back to school full time. She seemed proud and happy that I made that decision, and seeing her response filled me with joy. We started dating my first year of graduate school and got married two years later. It has been the best four and a half years of my life since March 13, 2004. I dedicate this dissertation to her, but she shares that line with two other important people.

My parents, Robert and Bobbie Riché, have always supported me throughout my life. I was not a tough kid or an athletic kid. I was a smart kid who liked to

talk a lot. My parents always listened, though. My parents always encouraged me to learn more. I thank them for instilling in me the passion for learning that has pushed me this far. I also thank them for teaching me that doing the right thing is right, regardless of how much it may hurt or inconvenience you. Most of all, I thank them for their constant love; without that I most definitely would not be here right now.

I would be amiss if I did not thank my adviser, Harrick Vin. The most important thing that Harrick has taught me is to always ask the question, "Why?" It is the perpetual asking, and answering, of this question that sets us apart as scientists. I recently had the opportunity to talk to Harrick at length about his time here at UTCS. Not only am I forever grateful for the lessons he has taught me specifically, I am very grateful for the hard work that he put into the last 15 years to make our department a wonderful place.

There is a chance that I may be Harrick's last doctoral student. While on some level I am honored, I feel more sadness about this fact than anything. I really hope one day I hear that I am no longer his last student, because UT is losing a great teacher and adviser.

Greg Lavender, my co-adviser, deserves my utmost thanks. During several rough parts of my career here, Greg was there to remind me of the positive things and help to get my spirits up and to keep me going. Also, I thank Greg for introducing me to the research areas of programming languages and software engineering. I knew nothing of these things. Greg's instruction in these areas has helped me develop a passion for an entire area of research I did not even really know about when I started.

I would like to thank all of my committee members for their insightful comments and suggestions: Mike Dahlin, Don Batory, and Raj Yavatkar. I hope to continue to work with all three of these great researchers and teachers and to continue to learn from them as well.

I thank Lorenzo Alvisi for continuing to remind me why I want to be a professor.

There are five graduate students that I have had the honor of knowing during the time I spent working on this dissertation—I want to thank them now. Ravi Kokku and Jayaram Mudigonda were mentors to me, and two people from whom I have learned so much. Jeff Napper, Harry Li, and Allen Clement have become some of my best friends and have continued to push me to work harder and smarter. Conversations with a white board and these five people have helped to push many of the ideas in this dissertation forward.

I thank all of LASR. This research group has been an amazing place to work and learn. Seeing it develop from a relatively small group to what it is today has been a great experience. Further, I thank Sara Strandtman for keeping LASR running. Nothing would actually work without Sara.

Finally, I thank God for giving me so many blessings in life. Reconciling faith and science is a tough thing to do, and it is always a work in progress. I feel that God wants us to learn and never stop asking questions. Treating faith as fact does not do the beautiful mystery that is faith justice, and using religion as a tool to judge and demean goes against everything God teaches us. I believe that God gives us all two very important things: brains and love. I intend to use both of these to their fullest extent for the rest of my life.

Taylor Louis Riché

The University of Texas at Austin
August 2008

The Lagniappe Programming Environment

Publication No. ____________________

Taylor Louis Riché, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Harrick M. Vin

Multicore, multithreaded processors are rapidly becoming the platform of choice for designing *high-throughput request processing applications.* We refer to this class of modern parallel architectures as *multi-⋆ systems.* In this dissertation, we describe the design and implementation of *Lagniappe*, a programming environment that simplifies the development of portable, high-throughput request-processing applications on multi-⋆ systems. Lagniappe makes the following four key contributions: First, Lagniappe defines and uses a unique *hybrid programming model* for this domain that separates the concerns of writing applications for uni-processor, single-threaded execution platforms (single-⋆ systems) from the concerns of writing applications necessary to efficiently execute on a multi-⋆ system. We provide separate tools to the programmer to address each set of concerns. Second, we present *meta-models of* *applications and multi-$\star$ systems* that identify the necessary entities for reasoning about the application domain and multi-$\star$ platforms. Third, we design and implement a platform-independent mechanism called the *load-distributing channel* that factors out the key functionality required for moving an application from a single-$\star$ architecture to a multi-$\star$ one. Finally, we implement a platform-independent *adaptation framework* that defines custom adaptation policies from application and system characteristics to change resource allocations with changes in workload. Furthermore, applications written in the Lagniappe programming environment are *portable*; we separate the concerns of application programming from system programming in the programming model. We implement Lagniappe on a cluster of servers each with multiple multicore processors. We demonstrate the effectiveness of Lagniappe by implementing several stateful request-processing applications and showing their performance on our multi-$\star$ system.

Contents

List of Figures

Chapter 1

Introduction

Moore's law and the accompanying improvements in fabrication technologies ($90nm$ to $65nm$ and beyond [23]) have increased significantly the number of transistors available to processor designers. Processor designers use these transistors to develop architectures with multiple, multithreaded cores. System designers have begun utilizing multiple (possibly heterogeneous) multicore processors to design *high-throughput request processing systems.* We refer to this entire class of modern parallel systems as *multi-$\star$ systems.*

Figure 1.1 shows an example multi-$\star$ system. The example shows a blade system (i.e., several main-boards that share a chassis with a common power infrastructure and a high-speed backplane) that has four blades. Three of the blades, blades 1 through 3, have two dual-core processors. Each core also has four hardware contexts, or threads, in it as well. Blade4, however, represents a highly parallel graphics card with many small cores that communicate to each other through a mesh network. This example is not far fetched. Apple's soon to be released library, OpenCL [7], will expose the parallel resources of graphics processors to programmers and allow them to schedule computational tasks to the graphics processor. While there is obvious heterogeneity in this example with processor architectures, communicating between processors also has drastically different costs. Take for example message passing between the two processors on Blade1 versus a processor on Blade1 sending a message to a processor on Blade2.

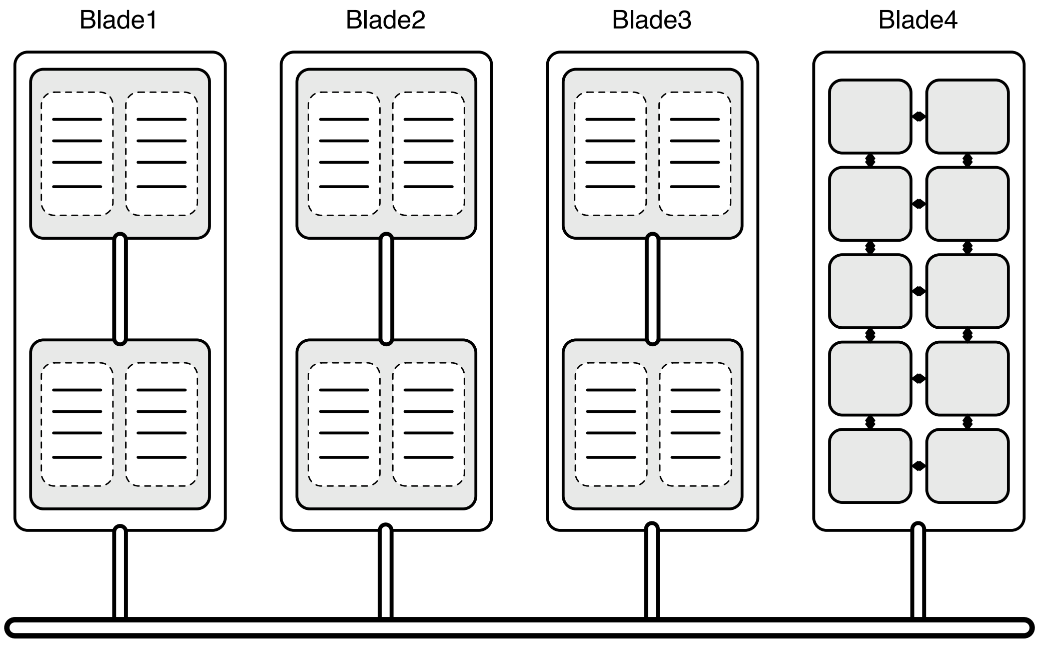

Figure 1.1: An example multi-$\star$ system with three general-purpose blades and one highly-parallel graphics blade.

Multi-$\star$ systems initially were designed to meet the demands of specific domains, such as networking [5, 22], graphics [36], and interactive games [14, 27]. For instance, in networking, multi-$\star$ systems have been used to design multiservice routers for GENI [44], Virtual Private Network (VPN) gateways, intrusion detection systems [8], content-based load distribution, and protocol gateways (IPv4/v6 gateway [43]). Now, multi-$\star$ systems rapidly are becoming the de facto platforms for many general-purpose high-throughput computing environments (e.g., web, database, and application servers) [15, 16, 34, 46, 47]. Engineers build these systems using the emerging class of general-purpose multicore processors from Sun [9], Intel [6], and AMD [1], among others.

1.1 Challenges of Multi-$\star$ Programming

Although multi-$\star$ systems are becoming commonplace, programming environments that simplify the development of portable, high-throughput applications on these systems has lagged behind. Multi-$\star$ systems are difficult to program for the following reasons:

- A request-processing application can be mapped onto a multi-$\star$ system in at least three different ways. The *pipeline* approach splits an application into independent stages and maps each stage to a processing element; thus, each request during its lifetime traverses multiple processing elements [47]. The *parallel* approach lets each element process a request from start to finish; processing elements available in a multi-$\star$ system process multiple requests in parallel. Finally, the *hybrid* approach replicates some parts of the application while staging others. Choosing the approach that delivers the highest throughput is difficult because the choice depends upon application, system, and workload characteristics [40].

- Most request-processing applications are stateful. The persistent state maintained by these applications is accessed and updated by a stream of related requests, referred to as a *flow*. In a multi-$\star$ system with multiple distributed memory levels and message-passing channels, providing efficient and coherent access to shared state is challenging. Further, the non-uniform memory architectures of many multi-$\star$ systems complicate the selection of an appropriate policy (e.g., request-level distribution vs. flow-level pinning) for distributing requests across processing elements [40].

- A request-processing application generally processes multiple types of requests. In most realistic deployments, however, the workload (both the composition of request types and volume of traffic) fluctuates significantly over time. Hence, a request-processing system must adapt resource allocations dynamically [32].

- Multi-$\star$ systems often contain a significant amount of heterogeneity–multiple types of processor cores with support for many different mechanisms for inter-processor communication and data sharing (e.g., hardware-supported coherent memory vs. explicit state-sharing and message-passing mechanisms). Today, some programming environments for multi-$\star$ systems expose these details to programmers. Exposing details, as is currently done, not only complicates the task of developing applications, but also makes applications not portable.

1.2 Programming Environment Requirements

Given the challenges a programming environment for multi-$\star$ systems must conquer, any new programming environment that hopes to successfully face these challenges must meet the following requirements in its design and implementation:

- Automate the mapping of applications onto multi-$\star$ systems;
- Efficiently utilize the available processing resources in a multi-$\star$ system using appropriate request distribution and persistent state management mechanisms;
- Dynamically and transparently adapt resources allocated to applications to match fluctuations in workload; and
- Achieve all of the above without exposing any multi-$\star$ system hardware details to programmers (thereby simplifying programming and ensuring portability of applications).

1.3 Contributions of the Dissertation

In this dissertation, we describe the design and implementation of the *Lagniappe programming environment*[1], which simplifies the development of portable, high-throughput request-processing applications on multi-$\star$ systems. Lagniappe meets the requirements we establish. We make the following four key contributions:

1. We define a unique *hybrid programming model.* This model separates the concerns of writing applications for uni-processor, single-threaded execution platforms (single-$\star$ systems) from the concerns of writing applications for execution on a multi-$\star$ system. We provide separate tools to the programmer to address each set of concerns. We further separate the concerns of application and system programming, making applications written in Lagniappe inherently portable.

2. We present *meta-models of applications and multi-$\star$ systems* that identify the necessary entities for reasoning about the application domain and multi-$\star$ platforms. Programmers use these meta-models to define instance models of their particular applications and platforms.

3. We design and implement a platform-independent mechanism called the *load-distributing channel* that factors out the key functionality required for moving an application from a single-$\star$ architecture to a multi-$\star$ one.

4. We design and implement a platform-independent *adaptation framework* that defines custom adaptation policies from application and system characteristics to change resource allocations with changes in workload.

We implement Lagniappe on a cluster of servers each with multiple multi-core processors. We demonstrate the effectiveness of Lagniappe by implementing

[1]Lagniappe is a Cajun-French word for "a little something extra." If programmers give us a relatively small amount of extra information, we can give them much added benefit.

several stateful request-processing applications and showing their performance on our multi-$\star$ system.

The dissertation is organized as follows. We present a brief overview of the application domain in Chapter 2. In Chapter 3, we discuss the state of the art and that no current solution meets all the requirements we present. In Chapter 4, we define our hybrid programming model. Chapter 5 describes the design, implementation, and operation of the Lagniappe programming environment. Chapter 6 describes our test applications and the setup and results of our experiments. Finally, we conclude in Chapter 7.

Chapter 2

Target Application Domain

In this chapter we describe in detail the Lagniappe target application domain: request-processing applications. We specifically focus on networking applications in this thesis; however, the properties we describe in this chapter apply generally to all request-processing applications. Figure 2.1 shows an application we call the attack detector. This application examines incoming requests looking for patterns that would suggest an attack is occurring. By attack we could mean worm propagation, distributed denial of service, or simple “hacking.” The exact definition is not important for this example.

We use this example throughout the section to help explain the properties of the domain. The basic behavior of the application is as follows. Requests enter the *Classifier* and are marked as either good, suspicious, or bad. Good requests move to the *Shallow Inspect* function and it performs a simple test to determine if the request is part of an attack. If request is not an attack, the application sends the request out. If *Shallow Inspect* thinks the request may be part of an attack, notifies *Classifier*. *Deep Inspect* performs a more computationally intense check on all requests related to any suspicious requests from *ShallowInspect*. If *Deep Inspect* performs a test that fails, it notifies *Classify* and all future related requests are dropped by *Classify*. Furthermore, *Deep Inspect* notifies *Mitigate* which tries to stop the newly-discovered attack.

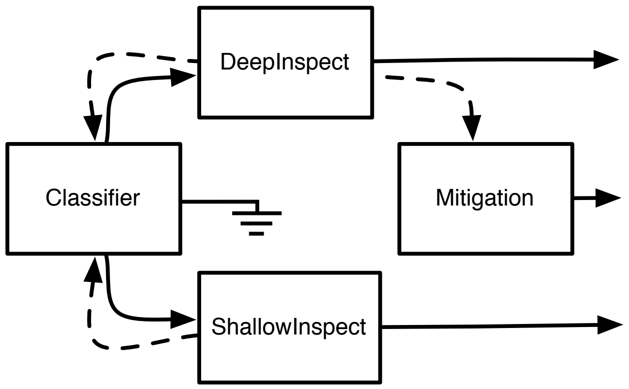

Figure 2.1: The attack detector, a sample application showing several properties of request-processing applications.

2.1 Application Structure

Request-processing applications are commonly constructed of a directed graph of separate functions. The Click modular router [31] proposed this graph construction as a way to provide modularity to a traditionally low-level domain. The figure shows that our attack detector example has four of these functions: *Classify*, *Shallow Inspect*, *Deep Inspect*, and *Mitigate*. A request enters the application at the head of the graph, and passes from one function to the next until a function either *drops* the request (i.e. the system reclaims the resources it assigned to request), or the request exits the application graph. The attack detector shows both of these cases, with the *Classifier* dropping some requests and sending others along.

In this thesis, we refer to an application as a collection of *operators*. Requests flow through *channels* between the operators. Each operator in the application may contain multiple inputs and outputs, and each operator may generate zero, one, or multiple requests while it processes an incoming request. Operators also tend to send two major classes of requests between themselves. The first type is *data requests*, these are the requests that the network injects into the application. The second type of requests are *control requests*. Operators send these control requests between themselves to relay status information. In our attack detector example, the solid lines in Figure 2.1 represent the transmission of data requests, while the dotted lines represent the transmission of control requests alerting other operators of an attack.

2.2 Request Characteristics

In some applications, requests may be unrelated. In other words, the processing of one request has no impact on the processing of any other request in the system. However, many applications process *flows* of requests. We define a flow to be any sequence of requests (not necessarily adjacent in arrival time) that share some property. We define a *flow signature* to be a function that takes requests to *flow identifiers*. Any requests that a flow signature maps to the same flow identifier are said to be in the same flow.

Note, an application may contain several different flow signatures. Each operator may have its own flow signature, as each operators need not be from the same application designer, and thus, can have different requirements for defining a flow. In our example, there is only one flow definition. *Shallow Inspect* and *Deep Inspect* send flow identifiers to *Classifier* and *Mitigate* when a request fails a test.

2.3 Operator State

The request-processing functionality of an operator may access “local” state that exists only while the operator is executing a particular request. However, many request-processing applications maintain *persistent state*—state that exists beyond the processing of any one request. Accesses to persistent state can dominate the time it takes an operator to process a request [37]. Persistent state is important in request-processing applications because the logic with an operator many times depends on the requests it has processed in the past.

