Flux: A Language for Programming High-Performance Servers

Flux: A Language for Programming High-Performance Servers

Brendan Burns Kevin Grimaldi Alexander Kostadinov Emery D. Berger Mark D. Corner
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{bburns,kgrimald,akostadi,emery,mcorner}@cs.umass.edu

Abstract

Programming high-performance server applications is challenging: it is both complicated and error-prone to write the concurrent code required to deliver high performance and scalability. Server performance bottlenecks are difficult to identify and correct. Finally, it is difficult to predict server performance prior to deployment.

This paper presents Flux, a language that dramatically simplifies the construction of scalable high-performance server applications. Flux lets programmers compose off-the-shelf, sequential C or C++ functions into concurrent servers. Flux programs are type-checked and guaranteed to be deadlock-free. We have built a number of servers in Flux, including a web server with PHP support, an image-rendering server, a BitTorrent peer, and a game server. These Flux servers match or exceed the performance of their counterparts written entirely in C. By tracking hot paths through a running server, Flux simplifies the identification of performance bottlenecks. The Flux compiler also automatically generates discrete event simulators that accurately predict actual server performance under load and with different hardware resources.

1 Introduction

Server applications need to provide high performance while handling large numbers of simultaneous requests. However, programming servers remains a daunting task. Concurrency is required for high performance but introduces errors like race conditions and deadlock that are difficult to debug. The mingling of server logic with low-level systems