Persistent state accesses fall into two classes. The first class of accesses do not depend on any type of relation between requests. The second class of accesses use flow identifiers as keys to map to particular data. We call persistent state that the operator accesses by a flow identifier *flow state*. In the attack detector application, the *Classify* operator maintains a mapping of flow identifiers to state representing the status of the flow. The choice is dependent on the semantics of a particular application. For example, in the attack detector, *Shallow Inspect* may keep a list of strings that indicate a worm attack. This state would change very rarely, and is not indexed differently between flows of requests. However, the table in *Classifier* may change frequently as attacks are discovered, and is solely accessed using flow identifiers.

2.4 Application Performance

One does not measure the performance of a request-processing application by the number of executions per second it performs. Rather, application designers traditionally create request-processing applications to meet specific *throughput* or *latency* guarantees. The programmer may construct the application to process the maximum number of requests per second, thus maximizing throughput. Also, the programmer may construct the application to process a request in a minimum amount of time, which we refer to as request latency. While programmers typically try to maximize application performance, many times they must deploy request-processing applications in environments where the application has specific performance goals it must meet, possibly according to a business-driven service level agreement (SLA).

In these cases, the programmer does not care to maximize performance, but rather to meet the throughput or delay guarantees he or she made in the SLA. For example, the attack detector application is deployed to the internet gateway of a small company. While the application could possibly support the rate supported by the physical network (for example, 1Gbps), the company may have simply only paid for the application to support a throughput of 100 Mbps.

Chapter 3

State of the Art

The state of the art in programming environments for multi-$\star$ systems is varied and deep. In this chapter, we look at the work that has preceded Lagniappe and discuss no existing tool or environment meets all our requirements for a multi-$\star$ programming environment for request-processing. First, we look at the programming language and software engineering concepts upon which we build Lagniappe. We then focus on related programming environments and whether these environments meet our requirements. Next, we examine some run-time environments that deal with parallel software execution resources (e.g. threads and events). Finally, we look at general-purpose software-system support for multi-$\star$ platforms.

3.1 Underlying Concepts

3.1.1 Dataflow Programming

Dataflow programming has been used to map computation to parallel resources for many years, and Johnston et. al [29] present a survey of the main ideas and seminal pieces of work in dataflow research.

Lagniappe, however, differs from the traditional view of even coarse-grain dataflow models, as an operator in Lagniappe does not need to wait for all its incoming ports to have data to execute. An operator can execute any time a request reaches a port. Thus, the model-driven aspect of Lagniappe does not define a dataflow graph in the traditional sense, but more accurately acts as a coordination language [38].

3.1.2 Model-Driven Engineering

Model-driven engineering (MDE) is a software-development methodology where programmers create models of systems as opposed to executable code. MDE environments apply transforms to models that move one type of model to another. Traditionally, the programmer (or an environment) applies transforms to the models to create increasingly system-specific models, eventually applying a transform that creates executable code.

Modeling framework often start with a meta-modeling language. In other words, the framework provides a model that allows programmers to create models of their software system. Programmers first create platform-independent models (PIM) that represent the software system independent of any type of implementation technology or details. Ideally, using a transform definition language, the programmer can then write transform rules for moving this PIM to a platform-specific model (PSM) that defines the software system in terms of specific technologies and implementations that are available on a particular platform. Finally, a transformation is applied to the PSM to create executable code.

Note, the meta-model used to create the PIM could be written using a model. In fact, each level of the framework could be thought of as an instance of a higher-level model. Accepted practice is to limit this model hierarchy to four levels [30]: a meta-modeling language, a meta-model, PIMs, and PSMs.

Lagniappe makes several diversions from the traditional MDE approach in its purest form. We define PIMs of request-processing applications and multi-$\star$ systems. Programmers create their own PSMs manually—we do not provide a set of transformation to move from PIMs to PSMs. However, we do provide a series of transformations that translate the PSMs of both the applications and the systems into executable code.

Another diversion from the traditional model in Lagniappe is the conscious decision to have programmers still provide procedural code along with their declarative model descriptions. We make this decision as we feel that the issues of single-$\star$ programming are well understood and are addressed by a large existing body of knowledge and implementations. However, it is in addressing the concerns of multi-$\star$ programming that can truly benefit from the high-level reasoning, ease of programming, and ease of reuse that a MDE approach provides.

3.2 Related Programming Environments

Related packet request-programming environments can be broken into two major groups: high-level and low-level environments.

3.2.1 High-Level Environments

High-level environments focus on the application features more than the architectural details. However, many of these environments do not truly achieve portability, nor do they take into account the important issue of persistent-state access in their designs.

Click [31] is the most well-known packet-processing programming environment. Click allows programmers to specify applications in terms of a connected graph of independent elements, Click initially was written for a single thread but follow-on work with MPClick [18] expands Click to utilize multiple threads. Click has no mechanisms for replicating elements. Thus, Click does not handle major workload changes that cannot be dealt with by moving elements from one processor to another to attempt to lower processor utilization. Click is written as a Linux module, with no real way to separate a Click application from the underlying Linux platform, and thus has no inherent support for processor heterogeneity beyond what Linux provides.

Observe that the notion of Lagniappe operators is very similar to the reusable elements defined in Click. In fact, one can convert many Click modules into Lagniappe operators with relative ease.

Flux [16] is a high-level environment that provides a combination programming environment in which the dataflow aspects of the system are modeled separately from the request processing. Flux can use any multithreaded software library for its underlying execution. The authors guarantee safe access to shared state by labeling atomic functions, and scheduling execution to guarantee atomicity. Flux does not meet our requirements as it only guarantees correctness of shared state but does nothing to provide efficient access. Also, Flux cannot adapt to any changes in workload as it is designed to be run on any available threading model.

A more recent environment is Aspen [45]. Aspen does not address the main issues that make multi-$\star$ development difficult, namely, state access, in the language design. While Aspen supports run-time adaptation, nothing guarantees efficient access to persistent state while balancing load among resources. In other words, Aspen does not take contention for shared state into account when scheduling resources.

Another recent environment, Merge [35] provides a programming environment for multi-$\star$ applications that fit into the map-reduce paradigm of side effect-free tasks. Merge provides a framework that schedules these tasks onto possibly heterogeneous resources by using a intermediate runtime library to mask the differences in resource implementation. While they show good results in both performance and programmability, the type of request-processing applications we focus on are not side effect free. Many request-processing applications do maintain persistent state that the application can possibly access and affect on every request. Merge does not meet all of our requirements as it focuses on a different, and specific, application domain.

Stream environments (such as Streamit [42] and Streamware [26]) provide programmers with new languages that focus on request processing. Stream programs are written in a fashion more similar to standard programming, i.e., no separation between coordination and execution. The stream compilers determine how to separate the functionality of the program (usually with keyword help from the programmer) into tasks that are then scheduled on the multi-$\star$ resources of the system. This approach is a much more fine-grained approach to parallelism. Stream languages work well in signal-processing applications and low-level network programs that are more computationally bound and less bound by accesses to persistent state. While stream programs do replicate operators, given the lack of tools to deal with persistent state, we feel that stream programming does not meet our requirements for the scope of applications upon which we focus.

3.2.2 Low-Level Environments

Low-level environments focus on particular architectures or architectural features, and are typically implemented to only work for a particular platform.

NesC [24] is a low-level dialect of C that specifically deals with the embedded restrictions of sensor networks; and thus, it does not provide enough flexibility for our needs. Nova [25] is a language specifically designed to be easy to compile for the Intel's IXP, a platform where hardware details are exposed explicitly to the programmer, which also keeps it from meeting our requirements. Baker [19] presents a slightly higher-level programming environment than NesC and Nova, but is specifically designed to compile to the IXP platform and exposes platform details to the programmer, thus losing portability.

3.3 Run-Time Environments

The literature contains several examples of run-time environments designed with request-processing applications in mind. Some of these environments include SEDA [47], Capriccio [46], and Tame [33]. These environments present APIs that allow programmers to create efficient and high-performance request-processing applications. However, while they do expose concurrency primitives to the programmer, they do not allow programmers to model the state-access patterns of the application and could possibly introduce unnecessary contention. As well, these systems, though multithreaded, are not explicitly designed to use multi-$\star$ platforms and the constraints that come from these platforms, specifically, the challenges of heterogeneity in the underlying resources.

3.4 Multi-$\star$-Aware Systems

Most modern operating systems are now aware of the underlying multi-$\star$ resources in their underlying platforms. Multicore processors are becoming ubiquitous enough to necessitate this functionality. However, even though most operating systems may support multi-$\star$ resources, at least on a local-machine level, for a programmer to use the resources effectively in a high-throughput request-processing application (i.e., guarantee efficient state access), a programmer must delve into the operating system APIs and manually assign the resources to the components of his or her application. Apple Computer is set to release a technology in their next operating system version that promises to allow more effective use of multi-$\star$ resources by automatically turning any application into a request-processing style application, thus increasing the inherent parallelism available [4]. However, Apple has released little technical information on this technology, which they are calling, “Grand Central,” so we cannot comment on whether it will meet our requirements.

Virtualization promises to provide an efficient way for programmers and system designers to take advantage of multi-$\star$ platforms. The basic premise is that all the specific details of the multi-$\star$ hardware is hidden under some number of virtual machines. Applications running these virtual machines know nothing of the underlying system details. Parallelism is achieved by running multiple copies of the virtual machine.

Systems such as VMWare [10] and XenSource [12] provide advanced virtual-machine functionality. VMWare’s VMotion product [11] allows for the migration of virtual machines across physical machines in a multi-$\star$ cluster. VMWare also can replicate virtual machines across physical machines or replicate virtual machines on the same machine. While virtualization addresses many of our requirements, it does still fall short of meeting all of them.

If VMWare replicates a virtual machine, VMWare does not handle the load-distribution. What policy to use, whether it be flow-pinning, round-robin, or some other policy is left up to a system administrator. No information from the application is automatically taken into account. Furthermore, the triggers for adaptation are based purely on processor utilization, and again have no application-specific knowledge. In a request-processing application that guarantees a certain delay bound, adapting because a processor has crossed a particular utilization threshold may be unnecessary. Whether a processor is running at 10% or 95% is inconsequential as long as the system processes requests within the delay guarantee.

Chapter 4

The Programming Model

We face a fundamental challenge in designing a programming model for request-processing applications on multi-$\star$ systems: the issues that concern single-$\star$ programming are distinctly different from the issues that concern multi-$\star$ programming.

Single-$\star$ issues are those that that refer to the request-processing functionality and the implementation of platform-resource, "drivers." Multi-$\star$ issues are those that refer to the coordination of operators, resources, and concurrent accesses to shared, persistent state.

For example, one single-$\star$ issue is the implementation of an operator's request-processing functionality. An associated multi-$\star$ concern is defining the access semantics of the operator, i.e., how does this operator access persistent state, and thus how should the system schedule multiple copies of the operator to reduce contention for shared state. Another example of a single-$\star$ issue is how a system programmer implements a processing resource, i.e. the functionality that makes the hardware resource available for scheduling by the operating system. A related multi-$\star$ issue is how many of these processing resources are in the system.

A programming model that forces programmers to face these challenges with the same tools has two potential dangers: 1) It could force the programmer to program at such a high-level that they lose the advantages of the specific underlying platform; or 2) Low-level mapping and coordination logic dominate the code, making it harder to write and maintain.

Lagniappe takes a fundamentally different approach, and provides a *hybrid programming model* that separates the concerns of single-$\star$ and multi-$\star$ programming and present to programmers different tools for addressing these distinct sets of concerns. Lagniappe provides programmers a traditional procedural environment for writing the single-$\star$ code. This environment allows programmers to implement the core request-processing and resource functionality in a language that is familiar to them. On the other hand, to address multi-$\star$ concerns, Lagniappe provides programmers a model-driven declarative environment that allows programmers to reason about the complicated multi-$\star$ concerns of coordination, state-access, and mapping at a high-level, creating program and system descriptions that are simple and promote reuse.

Furthermore, we separate the concerns of application and system programming by separating the programming of application functionality from system-resource implementation both at the procedural and declarative level. We separate these concerns to build portability into the system at a very basic level; i.e., application code is explicitly portable as no system resources are made visible to it.

In this chapter we first discuss the traditional procedural programming-model component in Section 4.1 and then we describe, in detail, the model-driven, declarative programming-model component in Section 4.2.

4.1 Procedural Component

The Lagniappe programming environment requires programmers to partition the request-processing application into a collection of *operators* that process each request (similar to the Click Modular Router [31]). Application programmers specify each operator using a standard procedural language (C++, in our implementation). Lagniappe places three requirements on the design of operators:

1. Operators must be *independent* of each other. The independence property is defined in terms of any persistent state that an operator may maintain. An operator is said to be independent of other operators if it does not access any persistent state maintained by the other operators. This independence property ensures that Lagniappe may execute these operators in parallel and without any contention for persistent state. This configuration simplifies the mapping of operators onto processors in a multi-$\star$ system.

2. Operator implementations must be *thread-safe*; access to persistent state maintained by the operator must be protected using locks and condition variables. This property ensures that multiple instances of operators can execute in parallel without violating correctness.

3. Each operator must import interfaces to initialize, access, install, and purge persistent state (as required by the application meta-model defined in Section 4.2.1) as well as implement a method that describes the state-access semantics of the operator. Information about persistent state allows Lagniappe to determine an appropriate request-distribution strategy across multiple instances of an operator and to migrate persistent state from one processor to another when the allocation of processors to operators dynamically changes.

In addition to these three hard requirements, Lagniappe recommends that these operators be as small in granularity as possible. This recommendation has two benefits: 1) Small granularity operators are often easier to reuse and 2) The smaller the operator granularity, the greater the flexibility available to the Lagniappe run-time environment for distributing these operators across the parallel processing resources available in the multi-$\star$ system.

In addition to the procedural definition of application functionality, Lagniappe requires that the programmer provide procedural implementations of the underlying multi-$\star$ resources in a system. Lagniappe uses these system, "drivers," to access the functionality that a particular multi-$\star$ system provides. Unlike the operator definitions, Lagniappe puts no restrictions on the procedural resource implementations.

4.2 Declarative Component

As we discuss in Chapter 1, the design of Lagniappe is driven by four requirements:

- Automate the mapping of applications onto multi-$\star$ systems;
- Efficiently utilize the available processing resources in a multi-$\star$ system using appropriate request distribution and persistent state management mechanisms;
- Dynamically and transparently adapt resources allocated to applications to match fluctuations in workload; and
- Achieve all of the above without exposing any multi-$\star$ system hardware details to programmers.

Our goal with the declarative model is to identify the necessary entities for reasoning about executing request-processing applications on multi-$\star$ systems so that we can design a programming environment that meets the above requirements. Specifically, we strive to improve programmability by providing programmers a sufficient set of entities with which they can model their applications and the underlying multi-$\star$ systems.

We present to programmers a custom model-driven engineering (MDE) framework. For Lagniappe, we define two meta-models: (1) the application meta-model and (2) the system meta-model. These meta-models formalize the specification of application and system features necessary for efficiently executing an application on a multi-$\star$ system. Application developers and system designers, respectively, create instances of the application and system meta-models to describe specific applications and target multi-$\star$ systems.

Again, observe that we separate the specification of a target multi-$\star$ system from that of request-processing applications thus making the specification of Lagniappe applications completely portable. Further, explicit specification of system features allows Lagniappe to utilize transparently the most appropriate state-sharing and communication mechanisms (without exposing any of these system details to application programmers). By combining the instances of the application and system meta-models, Lagniappe automatically generates platform-specific code. We discuss this process in more detail with code examples in Chapter 5.

In what follows, we describe the Lagniappe application and system meta-models.

4.2.1 Application Meta-Model

The application meta-model in Lagniappe specifies: (1) an application as a composition of operators and (2) features of the persistent state maintained by each operator. The class-diagram representation of the application meta-model is shown in Figure 4.1.

Application Specification:

In Lagniappe, a programmer models a request-processing application as a directed graph of Operator entities. Requests flow through this graph with each Operator operating on the requests. This graph specification is used by Lagniappe to determine efficient mapping of Operators to processing resources such that the inter-operator communication overhead is minimized.

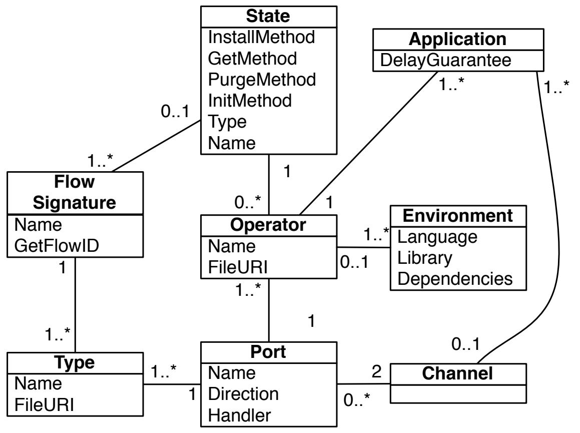

Figure 4.1: The Lagniappe application meta-model.

Formally, each Operator is associated with one or more Ports. Each Port has a *Direction*, which is specified as either *INCOMING* or *OUTGOING*. An *INCOMING* Port is associated with a handler function represented by the *Handler* property. These handlers, provided in the procedural specification, implement the request-processing functionality of the Operator. Each Port also is associated with a Type entity that defines the type of requests that flow through it. The execution environment required for each Operator is specified by the Environment entity.

An application is specified by connecting Ports of different Operators using Channel entities. These Channels define the channel of communication between Operators. We model a collection of Operators as an Application. An Application has a *DelayGuarantee* property that specifies the maximum delay in microseconds a request traversing the application can incur.

Operator State Specification:

The application meta-model includes specification of properties of persistent state maintained by the Operators. Lagniappe uses the specification of persistent state to: (1) facilitate concurrent execution of multiple instances of Operators on multiple resources available in a multi-$\star$ system; (2) determine an appropriate request distribution strategy across multiple instances of the same Operator; and (3) migrate the persistent state from one processor to another when the allocation of processors to Operators changes dynamically and the processors do not share hardware-coherent memory.

Formally, application programmers define persistent state by associating a State entity with each Operator. The application programmer must declare the type and name of the persistent state, as well as identify four methods—*Install*, *Get*, *Purge*, and *Init*—for accessing, installing, purging, and initializing persistent state. Note that these functions are black boxes to Lagniappe; application programmers must provide thread-safe implementations of these functions using the procedural specification.

To identify the flow to which a request may belong, Lagniappe requires programmers to specify a *GetFlowID* function in the Flow Signature specification to associate a unique flow ID with each request (the implementation of the *GetFlowID* function is provided as part of the procedural specification). Lagniappe uses this flow ID to route requests to appropriate instances of the Operator in the event that multiple instances of the Operator concurrently execute. This distribution of work minimizes contention for persistent state, and thus helps to fulfill the requirement to guarantee efficient state access automatically.

Multi-$\star$ Concerns

The application model captures two key multi-$\star$ concerns. The first is that of operator coordination and communication. We define the Port and Channel entities to model this communication. Furthermore, the Type entity provides typed-communication to the `Operator` entities, allowing the programmer to further restrict and describe the coordination between Operator entities.

The second multi-⋆ concern the application model addresses is that of concurrent access to persistent state. By providing to Lagniappe a `Flow Signature` entity, the programmer allows the system to make load-distribution and resource-allocation decisions based on that Operator entity's state-access semantics.

4.2.2 System Meta-model

A multi-⋆ system may contain one or more types of processors, several memory levels (shared vs. private as well as with or without support for hardware coherence), and one or more channels for interprocessor communication. The system meta-model formalizes the specification of all of these features of a multi-⋆ system. The class-diagram representation of the system meta-model is shown in Figure 4.2.

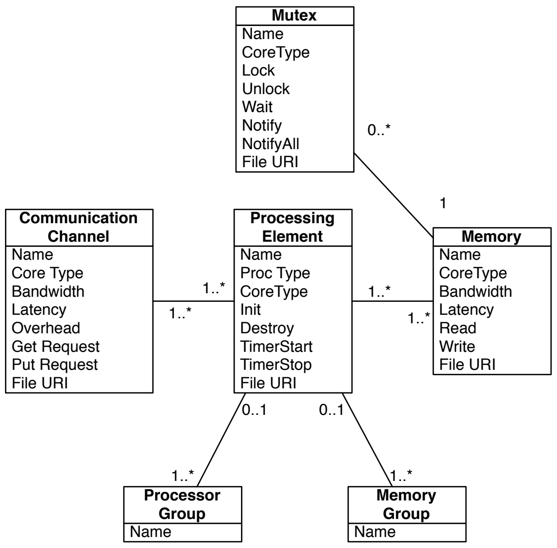

Figure 4.2: The Lagniappe system meta-model.

The Processing Element entity is the core of the system meta-model and represents the basic computational resource available in the system. The Processing Element entity is also the unit of resource allocation and deallocation. The *Name* property gives each Processing Element instance a unique identifier, and the *Proc*

Type property defines the type of the element (e.g., a Pentium or a SPARC core). Processing Element entities have two properties—*TimerStart* and *TimerStop*—that provide implementations that allow Lagniappe to measure the performance of a Processing Element implementation.

Two key multi-$\star$ concerns at the system level are the need to support multiple machines and the differences in efficiency in memory-coherency. We propose the Processor Group and the Memory Group entities to address these concerns, respectively. Each Processing Element can be a member of a Processor Group; all Processing Elements within a group are of the same type and require only one executable for the application. Similarly, each Processing Element can be a member of a Memory Group; all Processing Elements within a memory group share a hardware-supported coherent memory. Processing Elements between Memory Groups may still have access to software-supported coherent memory (for example, if an operating system exports only one memory space to all processors, regardless of hardware coherence). The *Init* and *Destroy* properties identify functions (whose implementations are provided as part of the procedural specifications) to initialize or stop a processing element.

The system meta-model also captures two paradigms for communication across Processing Elements: shared memory or message-passing channels. A Communication Channel entity represents a message-passing mechanism across processing elements. The *Name* property provides a unique identifier to a Communication Channel entity. Building on the LogP model of parallel computation [20], we define *Bandwidth*, *Latency*, and *Overhead* as properties that define the performance characteristics of a channel. The *GetRequest* and *PutRequest* properties define the methods used to interact with the channel. The *FileURI* property identifies the location where the procedural implementations for all of these functions can be found. The specification of the Memory entity is similar to the Communication Channel entity. The *Name*, *Bandwidth* and *Latency* properties specify basic features of a memory level, while *Read* and *Write* refer to the methods used for accessing the memory entity. Finally, each Memory entity is associated with a Mutex entity that the Memory uses to implement mutual exclusion. A system programmer provides implementations (using the procedural specification) for the *Lock*, *Unlock*, *Wait*, *Notify*, and *NotifyAll*.

Notice that a system designer creates an instance of the system meta-model and compiles it only once for each multi-$\star$ system. This model can be reused for all applications that execute on the system.

Chapter 5

The Programming Environment

While Chapter 4 describes the hybrid programming model that Lagniappe presents to programmers, in this chapter we describe the design and implementation of the Lagniappe programming environment.

5.1 Programming Environment Design

The Lagniappe programming environment has one main goal: to generate an executable application that is able to run on a particular multi-$\star$ system. The programmers provide procedural implementations of the single-$\star$ application and resource functionality as well as declarative descriptions of the multi-$\star$ concerns such as coordination and concurrent state access properties. In this section, we describe the three major transformations that turn these application and system instance models into platform-specific procedural code. The three transforms are:

1. Prepare application for execution on a multi-$\star$ platform.

2. Integrate resource adaptation framework with application.

3. Bind with application-specific and system-specific implementations.

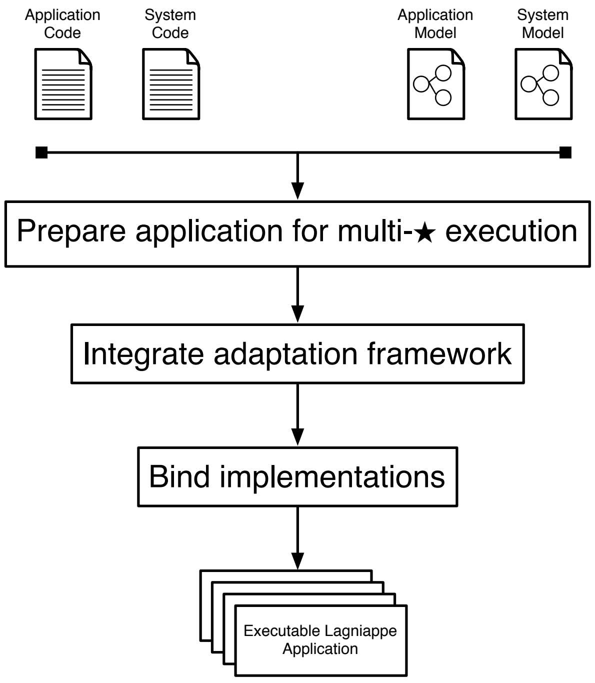
Application Code
System Code
Application Model
System Model
Prepare application for multi-★ execution
Integrate adaptation framework
Bind implementations
Executable Lagniappe Application

Figure 5.1: High-level architecture of the Lagniappe programming environment.

Figure 5.1 shows a high-level view of the Lagniappe programming environment, with the three transforms represented as black boxes. We now describe each of these transforms in more detail, and provide more insight into how each transform moves the models forward to a platform-specific multi-⋆ executable.

5.1.1 Prepare Application for Multi-⋆

There exist two facts that distinguish single-⋆ execution from multi-⋆ execution:

1. Operators may simultaneously run on multiple processing elements. We call each instance of an operator a *replica*.

2. The numbers of replicas necessary to meet application performance demands may change over time.

These facts imply interoperator communication must support the following three features:

1. *One-to-many communication*: Lagniappe may replicate operators during runtime. Thus, an operator may have to send a request to not just one instance of the next operator in the graph, but one of possibly many replicas.

2. *Changing the number of replicas*: Workload changes in request-processing applications, and thus the number of replicas necessary to meet the performance requirements of an application must also change. Therefore, the one-to-many channel must support changing the number of destination replicas during runtime.

3. *Multiple load-distribution policies*: One of our requirements is to guarantee efficient state access and reducing contention for shared state is a method we use to achieve this requirement. The state-access semantics define the most efficient load-distribution policy. For example, if an operator has flow state, a strategy called flow-pinning, where the processing of flows is scheduled to processing elements, may be the most efficient as requests from different flows do not contend for state access. However, operators without flow state may benefit from the lower overhead of a round-robin approach.

We introduce the *load-distributing channel*(LDC) mechanism to meet the previous requirements of interoperator communication. The LDC is a platform-independent mechanism and is a one-to-many, output-buffered communication channel that can distribute requests across recipients based on multiple load-distribution policies. The LDC is adaptive; it supports changing the number of destination operators during runtime.

The first transformation of the user-provided application model is to replace all connections between operators with LDCs. This transformation imbues the application with the mechanisms to support adaptation and the concurrent execution that can occur in a multi-$\star$ environment.

5.1.2 Integrate the Adaptation Framework

In the previous section we discuss a powerful mechanism, the load-distributing channel, that factors out the main difference between single-$\star$ and multi-$\star$ execution. However, the LDC is just a mechanism. Lagniappe must still make several policy decisions. The second transform the Lagniappe programming environment performs is to integrate the adaptation framework(AF) into the application, generating policies that control the LDC. Note, while the LDC is application-independent and platform-independent, the final policy decisions are neither. However, the power of the AF is that it automatically determines appropriate policies based on the models that the programmers provide. To provide this functionality, the AF addresses the following three questions:

1. *How many replicas of an operator need to execute?*: To address this question, the AF monitors the rate that load increases for an operator. LDCs support meter functionality, and this allows the AF to assign resources more aggressively in cases of quick, large changes and less aggressively to handle small shifts.

2. *When does Lagniappe change the number of replicas?* To address this question, the AF monitors the queue of requests waiting to be serviced by each operator. The queuing information for each recipient operator is exported to the Lagniappe library by the LDCs. The AF uses one of several preconfigured policies to trigger adaptation. For instance, the AF triggers adaptation if: (1) the queue length for a recipient operator consistently exceeds a threshold (determined by the delay tolerance of each operator); or (2) the queue for an operator remains empty consistently for extended periods of time. Changes

in workload *volume* or *composition* can cause these changes in the amount of work for an operator.

3. *Where do these replicas execute?*: Once an event triggers adaptation, the AF determines the new resource allocation strategy. The strategy depends upon the current state of resource allocations and the adaptation trigger. Consider, for instance, the case where adaptation is triggered when the queue for an operator instance consistently exceeds a threshold. In this case, if the system already executes multiple instances of this operator and the queues for other instances are consistently below the threshold, then the overload at the operator instance can be handled simply by adjusting the strategy for distributing load across the multiple operator instances. On the other hand, if the system is executing only one instance of the operator, then the overload can be handled either by increasing the share of processor capacity being allocated to the operator or by creating a new operator instance on another processor. In the latter case, the AF determines on which processor to allocate executing the new instance of the operator.

Note that the question: "when to adapt?" is a function of application requirements (e.g., delay or throughput bounds) and the system performance. On the other hand, the question: "where?" depends only on system characteristics (e.g., the available number of processing elements and the grouping of processing elements in memory and processor groups) and is independent of application features. However, the question of "how many?" is a function of the changes in workload and the relative performance of remaining system resources.

5.2 Programming Environment Implementation

Figure 5.2 shows the high-level architecture of the Lagniappe programming environment in more detail. We now see several additions from Figure 5.1. First, notice that we implement the LDC and AF in the Lagniappe library. The Lagniappe library is a platform-independent library that implements these two key components of the environment as well as several other mechanisms necessary to the final operation of the environment.

The three major transformations: 1) Preparing applications for multi-$\star$ by adding LDCs; 2) Integrating the AF; and 3) Binding the implementations all occur in the Lagniappe compiler. In this chapter we discuss the implementation details of both the Lagniappe library and the Lagniappe compiler.

5.2.1 Lagniappe Library Implementation

In this section, we examine the abstract class hierarchy that composes the Lagniappe library and from which programmers derive their application and system implementations. We also discuss in detail the implementations of several necessary platform-independent mechanisms.

Library Classes

Figure 5.3 shows the class hierarchy of the application-oriented components of the Lagniappe library. Note that several of the classes are abstract (indicated by the italicized names). These classes cannot operate on their own; Lagniappe presents these as the interfaces that the provided application code implements. However, it is the definition of these interfaces that allows us to implement the Lagniappe mechanisms in a way that is completely application-independent.

The `Application` class is the main container for a programmer-provided application; it contains a list of references to `CoreOp` instances. The introduction

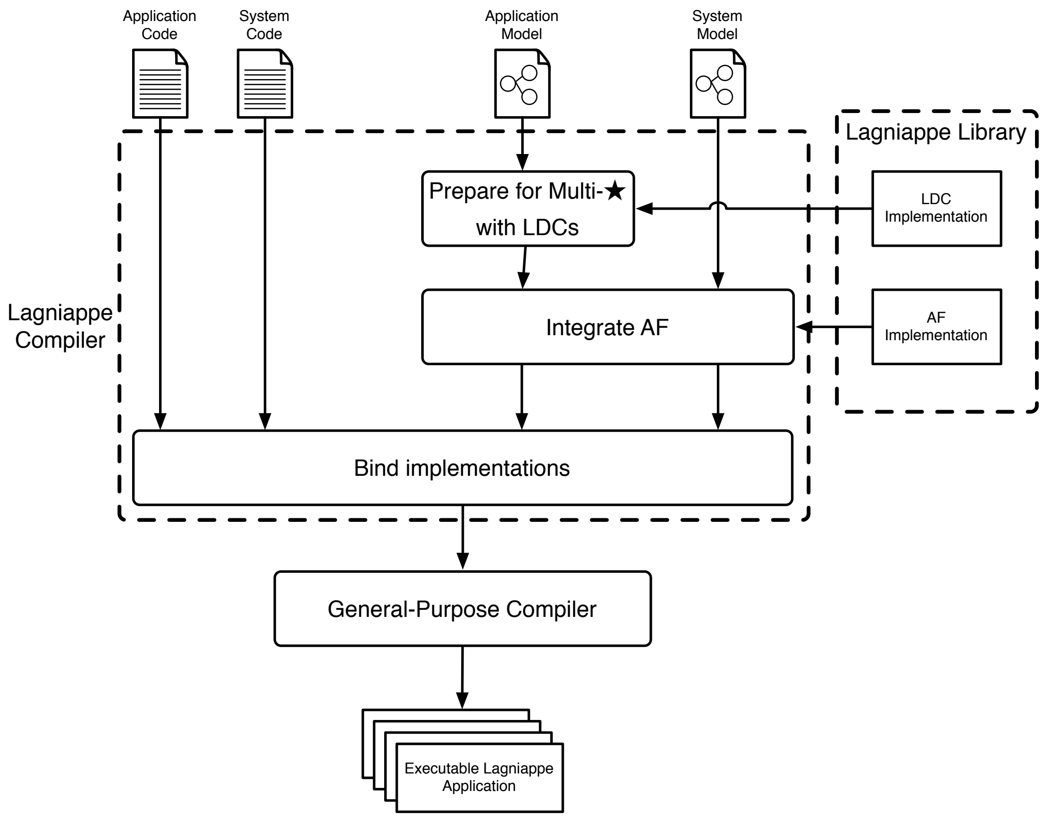

Figure 5.2: High-level architecture of the Lagniappe programming environment.

of the `CoreOp` class allows us to implement two types of abstract operators. The first is the `Operator` class. The `Operator` class is the basic parent class of all the operators that the Lagniappe compiler generates. All programmer-provided operators inherit from (and implement the interfaces of) `Operator`. `RequestGenerator`, however, has a different intention. We provide `RequestGenerator` to provide a wrapper class for the system-specific implementation of request-generating devices, whether that be a driver for the ethernet device that generates and sends network packets or socket-processing code that reads and writes http requests and responses. The `RequestGenerator` class provides a clean way for an application programmer to interface with the underlying system, without having to know any implementa-

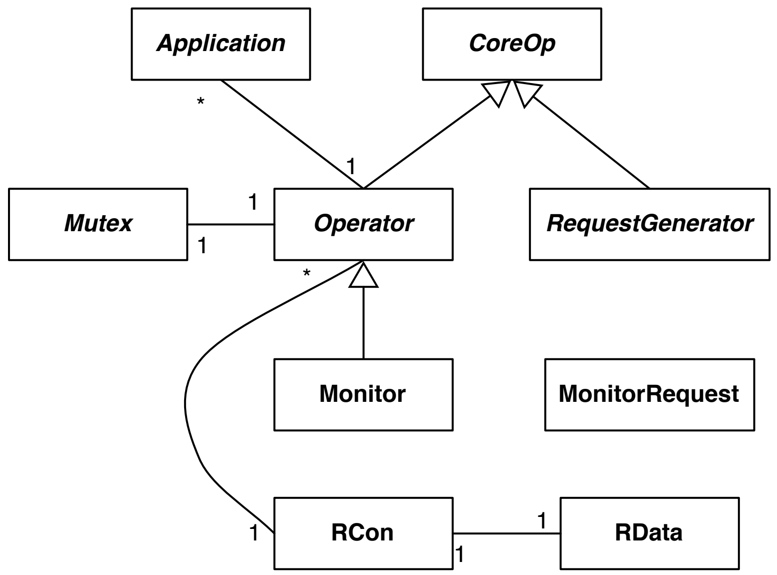

Figure 5.3: Class hierarchy of the application-oriented components in the Lagniappe Library.

tion details. Lagniappe treats `RequestGenerator` objects as “black boxes” during runtime, passing requests to and from them as it would to any other `CoreOp` implementation.

Each `Operator` implementation contains a reference to its own `Mutex` implementation. While the `Mutex` abstract class is actually a system component that the system programmer provides, `Operator` objects have access to this resource to protect any shared state that their handlers may access. Note, the `Operator` only references the `Mutex` abstract class, thus providing a constant interface to the application programmer regardless of the underlying implementation of mutual exclusion on the platform.

The `RCon` class implements the request continuations we discuss in Section 5.2.1. This class is a simple container comprising a reference to the `Operator` class, the name of the specific input Port entity from the application model, and the `RData` type. Note, we do not implement a class that represents the Port entity from the application model; the name of the Port is sufficient to find the right handler code to execute.

RData is a type wrapper around `void *`. We implement RData so simply to account for the large variation in request data formats across the full spectrum of request-processing applications.

Finally, the Monitor class implements the resource-assignment mechanism and adaptation framework. We discuss this implementation in more detail in Section 5.2.1. The MonitorRequest class is a request data type that contains the different resource requests and responses that the Monitor class can receive and send, respectively.

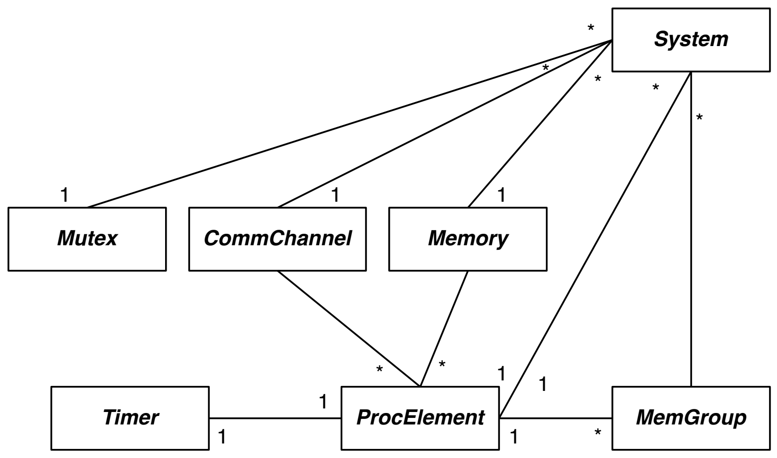

Figure 5.4: Class hierarchy of the system-oriented components in the Lagniappe Library.

Figure 5.4 shows the class hierarchy of the system-oriented components in the Lagniappe library. Note that for the system-oriented half of the Lagniappe library, all the classes are abstract. However, these system classes do implement a large amount of functionality related to the Lagniappe mechanisms. The classes in Figure 5.4 all define interfaces for the resource implementations as well as the system-independent implementations of the Lagniappe mechanisms.

The class structure closely follows the system meta-model from Section 4.2.2. For each of the major system entities, there exists a corresponding class definition. The `System` class is the main container for all the system entities.

The `ProcElement` class represents the main computational resource in a multi-$\star$ system. Each `ProcElement` instance can call upon a `Timer` implementation for profiling purposes. Each `ProcElement` contains references to the `CommChannel` and `Memory` implementations it uses for inter-processor communication.

The `MemGroup` objects contain references to the `ProcElement` objects within them. Note that there is no class to represent processor groups; the compiler uses these groups only during compile time. The effects of two `ProcElement` instantiations being in two different processor groups is noticeable in the relative cost of intergroup versus intragroup communication mechanisms (e.g. `CommChannel` implementations).

Request-Execution Engine

The request-execution engine is the mechanism that allows the main computational element of Lagniappe, the processing element, to process request data using the operator code that the programmer provides. To perform this task efficiently, we introduce the *request continuation* as the main unit of computation in the Lagniappe programming environment.

The request continuation builds on the mathematical concept of a continuation, a formalism designed to model "the rest of the computation" in an application [39, 41]. Lagniappe models computation as a directed graph of operators, thus, the subgraph that starts with the current operator and contains all of its children contains all the possible computation that request could encounter. Therefore, given an application subgraph (starting with the entry point of one operator) and the data from the actual request, a processing element has all the information necessary to execute that operator on the request data. Figure 5.5 shows a graphical representation of a request continuation. As the request data (the gray box) moves through the application, the possible computation remaining (represented by the double outline operators) changes.

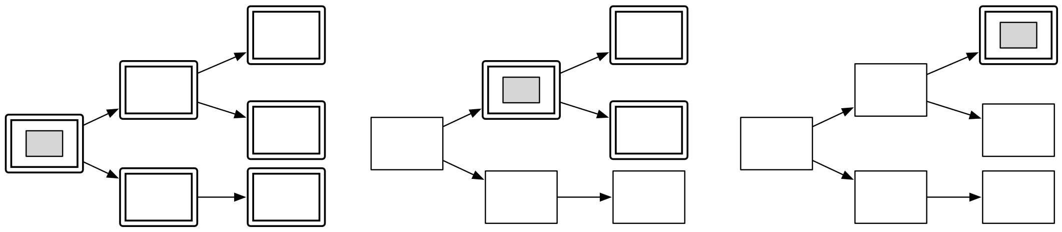

Figure 5.5: The evolution of the request continuation.

Computation in Lagniappe is straightforward. A processing element entity removes request continuations from the, possibly many, load-distribution channels that feed into it. The processing element now has an operator and request data, both of which it extracts from the request continuation. The processing element executes the operator on the request data, possibly generating multiple request continuations in the process. The processing element enqueues these new request continuations into the LDCs for the operators next in the application graph. The operators can be running on multiple processing elements; the LDC selects the particular processing elements.

We implement the Lagniappe request-execution engine inside of the Lagniappe Library. Lagniappe binds the platform-specific `ProcElement` implementations to the Lagniappe request-execution engine, thus enabling the engine to process requests at run-time.

A `ProcElement` implementation dequeues `RCon` instantiations from one of its several possible incoming `CommChannel` implementations. The `ProcElement` extracts the pointers to the `RData` and the `Operator` and the name of the incoming port. The `ProcElement` passes the name of the port and the `RData` to the `Operator`, where the `Operator` implementation calls the appropriate handler method. Notice that a `ProcElement` blindly executes any `Operator` and `RData` combination it receives from the `RCon`. We design the execution engine like this so that it is portable and simple. The engine is not dependent on any aspect of the application. It is the Adaptation Framework, that we implement as the `Monitor` class, that determines where an `Operator` instantiation can run. Also, there is no dependence on any specific system properties. The model defines that `ProcElement` implementations are associated with incoming CommChannel implementations. How a system programmer implements the execution core of the `ProcElement` or the message-passing functionality of the `CommChannel` has no bearing on the design or implementation of the Lagniappe Execution Engine.

Adaptation Framework

The Lagniappe library implements the adaptation framework in the `Monitor` class. The `Monitor` has an incoming and an outgoing connection to every `Operator` implementation in the application. Status messages are sent to the `Monitor` by `Operator` implementations when they request some type of run-time system intervention.

Instead of designing a separate subsystem for the `Monitor`, we implement the `Monitor` as a subclass of the `Operator` class, and have it run on a `ProcElement` the same as any of the other `Operator` implementations in the system. Lagniappe inserts the `Monitor` into the application graph with all-to-all connectivity. The `Monitor` handles initial resource allocation and makes decisions on how to adapt that allocation based on workload changes that the `ProcElement` implementations observe.

We implement the three major policy decisions of adaptation (how many, when, and where) in the `Monitor` class. When an `ProcElement` instantiation enqueues an `RCon` to the next `Operator` instance, the `Operator` checks to see if that

`RCon` caused its incoming queue to move beyond a threshold value. If the length of the queue grows beyond the threshold, the `Operator` sends a request for more resources to the `Monitor`. Upon receipt of a request for more resources, the `Monitor` checks to see if any resources are available and, if so, uses the resource-assignment mechanism to allocate the resource to the `operator` instance. To decide what to adapt, the `Monitor` tracks resource allocations across all memory and processor groups (it receives the membership information from the system model) and selects an appropriate `ProcElement` instance based on the current allocation of `Operator` objects to resources. The `Monitor` also takes into account the communication costs both in shared-memory and message-passing resources between the original and new allocation when making its decision.

Changes in the resource allocation necessitate persistent state management if Lagniappe replicates `Operator` instances to `ProcElement` instances that are not in the same memory group, i.e., `ProcElement` instances that do not both have access to a coherent shared memory. The resource assignment mechanism maintains efficient access to persistent state by moving state to the most local `Memory` implementation a `ProcElement` instance can access, while ensuring correctness. Lagniappe performs this state movement by employing the application-specific state-maintenance methods the application programmer provides. These methods are black boxes to Lagniappe; the format and composition of application-specific state is not necessary for its migration.

5.2.2 Lagniappe Compiler Implementation

The Lagniappe compiler implements the three transforms we discuss in section 5.1. The Lagniappe compiler generates wrapper classes, inherited from the classes in the Lagniappe library, around the platform-specific resource implementations and the application request-processing implementations. The Lagniappe compiler also generates a profiler that measures the execution time of the application. The compiler, using this profiler data, generates adaptation policies specific to the application running on the particular multi-$\star$ platform. While, ideally, these values would be determined during runtime, our current implementation uses a profiler, and we discuss it here or completeness.

In this section, we discuss the steps necessary to compile an application model, the design and operation of the profiler, the steps necessary to compile a system model, and how Lagniappe ties these all together to meet the requirements. However, first we describe the basic implementation of the compiler.

General Implementation Details

The Lagniappe compiler is built using a derivative of ANTLR (ANother Tool for Language Recognition) [2] called ANTXR (ANother Tool for XML Recognition) [3]. ANTXR allows for the inclusion of XML keywords directly into an ANTLR grammar. We build the logic for code generation and abstract-syntax tree (AST) walking in Java.

The basic operation of the translator is as follows. ANTXR builds an AST using the XML-based grammar from the application and system meta-models. For each entity, we create a Java class that has as private members the properties and relationships we define in the meta-models.

We implement methods within each of the entity Java classes that generate the necessary code relative to the specific properties or relationships that entity maintains. The Lagniappe compiler walks the AST and recursively constructs and collects the objects. Once the compiler collects all the objects, the compiler generates specific code based on the mode in which it was run. The programmer selects the mode by executing the compiler with different command-line flags. The different modes recognize different ASTs and call different code-generation methods.

The *application* mode generates the C++ implementations of the Operator entities in the model. The *system* mode generates the C++ implementations of the various resource entities in the system model. Lagniappe has a *main* and *prof* mode that generate the `main` function for the standard application and the profiler, respectively. Finally, the *final* mode reads in the profile data and generates the implementations with the specific adaptation policies included. We now look at the specific functionality within these translation tasks.

Application Compiler

```
<application name="sample">
  <delay_guarantee>10000</delay_guarantee>
  <operator name="net_op" fileURI="NetRequestDevice.hh"
            requestGenerator="true">
    ...
  </operator>
  <operator name="stateless_op">
    ...
  </operator>
  <operator name="audit_op">
    ...
  </operator>
  <operator name="stateful_op">
    ...
  </operator>
  <connector>
    ...
  </connector>
  ...
</application>
```

Figure 5.6: Lagniappe XML code for the *sample* application.

The application translator generates classes derived from `Operator` as well as an instance of `Application` that is specific to the particular application model. To describe this process, we use an example application entitled *sample*. The XML Lagniappe model code for this application is shown in Figure 5.6. The details of the entities are abbreviated, but we explore them as we move through the process of the application translator.

1. For each Operator entity in the application model, a new class is generated that is derived from `Operator`. In the *sample* application, the compiler generates a new class for each of the *stateless_op*, *audit_op*, and *stateful_op* entities entitled

`stateless_op`, `audit_op`, and `stateful_op`, respectively.

2. For each Port of type *INCOMING*, the associated *Handler* is declared as a private method. For each Port of type *OUTGOING*, a private method is declared with the *Name* of the port. The application programmer uses this method to send requests from the respective port. Figure 5.7 shows the XML description of the *stateless_op* operator. It has one Port entity that is marked as *INCOMING*, *input_port*. The compiler creates a private method in the `stateless_op` class called `validateRequest`. This method has as an argument a pointer to a `NetRequest` object, the data from the incoming `RCon`. The programmer is expected to provide the implementation of this method. For the *OUT* type Port in our example, `output_port`, the compiler creates a private method by that name that creates an `RCon` object from the request data passed to the method. Note, that *stateless_op* has two *OUT* ports; the second one is of a different type, `NetFlowMessage`. The compiler lists the class definition header files in the include section of the header file for the class `stateless_op`.

```
<operator name="stateless_op">
  <port name="input_port">
    <direction>IN</direction>
    <handler>validateRequest</handler>
    <type name="NetRequest" fileURI="NetRequest.hh">
      <core_type>requestTypes::NetRequest</core_type>
    </type>
  </port>
  <port name="output_port">
    <direction>OUT</direction>
    <handler>null</handler>
    <type name="NetRequest" fileURI="NetRequest.hh">
      <core_type>requestTypes::NetRequest</core_type>
    </type>
  </port>
  <port name="control_port">
    <direction>OUT</direction>
    <handler>null</handler>
    <type name="NewFlowMessage" fileURI="NetFlowMessage.hh">
      <core_type>requestTypes::NewFlow</core_type>
    </type>
  </port>
</operator>
```

Figure 5.7: Lagniappe XML code for a stateless operator.

3. If an Operator entity is marked as a request generator, the compiler reads the Port entities associated with this Operator and saves them for later construc-

tion of the application graph. In our example, *net_op* is a request generator. The programmer provides the complete implementation of the `net_op` class according to the interface for a `RequestGenerator` object that the Lagniappe library defines. In our example, *net_op* injects requests of type `NetRequest` into the application and accepts `NetRequest` objects to send somewhere. The assumption is the system programmer provides this special operator implementation so that applications can run properly on his or her multi-$\star$ system.

4. Each derived instance of `Operator` contains a pointer to an instance of the `Mutex` class. The `Mutex` pointer allows application programmers to write thread-safe handlers. The `Mutex` pointer is initialized during the application setup to point to a platform-specific mutual exclusion implementation. The application programmer may treat the `Operator` instance as a monitor; the `Mutex` provides interfaces for locking and unlocking, as well as waiting and signaling condition variables.

```
<operator name="stateful_op">
  <state fileURI="FileRecord.hh">
    <init_method>fileInit</init_method>
    <install_method>fileInstall</install_method>
    <get_method>fileGet</get_method>
    <purge_method>filePurge</purge_method>
    <flow_sig name="SourceID">
      <core_type>requestTypes::NetRequest</core_type>
      <get_flow_ID>getSourceID</get_flow_ID>
    </flow_sig>
    <dataItem name="fileLockMap">
      <dataType>std::map&lt;lagniappe::FlowID,
        state_records::FileRecord_p&gt;</dataType>
    </dataItem>
  </state>
  <port name="inputPort">
    <direction>IN</direction>
    <handler>lookupWebContent</handler>
    <type name="NetRequest" fileURI="NetRequest.hh">
      <core_type>requestTypes::NetRequest</core_type>
    </type>
  </port>
  <port name="outputPort">
    <direction>OUT</direction>
    <handler>null</handler>
    <type name="NetRequest" fileURI="NetRequest.hh">
      <core_type>requestTypes::NetRequest</core_type>
    </type>
  </port>
</operator>
```

Figure 5.8: Lagniappe XML code for a stateful operator.

5. If the `Operator` relates to a `State` entity, the persistent state is declared as

a private member variable of the `Operator`, and the state access methods are declared as private methods. The compiler generates public wrapper methods for these private ones. These public methods always support the same interface; the Lagniappe library can always call the state maintenance methods regardless of the specific implementations, and it knows nothing of the internal implementations of these methods. Figure 5.8 shows the *stateful_op* operator, and its associated State entity. The compiler declares a private member variable `fileLockMap` of type `std::map<lagniappe::flowID, state_records::FileRecord>`. The type `FileRecord` is a user-provided type that describes the data associated with any particular flow identifier. The definition of this file is specified in the model in the property *FileURI* and the compiler includes the file (`FileRecord.hh`) in the generated header file. The programmer provides four state maintenance methods: `fileInit`, `fileInstall`, `fileGet`, and `filePurge`. The compiler declares all four of these methods as private and, as previously described, wraps the latter three in standard interface methods. The compiler writes `fileInit` into the class constructor. This placement allows the programmers to initialize their persistent state at startup.

6. If the State relates to a Flow Signature entity, as our *stateful_op* entity does, the *GetFlowID* property of the Flow Signature entity is used by the `Operator` instance's load distributor to perform flow-based load distribution (i.e. flow-pinning). The compiler creates a private method entitled `getFlowID` that the programmer may use to get the same flow identifier that Lagniappe uses. Also, the compiler creates a map from `flowID` to `ProcElement` named `flowMap`. In our example, the programmer provides a method `getSourceID`. (Note, the relationship to a type.) Lagniappe expects the programmer to implement within the request type definition, not the operator code.

7. The load distribution functionality is different depending on whether the State is associated with a Flow Signature entity. Lagniappe implements the load distribution policy in a virtual method named `getProc`. If there is no state if the state is not flow-indexed, the compiler implements the method as a simple round-robin that cycles through the list of `ProcElement` instances that the monitor has assigned to the `Operator`. However, if there is flow-indexed state, the compiler implements flow-pinning. The `getProc` method takes a pointer to a `RCon` and calls the `getFlowID` method on the properly type-converted `RData`. The method looks up the flow identifier of the `RData` using the provided method then looks up the flow identifier in the `flowMap` data structure. If the corresponding value is empty, it assigns a new `ProcElement` from the list to the flow identifier. Otherwise, the method returns the stored `ProcElement`.

```
<connector>
  <con_reference>net_op.out</con_reference>
  <con_reference>stateless_op.input_port</con_reference>
</connector>
<connector>
  <con_reference>stateless_op.output_port</con_reference>
  <con_reference>stateful_op.input_port</con_reference>
</connector>
<connector>
  <con_reference>stateless_op.control_port</con_reference>
  <con_reference>audit_op.new_flows</con_reference>
</connector>
<connector>
  <con_reference>stateful_op.ouput_port</con_reference>
  <con_reference>net_op.in</con_reference>
</connector>
```

Figure 5.9: Lagniappe XML code for the connections between operators.

8. The compiler creates a class that is inherited from the `Application` class that implements its two major abstract methods: `buildOperators` and `connect-Graph`. `buildOperators` iterates through the list of Operator entities and performs the operations we describe in this list for each Operator. `connectGraph` iterates through the Connection entities and sets the port connectivity for each Operator and Port. If an *OUTGOING* Port is not connected, the application compiler adds a stub that drops all requests leaving that Port. Figure 5.9 shows the connections in the sample application. Requests flow from the *out*

port of *net_op* to the *input_port* of *stateless_op*. Next, data requests of type `NetRequest` leave *stateless_op* and enter the *input_port* of *stateful_op*. However, requests of type `NetRequest` leave the *control_port* of *stateless_op* and enter the *new_flows* port of *audit_op*. Finally, requests of type `NetRequest` leave *data_out* on *stateful_op* and exit the system through the *in* port of *net_op*.

9. The compiler instantiates the `Monitor` and generates code that connects the `Monitor` object to every operator in the application graph, and vice versa. The `Monitor` is responsible for allocating the initial mapping of operators and tracks resource needs and usage. The `Monitor` responds to requests from the `Operator` instances for more resources during runtime when the workload changes.

10. The application compiler creates the main application file that creates an instance of both the generated `Application` class and the `System` class. It calls `createResources`, `buildOperators`, and `connectGraph`. Finally, the main function calls the `schedule` method of the application to schedule the operators to resources.

System Compiler

The platform translator generates classes derived from `ProcElement`, `CommChannel`, and `Memory`. The compiler also generates an instance of `System` specific to the system model. Figure 5.10 shows an example of a Lagniappe system model in XML. We now describe the specific steps of the system compiler:

1. For each Processing Element, Communication Channel, Memory, Timer, and Mutex entity, a new class is defined that is derived from `ProcElement`, `CommChannel`, `Memory`, `Timer`, and `Mutex`, respectively. A private member variable is declared of *core_type*, the type that the platform programmer provides

to implement the platform-specific resource. The classes' respective abstract methods are instantiated. Figure 5.11 shows the Lagniappe models for two Processing Element entities that Lagniappe uses to generate the classes `Proc0` and `Proc1`. Figure 5.12 shows the other entities in the system. The compiler generates message-passing classes `CC0` and `CC1`, a memory resource `Mem0`, and a mutual exclusion implementation `Mutex0`.

```
<system name="proc2sys">
  <pe name="Proc0" fileURI="Thread.hh">
    ...
  </pe>
  <pe name="Proc1" fileURI="Thread.hh">
    ...
  </pe>
  <memory name="Mem0" fileURI="distMem.h">
    ...
  </memory>
  <comm_channel name="CC0" fileURI="memQueue.hh">
    ...
  </comm_channel>
  <comm_channel name="CC1" fileURI="memQueue.hh">
    ...
  </comm_channel>
  <mutex name="Mutex0" fileURI="simpleMutex.h">
    ...
  </mutex>
  <mem_group name="memgrp1">
    ...
  </mem_group>
</system>
```

Figure 5.10: Lagniappe XML code for a sample multi-$\star$ system.

```
<pe name="Proc0" fileURI="Thread.hh">
  <init_driver>init</init_driver>
  <kill_driver>kill</kill_driver>
  <proc_type>IA32</proc_type>
  <core_type>Thread</core_type>
  <timer name="timer1" fileURI="BasicTimer.h">
    <core_type>timer::BasicTimer</core_type>
    <start>start</start>
    <stop>stop</stop>
  </timer>
  <chan_reference>CC0</chan_reference>
  <chan_reference>CC1</chan_reference>
  <mem_reference>Mem0</mem_reference>
  <mtx_reference>Mutex0</mtx_reference>
</pe>
```

Figure 5.11: Lagniappe XML code for a processing element.

2. For the classes derived from `Memory` and `CommChannel`, the *Bandwidth* and *Latency* values are stored as constants within the generated classes. The resource assignment mechanism uses these values to determine the best channel implementation to use for pairwise communication between processing elements

```
<memory name="Mem0" fileURI="distMem.h">
  <core_type>distMem</core_type>
  <latency>30</latency>
  <bandwidth>100</bandwidth>
  <read_driver>Read</read_driver>
  <write_driver>Write</write_driver>
  <proc_reference>Proc0</proc_reference>
  <mtx_reference>Mutex0</mtx_reference>
</memory>
...
<comm_channel name="CC0" fileURI="memQueue.hh">
  <core_type>memQueue</core_type>
  <latency>30</latency>
  <bandwidth>100</bandwidth>
  <put_driver>push</put_driver>
  <get_driver>pop</get_driver>
  <proc_reference>Proc0</proc_reference>
  <mtx_reference>Mutex0</mtx_reference>
</comm_channel>
...
<mutex name="Mutex0" fileURI="simpleMutex.h">
  <core_type>simpleMtx</core_type>
  <lock>lock</lock>
  <unlock>unlock</unlock>
  <wait>wait</wait>
  <notify>notify</notify>
  <notifyAll>notifyAll</notifyAll>
</mutex>
```

Figure 5.12: Lagniappe XML code for a communication channel, a memory resource, and a mutual exclusion resource.

during runtime. As well, the adaptation framework takes these values into account to determine what resources to use during adaptation. Our example resources have equal values, as shown in Figure 5.12.

3. The `Mutex` classes allow the `Memory` and `CommChannel` classes to implement monitor functionality by exporting `lock` and `unlock` methods, along with waiting and signalling on condition variables. Lagniappe initializes an instance of the `Mutex` class that has a relation the `ProcElement` classes for each `Operator` instance, so that the `Operator` instances can have monitor functionality.

4. Lastly, an instance of the `System` class is generated. The abstract method `createResources` is implemented. First, `Memory` and `CommChannel` classes are generated for each model instance. Then, the compiler uses the entity relationships to define the connectivity of the `ProcElement` instances. In our example, the `Proc0` class sends `RCon` objects to `Proc1` using the `CC1` implementation, and vice versa.

5.3 Multi-Machine Lagniappe

For us to expand Lagniappe to run on multiple machines in a coordinated fashion, several design challenges arise. First, Lagniappe must coordinate resource allocation decisions across all of the machines. Second, each machine in the cluster can possibly require access to the resources across all the other machines in the cluster. Finally, resources in the cluster are heterogeneous, for example, communicating between processors in two different machines is obviously more expensive than communicating between processors in the same machine. In this section we discuss how our design for multi-machine Lagniappe meets these challenges. We also discuss briefly the main logistical challenge for a system programmer in providing resource implementations for a multi-machine system: data serialization.

5.3.1 Coordinating Resource Allocation

All resource decisions across the entire multi-machine cluster must be made with the notion that the local machine is not the only machine in the system. This knowledge is required to keep resource decisions coherent. We design Lagniappe to make decisions locally, but broadcast these decisions globally. In more detail, when an operator on one machine (or more accurately, within a processor group) requests more resources, the monitor running on that processor group attempts to fulfill the request with the free processors in the group. If the monitor can, it does, and it alerts the other monitors of this decision.

However, Lagniappe must handle the case where a monitor cannot handle a request for more resources locally. At start-up, one monitor assumes the position of the *master* and monitors on all other systems become *slaves*. The master handles the system-wide decisions that must be made. The programmer selects the master at start-up by issuing a command to the Lagniappe console on one of the machines. Once one monitor is the master, it notifies the other monitors in the system with special monitor requests of its status. The master monitor updates all slave monitors of any non-local resource decisions that it makes.

The master monitor also performs the initial mapping of operator entities to processing element. The master monitor does not fill up a processor group with operators. It instead leaves room for replication inside of the processor group.

5.3.2 System-Wide Resource Access

Given the hop-by-hop nature of the request routing in the Lagniappe system, each processing element resource must be accessible within each processor group. In other words, when a request leaves an operator, the current processing element looks up the next-hop operator and then the processing element where that next-hop operator is running. If the next-hop processing element is on another machine, there is no way to have a pointer to that remote object. To solve this problem we introduce the concept of the *ghost processor*.

A ghost processor exports the interfaces of the `ProcElement` class, but the Lagniappe compiler does not link to it the system-specific processor implementation. Ghost processors are simply placeholders to ensure the routing algorithm works seamlessly, regardless of the number of processor groups in the system. The monitor of a processor group also uses the ghost processors in its list of `Operator` object to `ProcElement` object assignments.

5.3.3 Resource Heterogeneity

Multi-machine Lagniappe guarantees the presence of heterogeneous resource types within the system. For example, the communication cost between machines is guaranteed to be higher than that between processors of the same machine. The master monitor maps the operators of an application to the processing elements of the system with two guiding principles: 1) leave room within the processor groups to facilitate possible replication and 2) change processor groups a minimal number of times during initial mapping to introduce the smallest amount of communication latency possible.

During replication, the monitor replicates an operator within the most local group possible, either the processor's memory group or processor group. Only if there is no more room in the processing groups does the monitor replicate the operator into another group.

5.3.4 Data Serialization

The main challenge that multi-machine Lagniappe presents to the system programmer is to create Communication Channel entity implementations that can serialize the C++ request continuation structures. To address this challenge, we create a `SerialRCon` class that automatically translates the pointer references to `Operator` instantiations to the given names of the `Operator`. While the request data must still be serialized, the constructors to and from the `SerialRCon` class automatically handle the translation from Operator names to `Operator` objects.

5.4 Requirements Discussion

Once a programmer compiles the application model, runs the profiler, and compiles the system model, he or she has all the code necessary for a functioning application that is tuned for the particular multi-$\star$ system. We now address how this new application meets our requirements.

- **Automate mapping**: The mapping of `Operator` instances to `ProcElement` instances happens automatically. The `Monitor` instantiation initially maps the `Operator` instances, and then handles all further request for resources. The application programmer does not write any code to map his or her application entities to any hardware resources.

- **Efficient resource usage**: The Lagniappe Compiler uses the state access properties of each `Operator` entity to define a custom load-distribution policy for the `Operator` instantiation. The `Monitor` performs state maintenance (for when the system replicates the `Operator` across memory or processor groups, i.e. where no hardware-coherent memory exists) using the "blackbox" methods the application programmer provides.

- **Adaptation**: The profiler automatically determines adaptation thresholds (answering the question, "when to adapt?") for the `Operator` instantiations. The user-provided request generation methods allow the profiler to perform an accurate profile of run-times. The compiler answers the question, "How to adapt?" by using the system model to determine the most efficient resource to use during adaptation.

- **Portability**: The separation of the application model from the system model allows the application programmer to focus purely on his or her application code. No knowledge of the configuration of the underlying hardware is necessary. A programmer can thus move a Lagniappe application to any multi-$\star$ platform that has a Lagniappe model.

Chapter 6

Experimental Evaluation

6.1 Application Description

We test the performance and features of Lagniappe using three applications: *naptpt*, *attack*, and *classify*. The *attack* and *classify* applications operate at the application-level of the network stack, and the *naptpt* application operates at the lower packet-level of the network stack [28]. This difference in level helps show Lagniappe's generality across the request processing domain.

6.1.1 The *naptpt* Application

The *naptpt* application is a simple implementation of network address and port translation - protocol translation [43]. *naptpt* provides a gateway between a IPv6 [21] local network and the IPv4 [13] internet. We also include some basic deep packet inspection, as one can imagine an application like this running at the internet connection of a small business or home. Appendix A.1 lists the Lagniappe XML code for the *naptpt* application.

Figure 6.1 shows the application graph of the *naptpt* application. *naptpt* has three operators: *deep-inspect*, *six-to-four*, and *four-to-six*. Network packets flow

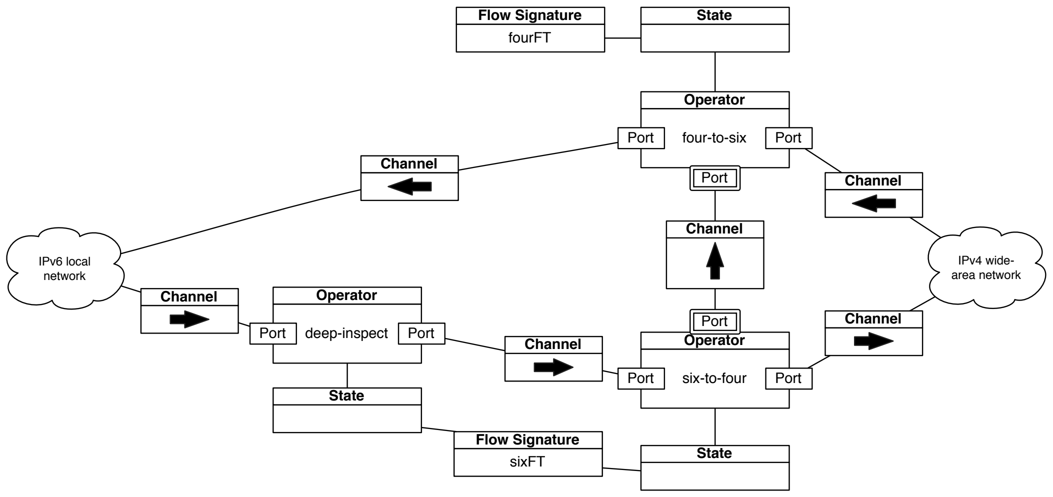

Figure 6.1: The application graph for the *naptpt* application.

through the application in two directions, as shown in Figure 6.1 and *six-to-four* sends messages to *four-to-six* when it encounters a new flow of packets.

Network packets from the IPv6 local network flow into the application and enter the *deep-inspect* operator. The *deep-inspect* operator removes the packet headers and scans the payload for patterns that may indicate the packet contains either signatures of some type of attack or content that the application programmer wants to block. If any of the searches find objectionable content, *deep-inspect* records this information in a record tied to the flow ID of the packet. Flow IDs for the *deep-inspect* operator are determined by a packet's IP source and destination address, TCP source and destination port, and the protocol identifier from the IP header. These five fields of information are commonly referred to as the packet's five-tuple. If the flow has had more than some threshold of infractions, the packet is (and all future packets from that flow are) destroyed. However, if the packet passes all the inspections, *deep-inspect* sends the packet out its only *OUTGOING* port.

The *six-to-four* operator takes TCP/IPv6 packets and translates them to a valid IPv4/TCP packet. The operator sets the IPv4 source address of the server's

IPv4 network interface as the IPv4 as the IPv4 source address in the new packet. The destination in our small application is always the same; however, there are accepted ways to embed IPv4 addresses in IPv6 addresses in the literature [17]. The TCP destination port remains the same; however, the operator determines the TCP source port based on the flow ID of the IPv6 packet. If the operator has seen this flow before, it simply uses the source port in the saved flow state. However, if the flow is new, the operator creates a new port number specifically for this flow and saves the port to the corresponding flow state. The operator packages the IPv6 five-tuple and new IPv4 five-tuple in a control message and sends it to the *four-to-six* operator for processing. *six-to-four* calculates checksums for both the IP layer and the TCP layer. Finally, *six-to-four* writes the appropriate ethernet MAC addresses to the packet data. The operator then sends the packet out the appropriate *OUTGOING* port.

The *four-to-six* operator first looks up flow state for an incoming IPv4 packet according to the packet's five-tuple. If the flow-state does not exist, the operator drops the packet as this packet is from an unsolicited flow. If the flow state exists, the operator creates a new IPv6 header and sets the source and destination addresses from the flow state. The operator also sets the TCP source and destination ports from the flow state. *four-to-six* then sets the TCP checksum and sends the packet out its only *OUTGOING* port.

When the *four-to-six* operator receives a new control message from the *six-to-four* operator, it initializes a new flow state record with the two five-tuples (IPv4 and IPv6) in the message. The operator now accepts any incoming IPv4 packets that have the corresponding five-tuple flow ID.

6.1.2 The *attack* Application

The *attack* application implements a simple webserver with some increased security-oriented functionality. The application performs a two-level inspection to identify intrusions and attacks efficiently. Also, we model the computation necessary to encrypt and sign the file data being fetched and sent to the remote client. Appendix A.2 lists the Lagniappe XML code for the *attack* application.

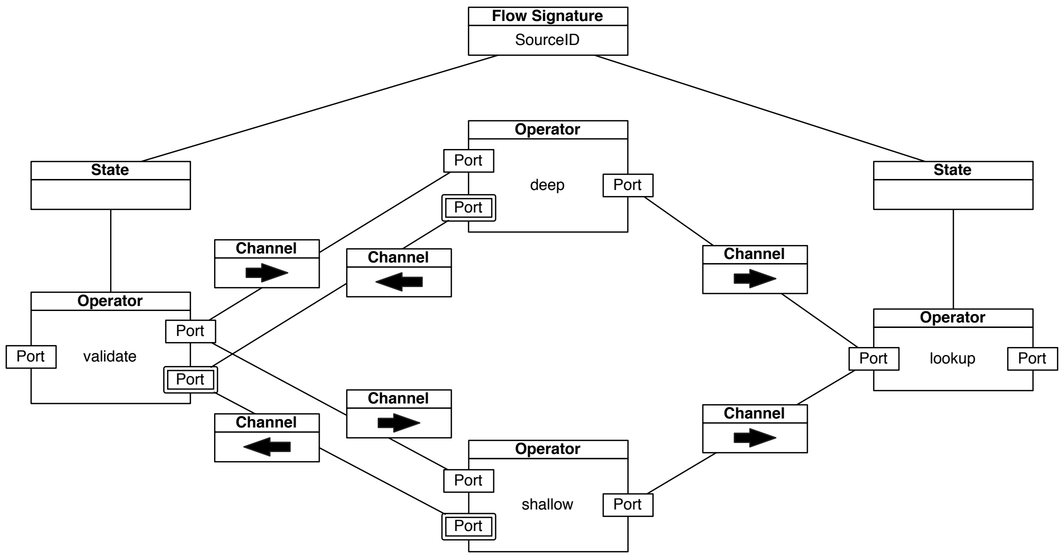

Figure 6.2: The application graph for the *attack* application.

Figure 6.2 shows the application graph for the *attack* application. The application comprises four operators: *validate*, *shallow*, *deep*, and *lookup*. The dotted lines signify control messages that *deep* and *shallow* send back to *validate* when their respective tests find positive signs of intrusion or attack. Requests that are not dropped for being malicious move to the *lookup* operator where the operator reads the appropriate file, does some security processing, and serves the file to the client.

The *validate* operator has two major functions: validate the http request and check to see if the flow has been marked as suspicious or dangerous. First, the operator parses the http request string and extracts the file name. If the request is not valid, *validate* drops the request. If the request is valid, the operator looks up the flow according to a combination of source address, source port, and file name. A flow can be in three possible states: *INNOCENT*, *SUSPICIOUS*, and *GUILTY*. If a flow is marked as *INNOCENT*, the operator forms a new request with the file name, source address, and source port and sends it out the data port for innocent requests. If a flow is marked as *SUSPICIOUS*, the operator similarly creates a new request and sends it out the data port for suspicious packets. Finally, if the flow is marked as *GUILTY*, the operator drops the request.

The *validate* operator's control message handler simply updates flow state with the appropriate values depending on the message. If the operator receives a message from *shallow* marking a flow as *SUSPICIOUS*, *validate* updates the state appropriately. Similarly, *validate* may receive messages from the *deep* operator marking a flow as either *INNOCENT* or *GUILTY*, and the *validate* operator marks the flow state appropriately.

The *shallow* operator performs a simple length check on the file name of an incoming request. If the check fails, the operator sends a control message back to the *validate* operator marking the flow as *SUSPICIOUS*. Regardless, the *shallow* operator forwards the request on to the *lookup* operator.

The *deep* operator performs a suite of checks looking for malicious flows based on the content of the requests. If it does not find anything, the operator reports back to the *validate* operator with an *INNOCENT* control message. If the tests fail, the *deep* operator reports back to the *validate* operator with a GUILTY control message.

The *validate* operator starts by looking up the flow information about an incoming request and getting a lock on the flow state. The operator then reads the requested file from the local file system and puts this data into a new request. The operator then uses the read file data to model security processing (signatures). We add this step to model a login or secure web application. After the signatures are updated, the operator releases the lock and sends the newly formed request out its data port.

6.1.3 The *classify* Application

The *classify* application implements a webserver that classifies and distributes incoming requests to different operators for file lookup based on the contents of the request. The main difference between this application and *attack* is that two of the operators in *classify* are of equal weight in terms CPU utilization, and thus can showcase Lagniappe's handling of changes in workload composition during runtime. Appendix A.3 lists the Lagniappe XML code for the *classify* application.

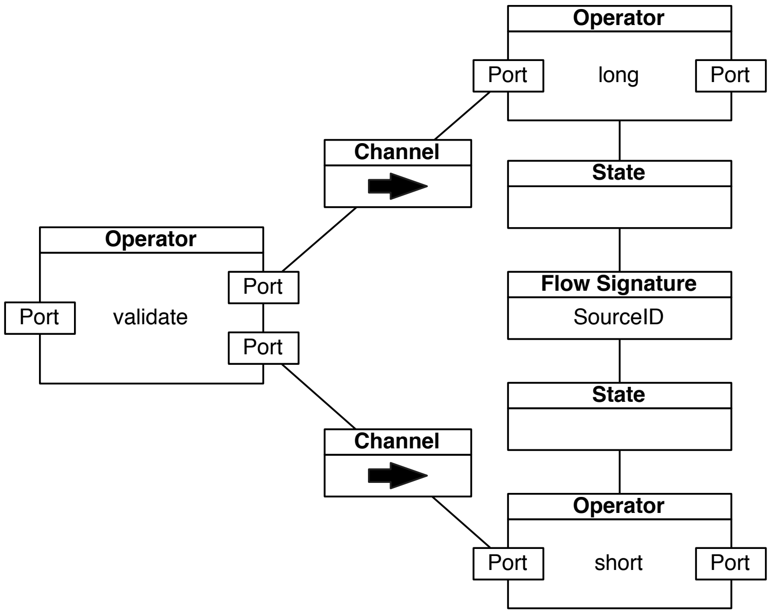

Figure 6.3: The application graph for the *classify* application.

Figure 6.3 shows the application graph for the *classify* application. The application has three operators: *validate*, *short*, and *long*. The *validate* operator takes incoming http requests, validates their format, and extracts the requested file name. Shorter length file names are sent out the *short_requests* output port and longer length file names are sent out the *long_requests* output port.

The *short* and *long* operators are actually identical in composition. They each have one input port and one output port. The operators extract the file names from the incoming request, perform the file read and a simple signature generation, and then send the read file data out the *OUTGOING* port. The handler locks the operator during the file read and signature generation as in the *attack* application. Again, we use this application to model multiple operators with similar run-times.

6.2 Experimental Setup

We have two 1.6 GHz Pentium Xeon four-way, quad-core servers that serve as our primary multi-$\star$ cluster. These servers provide sixteen cores each. We equip each server with Intel Pro dual-port, gigabit ethernet interfaces. One machine is equipped with 128 GBs of memory and the other is equipped with 16 GBs of memory. We use the larger capacity machine to run our Lagniappe applications and the lower capacity machine generates workload.

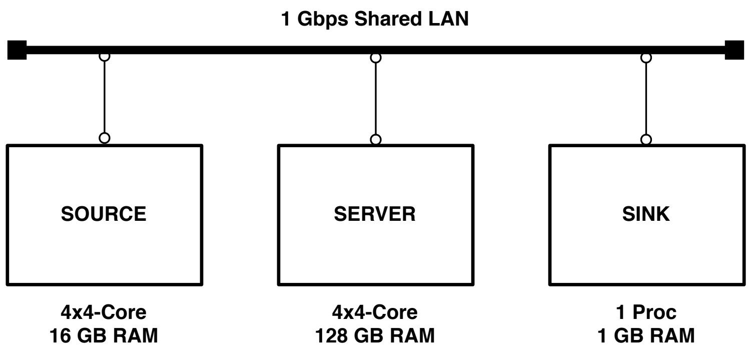

Figure 6.4: Physical setup of the experimental testbed for the *naptpt* experiments.

Figure 6.4 shows the connectivity between the machines for the *naptpt* experiments. For the *naptpt* experiments, we also use a single-processor 3.00 GHz Pentium Xeon server as a sink for our TCP streams. We refer to the low-capacity, four-way machine as SOURCE, the high-capacity, four-way machine as SERVER, and the single-processor machine as SINK. All three machines are connected to each other across a gigabit LAN.

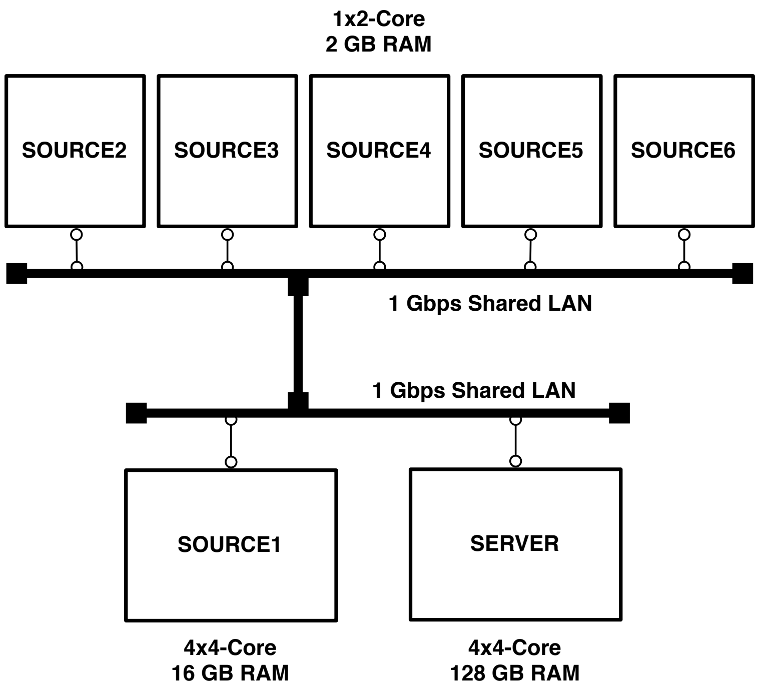

Figure 6.5: Physical setup of the experimental testbed for the *attack* experiments.

Figure 6.5 shows the connectivity between the machines for the *attack* experiments. SOURCE1 and SERVER are connected as in the *naptpt* case. However, a load on SERVER larger than fifty simultaneous clients overwhelms the machine, so we use SERVER2 through SERVER6 to step up the load with ten simultaneous clients running on each between fifty total clients and one-hundred total clients.

The experimental setup for the *classify* application is identical to that of the *attack* application.

For both experiments, we run varying numbers of clients on Source. For the *attack* application, a client requests a series of webpages at approximately 1.7 kilo-requests per second. For the *classify* application, half of the clients request one particular webpage and the other half request the another. For the *naptpt* application, each client is a simple TCP stream that attempts to send data at 10 Mb per second. Notice the difference in units between the two applications. In the *attack* application, the files the application serves are small (156 Bytes). Larger file sizes make the bandwidth of the network connection the limiting factor in performance. By using small file sizes we can greater exercise the Server machine and Lagniappe by determining how many aggregate requests could be processed per unit time. We use Mb/s in the *naptpt* application because the application never approaches the network connections total bandwidth and because we use TCP requests of uniform length.

6.3 Experimental Results

We show three sets of results that show the benefits and features of the Lagniappe Programming Environment. The first set of results shows how without any changes to an application, Lagniappe takes advantage of increased resources to deliver higher performance. The second set of results shows how Lagniappe can adapt the resource usage of an application to adjust to changes in workload. The final set of results shows the benefit of Lagniappe's use of state-access semantics when determining a load-distribution policy for an application's components.

6.3.1 Applications Automatically Use Resources

Figure 6.6 shows the throughput in Mb/s on the y-axis as we increase load on the x-axis. The clients are running on the Source machine and the Sink machine is running a simple TCP receiver. The Server machine sits in the middle running

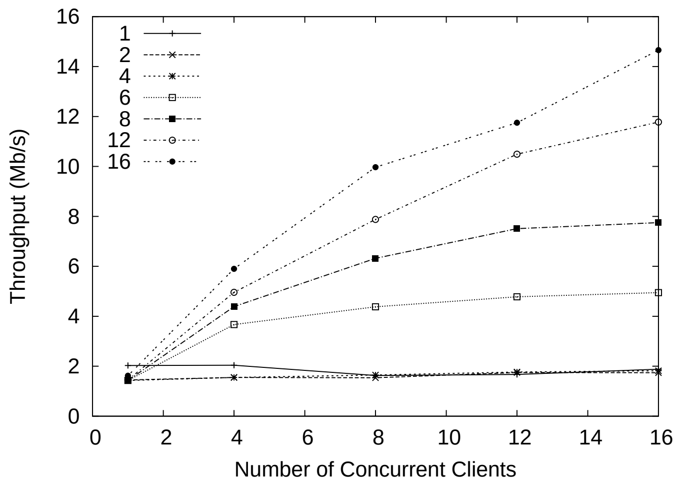

Figure 6.6: Throughput for naptpt for different processor configurations and different workloads.

naptpt. As we increase the number of processors available in the system, the total throughput of *naptpt* increases. The only exceptions to this are in the cases of one, two, and four processors. In these configurations, there are more operators than processors, and thus, we are not able to take advantage of the extra parallelism through replication. The increased cost of using multiple processors (for example, going outside of cache to pass requests) actually causes the throughput to drop as we increase up to four processors under low loads. However, after the number of processors is greater than the number of operators, the gains from replication greatly outnumber the added communication costs.

Figure 6.7 shows a similar situation for the *attack* application, except that here the y-axis represents throughput in Kilorequests/s. Once the number of processors increases beyond the number of operators, the deliverable throughput of the *attack* application increases.

When we move from 12 processors to 16 processors in the *attack* application,

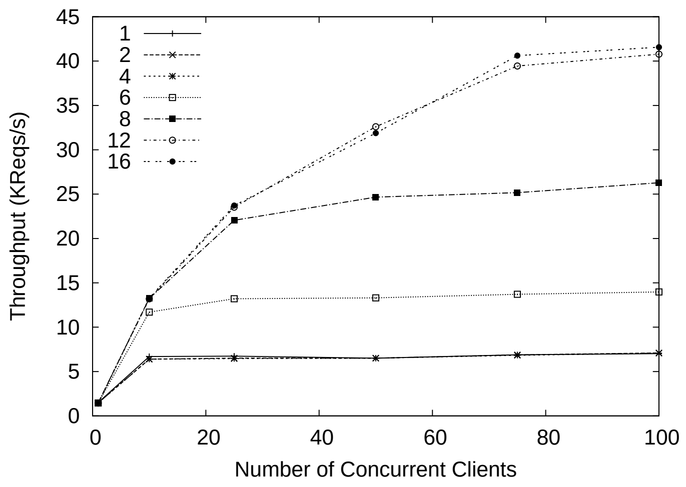

Figure 6.7: Throughput for attack for different processor configurations and different workloads.

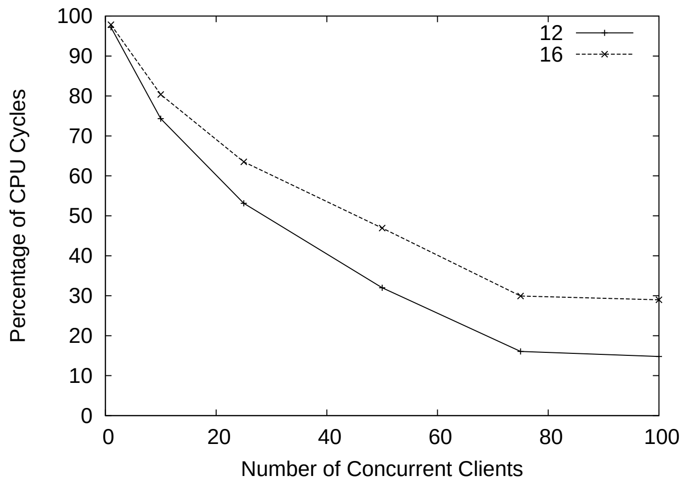

Figure 6.8: Comparison of average processor idle time for the *attack* application between the twelve and sixteen processor cases.

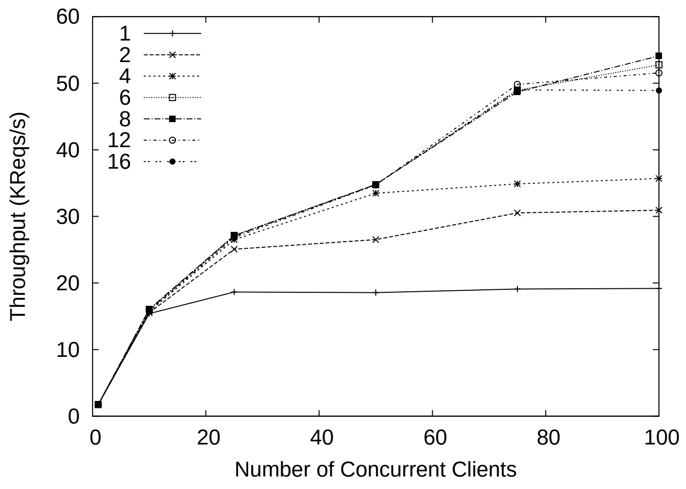

Figure 6.9: Throughput for *classify* for different processor configurations and different workloads.

we do not see a large increase in throughput. When we add four more processors, the utilization of the processors drops, as Figure 6.8 shows.

Figure 6.9 shows a different throughput curve than the other two applications present. The *classify* application does not have one operator that is much heavier than the rest in terms of computation. In fact, all the operators are relatively lightweight. Thus, *classify* maximizes its throughput at a much lower number of processors than the other two applications. More processors just adds more overhead with overall gain in throughput. Note, in contrast to the *naptpt* and *attack* applications, throughput increases with four processors. This difference is because *classify* only has three operators, and the workload is split evenly between them. Thus, once *short* and *long* have their own processor, they can increase their throughput.

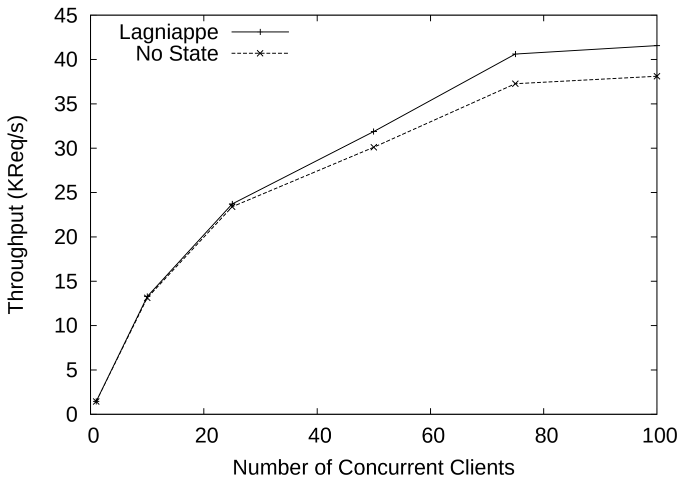

Figure 6.10: Throughput comparison to not using state information for load distribution.

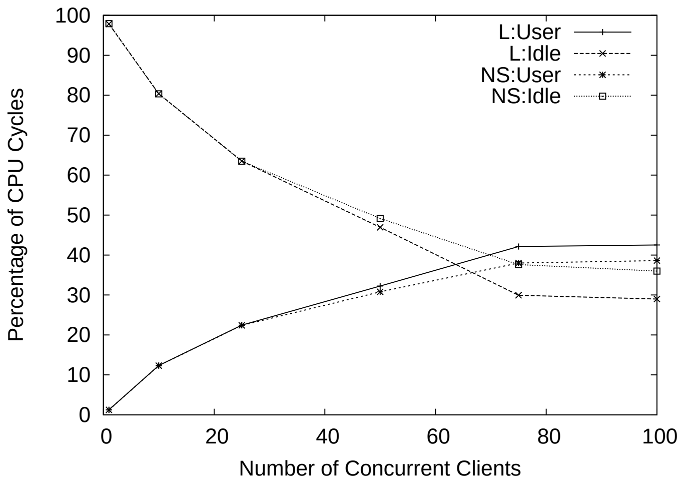

Figure 6.11: Time spent processing and idle for both Lagniappe and a system that does not use state information for load distribution.

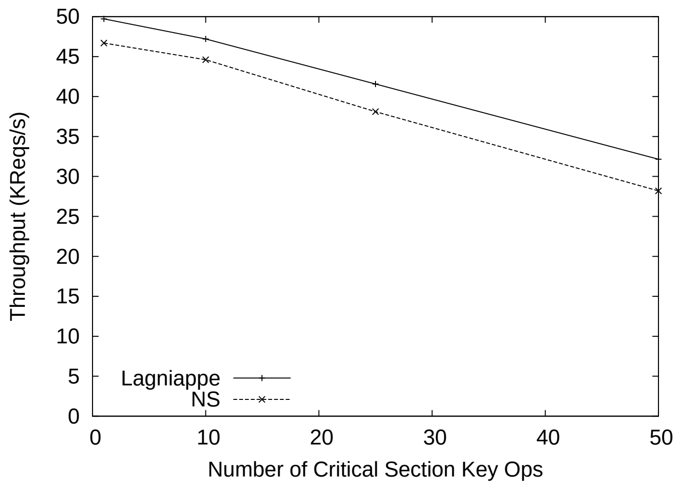

Figure 6.12: Throughput of Lagniappe and the NoState system as the size of the critical section changes for the *attack* application.

6.3.2 Lagniappe Reduces Contention

Figure 6.10 shows the throughput from the *attack* application shown in the 16-processor case in Figure 6.7 compared to the throughput of the same setup, but with no use of the state-access semantics by Lagniappe. We refer to this setup as the NoState system. The y-axis is throughput in Kilorequests/s and the x-axis is number of clients. We see that the NoState case performs worse than the standard Lagniappe case by 8.3%.

This result is to be expected; by abandoning flow-based request distribution, the NoState case increases contention and thus processors spend more time idle and not processing requests. We show this drop in processor utility in Figure 6.11. When a processor attempts to acquire a lock that is active, the processor is put to sleep and the operating system counts the cycles as idle. For the Lagniappe case, we see more time spent active in user mode (i.e. time spent actually processing requests)

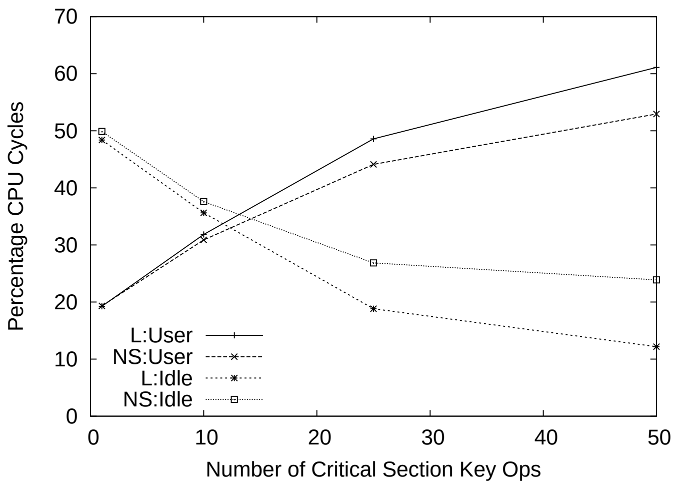

Figure 6.13: User and idle CPU utilization of the Lagniappe and NoState systems for the *attack* application as the size of the critical section changes.

and less time spent idle (i.e. time spent waiting on locks) than in the NoState case.

We now explore the range of difference between the performance of Lagniappe and the NoState case with differing critical section size. In Figure 6.12 we see the throughput of the Lagniappe system and the NoState system as we increase the number of signature-generation steps on the x-axis. We see the difference climb to 12.3% with the large 50-operation critical section. Even with just one signature generation, we see an improvement of 6.1% from using the state information to make load-distribution decisions (note, Lagniappe performs this task automatically).

Figure 6.13 explains the difference in throughput. In this graph the x-axis is the same as before, but the y-axis is percentage of CPU cycle usage. We see the difference between both User and Idle time increase, however, the massive increase in Idle time, almost 90% for the largest critical section, shows the amount of time spent waiting on locks increasing as the critical section increases. Lagniappe can automatically generate a load-distribution policy that takes the state access semantics

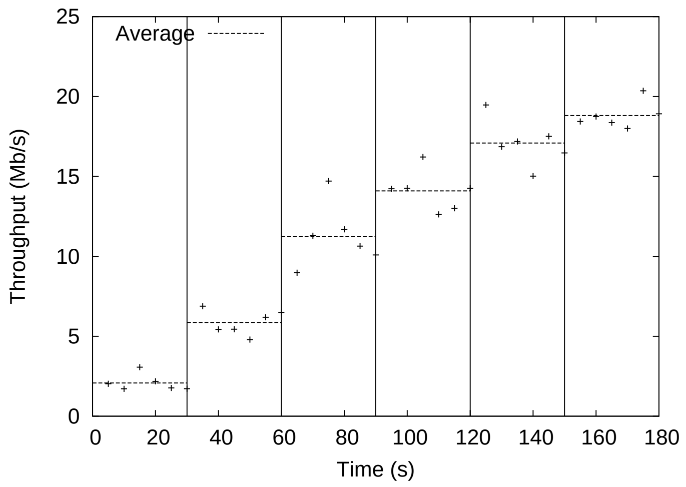

Figure 6.14: Throughput for naptpt over time as load increases to show adaption of resource usage with increasing load.

into account and pins flows to processing elements, providing the types of benefits we see over an state-agnostic system in these graphs.

6.3.3 Lagniappe Adapts to Changes in Workload

Figure 6.14 shows the throughput averages over five-second intervals that we sample over the course of a three minute-period. The y-axis represents throughput in Mb/s and the x-axis is time. Every thirty seconds (represented by the vertical lines) we increase the load. The horizontal lines represent the average throughput during the entire thirty second period. We see that Lagniappe allows the *naptpt* application to add more resources as they are needed and allows the average throughput that the application produces to increase.

Figure 6.15 shows a similar graph for the *attack* application. In this case, we again measure throughput on the y-axis in terms of Kilorequests/s. Figure 6.16

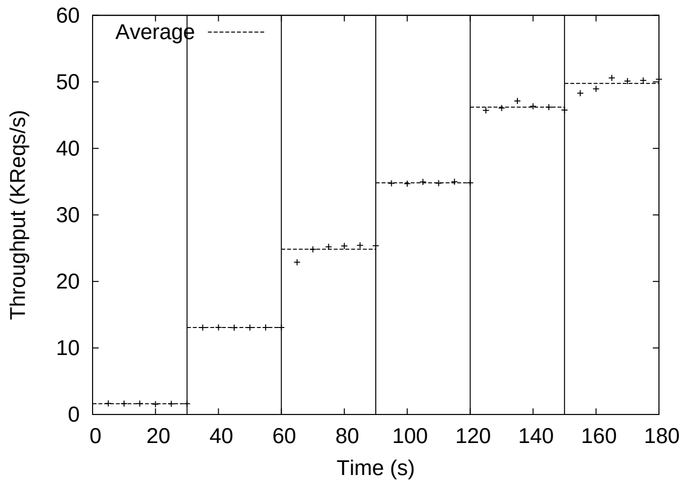

Figure 6.15: Throughput for attack over time as load increases to show adaptation of resource usage with increasing load.

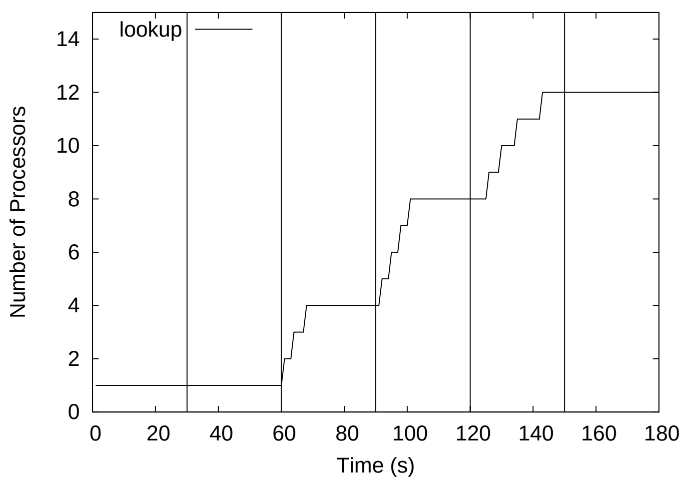

Figure 6.16: Number of processors assigned to the *lookup* operator during the experiment.

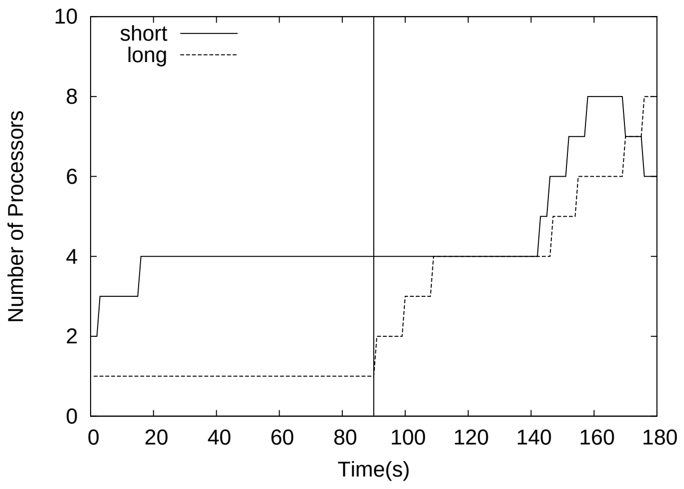

Figure 6.17: The number of processors assigned to operators in the *classify* application as the workload changes over time.

shows the number of processors that Lagniappe assigns to the *lookup* operator. We see as the workload increases Lagniappe provides more resources to the heavy-weight operator.

In Figure 6.17 we show the how the *classify* application deals with changes, not in workload volume as in the previous two experiments, but in workload composition. On the y-axis, we show the number of processors assigned to an operator. On the x-axis, we show time in seconds. In this experiment we start with twelve clients exercising the *short* operator. At the halfway mark (90 seconds) we add another twelve clients; however, these are generating workload that exercises the *long* operator. We see that the adaptation framework assigns the *long* operator processing resources. The *long* operator does eventually pass *short*, but they reach a relatively steady state.

Chapter 7

Conclusion

In this dissertation we describe the Lagniappe Programming Environment. Lagniappe allows programmers to design and implement high-throughput request-processing applications that automatically take advantage of multi-$\star$ resources.

Lagniappe makes the following four key contributions: First, Lagniappe defines and uses a unique *hybrid programming model* that separates the concerns of writing applications for uni-processor, single-threaded execution platforms (single-$\star$ systems) from the concerns of writing applications necessary to efficiently execute on a multi-$\star$ system. We provide separate tools to the programmer to address each set of concerns. Second, we present *meta-models of applications and multi-$\star$ systems* that identify the necessary entities for reasoning about the application domain and multi-$\star$ platforms. Third, we design and implement a platform-independent mechanism called the *load-distributing channel* that factors out the key functionality required for moving an application from a single-$\star$ architecture to a multi-$\star$ one. Finally, we implement a platform-independent *adaptation framework* that defines custom adaptation policies from application and system characteristics to change resource allocations with changes in workload. Furthermore, applications written in the Lagniappe programming environment are *portable*; we further separate the concerns of application programming from system programming in the programming model.

We implement Lagniappe on a cluster of servers each with multiple multicore processors. We demonstrate the generality of Lagniappe by implementing several stateful request-processing applications. We show that our implementation achieves three goals: 1) Lagniappe applications use all resources in a multi-$\star$ system without any explicit resource knowledge; 2) Lagniappe increases throughput by guaranteeing efficient state access through custom policy generation based on the statefulness of operators; and 3) Lagniappe automatically adapts to changes in both workload volume and composition.

Appendix A

Lagniappe Source Code for Applications

A.1 The *naptpt* Application

```
<application name="naptpt">
  <delay_guarantee>1000</delay_guarantee>
  <operator name="eth0" fileURI="PhysInterface.hh" requestGenerator="true">
    <port name="in">
      <direction>IN</direction>
      <handler>null</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <port name="out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <core_type>requestDevices::PhysInterface</core_type>
  </operator>
  <operator name="eth1" fileURI="PhysInterfaceSix.hh" requestGenerator="true">
    <port name="in">
      <direction>IN</direction>
      <handler>null</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <port name="out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <core_type>requestDevices::PhysInterfaceSix</core_type>
  </operator>
  <operator name="four_to_six_op">
    <state fileURI="SixMap.hh">
      <init_method>my_init</init_method>
      <install_method>my_install</install_method>
      <get_method>my_get</get_method>
      <purge_method>my_purge</purge_method>
      <flow_sig name="fivetuple">
        <core_type>pktTypes::Packet</core_type>
        <get_flow_ID>getUDPFiveTupleID</get_flow_ID>
      </flow_sig>
      <dataItem name="flowMapping">
        <dataType>std::map&lt;lagniappe::FlowID, SixMap_p&gt;</dataType>
      </dataItem>
      <dataItem name="dstAddr">
        <dataType>in6_addr</dataType>
      </dataItem>
    </state>
    <port name="data_in">
      <direction>IN</direction>
      <handler>handleMyPort</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <port name="data_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <port name="new_connection_in">
      <direction>IN</direction>
      <handler>newConnection</handler>
      <type name="ConnInfo" fileURI="ConnInfo.hh">
        <core_type>requestTypes::ConnInfo</core_type>
      </type>
    </port>
  </operator>
  <operator name="six_to_four_op">
    <state fileURI="FourMap.hh">
      <init_method>my_init</init_method>
      <install_method>my_install</install_method>
      <get_method>my_get</get_method>
      <purge_method>my_purge</purge_method>
      <flow_sig name="sixfivetuple">
        <core_type>pktTypes::Packet</core_type>
        <get_flow_ID>getUDPSixFiveTupleID</get_flow_ID>
      </flow_sig>
      <dataItem name="flowMapping">
        <dataType>std::map&lt;lagniappe::FlowID, FourMap_p&gt;</dataType>
      </dataItem>
      <dataItem name="currPort">
        <dataType>uint32_t</dataType>
      </dataItem>
      <dataItem name="srcAddr">
        <dataType>uint32_t</dataType>
      </dataItem>
      <dataItem name="dstAddr">
        <dataType>uint32_t</dataType>
      </dataItem>
    </state>
    <port name="data_in">
      <direction>IN</direction>
      <handler>handleMyPort</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <port name="data_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <port name="new_connection_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="ConnInfo" fileURI="ConnInfo.hh">
        <core_type>requestTypes::ConnInfo</core_type>
      </type>
    </port>
  </operator>
  <operator name="deep_inspect_op">
    <state fileURI="InspectRecord.hh">
      <init_method>my_init</init_method>
      <install_method>my_install</install_method>
      <get_method>my_get</get_method>
      <purge_method>my_purge</purge_method>
      <flow_sig name="sixfivetuple">
        <core_type>pktTypes::Packet</core_type>
        <get_flow_ID>getUDPSixFiveTupleID</get_flow_ID>
      </flow_sig>
      <dataItem name="flowMapping">
```

```
        <dataType>std::map&lt;lagniappe::FlowID, InspectRecord_p&gt;</dataType>
      </dataItem>
    </state>
    <port name="data_in">
      <direction>IN</direction>
      <handler>handleMyPort</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
    <port name="data_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="Packet" fileURI="Packet.hh">
        <core_type>pktTypes::Packet</core_type>
      </type>
    </port>
  </operator>
  <connector>
    <con_reference>eth1.out</con_reference>
    <con_reference>deep_inspect_op.data_in</con_reference>
  </connector>
  <connector>
    <con_reference>deep_inspect_op.data_out</con_reference>
    <con_reference>six_to_four_op.data_in</con_reference>
  </connector>
  <connector>
    <con_reference>six_to_four_op.data_out</con_reference>
    <con_reference>eth0.in</con_reference>
  </connector>
  <connector>
    <con_reference>eth0.out</con_reference>
    <con_reference>four_to_six_op.data_in</con_reference>
  </connector>
  <connector>
    <con_reference>four_to_six_op.data_out</con_reference>
    <con_reference>eth1.in</con_reference>
  </connector>
  <connector>
    <con_reference>six_to_four_op.new_connection_out</con_reference>
    <con_reference>four_to_six_op.new_connection_in</con_reference>
  </connector>
</application>
```

A.2 The *attack* Application

```
<application name="attack">
  <delay_guarantee>100000</delay_guarantee>
  <operator name="net1" fileURI="NetRequestDevice.hh" requestGenerator="true">
    <port name="in">
      <direction>IN</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <core_type>requestDevices::NetRequestDevice</core_type>
  </operator>
  <operator name="validate_op">
    <state fileURI="FlowRecord.hh">
      <init_method>my_init</init_method>
      <install_method>my_install</install_method>
      <get_method>my_get</get_method>
      <purge_method>my_purge</purge_method>
      <flow_sig name="fivetuple">
        <core_type>requestTypes::NetRequest</core_type>
        <get_flow_ID>getSourceID</get_flow_ID>
      </flow_sig>
      <dataItem name="flowStateMap">
        <dataType>std::map&lt;lagniappe::FlowID, state_records::FlowRecord_p&gt;</dataType>
      </dataItem>
    </state>
    <port name="in">
      <direction>IN</direction>
      <handler>validateRequest</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="control">
      <direction>IN</direction>
      <handler>newControlMessage</handler>
      <type name="ControlMessage" fileURI="ControlMessage.hh">
        <core_type>requestTypes::ControlMessage</core_type>
      </type>
    </port>
    <port name="valid_requests">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="suspicious_requests">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
```

```
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
  </operator>
  <operator name="shallow_op">
    <port name="data_in">
      <direction>IN</direction>
      <handler>shallowInspect</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="control_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="ControlMessage" fileURI="ControlMessage.hh">
        <core_type>requestTypes::ControlMessage</core_type>
      </type>
    </port>
    <port name="data_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
  </operator>
  <operator name="deep_op">
    <port name="data_in">
      <direction>IN</direction>
      <handler>deepInspect</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="control_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="ControlMessage" fileURI="ControlMessage.hh">
        <core_type>requestTypes::ControlMessage</core_type>
      </type>
    </port>
    <port name="data_out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
  </operator>
  <operator name="lookup_op">
    <state fileURI="FileRecord.hh">
      <init_method>my_init</init_method>
      <install_method>my_install</install_method>
      <get_method>my_get</get_method>
      <purge_method>my_purge</purge_method>
      <flow_sig name="fivetuple">
        <core_type>requestTypes::NetRequest</core_type>
        <get_flow_ID>getSourceID</get_flow_ID>
      </flow_sig>
      <dataItem name="fileLockMap">
        <dataType>std::map&lt;lagniappe::FlowID, state_records::FileRecord_p&gt;</dataType>
      </dataItem>
    </state>
    <port name="in">
      <direction>IN</direction>
      <handler>lookupWebContent</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
  </operator>
  <connector>
    <con_reference>net1.out</con_reference>
    <con_reference>validate_op.in</con_reference>
  </connector>
  <connector>
    <con_reference>validate_op.valid_requests</con_reference>
    <con_reference>shallow_op.data_in</con_reference>
  </connector>
  <connector>
    <con_reference>validate_op.suspicious_requests</con_reference>
    <con_reference>deep_op.data_in</con_reference>
  </connector>
  <connector>
    <con_reference>shallow_op.data_out</con_reference>
    <con_reference>lookup_op.in</con_reference>
  </connector>
  <connector>
    <con_reference>deep_op.data_out</con_reference>
    <con_reference>lookup_op.in</con_reference>
  </connector>
  <connector>
    <con_reference>shallow_op.control_out</con_reference>
    <con_reference>validate_op.control</con_reference>
  </connector>
  <connector>
    <con_reference>deep_op.control_out</con_reference>
    <con_reference>validate_op.control</con_reference>
  </connector>
  <connector>
    <con_reference>lookup_op.out</con_reference>
    <con_reference>net1.in</con_reference>
  </connector>
</application>
```

A.3 The *classify* Application

```
<application name="classify">
  <delay_guarantee>10000</delay_guarantee>
  <operator name="net1" fileURI="NetRequestDevice.hh" requestGenerator="true">
    <port name="in">
      <direction>IN</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="out">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <core_type>requestDevices::NetRequestDevice</core_type>
  </operator>
  <operator name="validate_op">
    <port name="in">
      <direction>IN</direction>
      <handler>validateRequest</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="short_requests">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
    <port name="long_requests">
      <direction>OUT</direction>
      <handler>null</handler>
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
  </operator>
```

```
<operator name="short_lookup_op">
  <state fileURI="FileRecord.hh">
    <init_method>my_init</init_method>
    <install_method>my_install</install_method>
    <get_method>my_get</get_method>
    <purge_method>my_purge</purge_method>
    <flow_sig name="fivetuple">
      <core_type>requestTypes::NetRequest</core_type>
      <get_flow_ID>getSourceID</get_flow_ID>
    </flow_sig>
    <dataItem name="fileLockMap">
      <dataType>std::map&lt;lagniappe::FlowID, state_records::FileRecord_p&gt;</dataType>
    </dataItem>
  </state>
  <port name="in">
    <direction>IN</direction>
    <handler>lookupWebContent</handler>
    <type name="NetRequest" fileURI="NetRequest.hh">
      <core_type>requestTypes::NetRequest</core_type>
    </type>
  </port>
  <port name="out">
    <direction>OUT</direction>
    <handler>null</handler>
    <type name="NetRequest" fileURI="NetRequest.hh">
      <core_type>requestTypes::NetRequest</core_type>
    </type>
  </port>
</operator>
<operator name="long_lookup_op">
  <state fileURI="FileRecord.hh">
    <init_method>my_init</init_method>
    <install_method>my_install</install_method>
    <get_method>my_get</get_method>
    <purge_method>my_purge</purge_method>
    <flow_sig name="fivetuple">
      <core_type>requestTypes::NetRequest</core_type>
      <get_flow_ID>getSourceID</get_flow_ID>
    </flow_sig>
    <dataItem name="fileLockMap">
      <dataType>std::map&lt;lagniappe::FlowID, state_records::FileRecord_p&gt;</dataType>
    </dataItem>
  </state>
  <port name="in">
    <direction>IN</direction>
    <handler>lookupWebContent</handler>
    <type name="NetRequest" fileURI="NetRequest.hh">
      <core_type>requestTypes::NetRequest</core_type>
    </type>
  </port>
  <port name="out">
    <direction>OUT</direction>
    <handler>null</handler>
```

```
      <type name="NetRequest" fileURI="NetRequest.hh">
        <core_type>requestTypes::NetRequest</core_type>
      </type>
    </port>
  </operator>
  <connector>
    <con_reference>net1.out</con_reference>
    <con_reference>validate_op.in</con_reference>
  </connector>
  <connector>
    <con_reference>validate_op.short_requests</con_reference>
    <con_reference>short_lookup_op.in</con_reference>
  </connector>
  <connector>
    <con_reference>validate_op.long_requests</con_reference>
    <con_reference>long_lookup_op.in</con_reference>
  </connector>
  <connector>
    <con_reference>short_lookup_op.out</con_reference>
    <con_reference>net1.in</con_reference>
  </connector>
  <connector>
    <con_reference>long_lookup_op.out</con_reference>
    <con_reference>net1.in</con_reference>
  </connector>
</application>
```

Bibliography

[1] AMD Quad-Core Opteron. http://www.amd.com/opteron/.

[2] ANTLR, ANother Tool for Language Recognition. http://www.antlr.org.

[3] ANTXR, ANother Tool for XML Recognition. http://javadude.com/tools/antxr/index.html.

[4] Apple Snow Leopard. http://www.apple.com/macosx/snowleopard/.

[5] Intel IXP Family of Network Processors. http://www.intel.com/ design/network/products/npfamily/.

[6] Intel Multi-Core: An Overview. http://www.intel.com/multicore/overview.htm.

[7] OpenCL: What You Need to Know. http://www.macworld.com/article/134858/2008/08/snowleopar

[8] Snort: The Open Source Network Intrusion Detection System. http://www.snort.org/.

[9] Sun UltraSPARC T2 Processor - Overview. http://www.sun.com/ processors/UltraSPARC-T2/.

[10] VMWare. http://www.vmware.com/.

[11] VMotion. http://www.vmware.com/products/vi/vc/vmotion.html.

[12] XenSource. http://www.citrixxenserver.com/Pages/default.aspx.

[13] Internet Protocol. IETF RFC 791, September 1981.

[14] J. Andrews and N. Baker. XBox 360 System Architecture. In *IEEE MICRO,* 26(2), 2006.

[15] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. University of California at Berkeley, Technical Report UCB/EECS-2006-183, December 2006.

[16] Brendan Burns, Kevin Grimaldi, Alex Kostadinov, Emery Berger, and Mark Corner. Flux: A Language for Programming High-Performance Servers. In *Proceedings of the USENIX Annual Technical Conference,* 2006.

[17] B. Carpenter and K. Moore. Connection of IPv6 Domains Via IPv4 Clouds. http://tools.ietf.org/html/rfc3056, February 2001.

[18] Benjie Chen and Robert Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *Proceedings of the USENIX Annual Technical Conference,* pages 333-346, Boston, Massachusetts, June 2001.

[19] Michael K. Chen, Xiao-Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation,* Chicago, Illinois, June 2005.

[20] David Culler, Richard Karp, Ramesh Subramonian, and Thorsten Von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of*

the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 1–12, San Diego, California, May 1993.

[21] S. Deering, Cisco, R. Hinden, and Nokia. Internet Protocol, Version 6 (IPv6) Specification. http://tools.ietf.org/html/rfc2460, December 1998.

[22] Will Eatherton. The Push of Network Processing to the Top of the Pyramid. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems,* Princeton, New Jersey, October 2005.

[23] Philip Emma. The End of Scaling? Revolutions in Technology and Microarchitecture as We Pass the 90 Nanometer Node. In *Proceedings of the International Symposium on Computer Architecture,* Boston, Massachusetts, June 2006.

[24] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The *nesC* Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation,* San Diego, California, June 2003.

[25] Lal George and Matthias Blume. Taming the IXP Network Processor. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation,* pages 26–37, San Diego, California, June 2003.

[26] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: Programming General-Purpose Multicore Procesors Using Streams. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 297-307, Seattle, Washington, March 2008.

[27] Peter Hofstee. Power Efficient Processor Architecture and the Cell Processor. In *Proceedings of the International Conference on High-Performance Com-*

puter Architecture, IEEE Computer Society, San Francisco, California, February 2005.

[28] ITU-T. INformation TEchnology - OPen SYstems INterconnection - BAsic REference MOdel. Recommendation X.200, July 1994.

[29] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. In *ACM Computing Surveys,* 36(1):1–34, March 2004.

[30] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained.* Addison-Wesley, Boston, Massachusetts, 2003.

[31] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *ACM Transactions on Computer Systems,* 18(3):263–297, August 2000.

[32] Ravi Kokku, Taylor L. Riché, Aaron Kunze, Jayaram Mudigonda, Jamie Jason, and Harrick M. Vin. A Case for Run-Time Adaptation in Packet Processing Systems. In *Proceedings of the Workshop on Hot Topics in Networks,* Boston, Massachusetts, November 2003.

[33] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events Can Make Sense. In *Proceedings of the USENIX Annual Technical Conference,* Santa Clara, California, June 2007.

[34] Peng Li and Steve Zdancewic. Combining Events and Threads for Scalable Network Services. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation,* 2007.

[35] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A Programming Model for Heterogeneous Multi-Core Systems. In *Pro-*

ceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pages 287–296, Seattle, Washington, March 2008.

[36] Michael Macedonia. The GPU Enters Computing's Mainstream. In *IEEE Computer,* 36(10):106–108, 2003.

[37] Jayaram Mudigonda. Addressing the Memory Bottleneck in Packet Processing Systems. Ph.D. Thesis, The University of Texas at Austin, 2005.

[38] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. In *Advances in Computers,* 46, August 1998.

[39] John C. Reynolds. The Discoveries of Continuations. In *LISP and Symbolic Computation: An International Journal,* 6:233–247, 1993.

[40] Taylor L. Riché, Jayaram Mudigonda, and Harrick M. Vin. Experimental Evaluation of Load Balancers in Packet Processing Systems. In *Workshop on Building Block Engine Architectures for Computers and Networks,* Boston, MA, October 2004.

[41] Christopher Strachey and Christopher P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. In *Journal of Higher-Order and Symbolic Computation,* 13:135–152, 2000.

[42] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction,* Grenoble, France, April 2002.

[43] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF RFC 2766, 2000.

[44] Jonathan S. Turner. A Proposed Architecture for the GENI Backbone Platform. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems,* San Jose, California, October 2006.

[45] Gautam Upadhyaya, Vijay S. Pai, and Samuel P. Midkiff. Expressing and Exploiting Concurrency in Networked Applications in Aspen. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* San Jose, California, March 2007.

[46] J. Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric A. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Symposium on Operating Systems Principles,* pages 268-281, Bolton Landing, New York, October 2003.

[47] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Symposium on Operating Systems Principles,* pages 230-243, Banff, Canada, October 2001.

Vita

Taylor was born in 1978 in Baton Rouge, Louisiana. He is the son of Robert and Bobbie Riché. Taylor graduated *magna cum laude* with honors from Tulane University in 2000 receiving a Bachelor of Science and Engineering degree in Computer Engineering and Mathematics. He then worked at International Business Machines Corporation for one year as a software engineer designing and implementing the input-output subsystem for their line of IA64 servers. He left IBM in 2001 to start the doctoral program in the Department of Computer Sciences at the University of Texas at Austin. Taylor received a Masters of Science in Computer Sciences degree from the University of Texas at Austin in 2004. He married Carrie Annette Jacobs in March of 2004.

Permanent Address: 1510 W. North Loop Blvd. # 424
Austin, TX 78756

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.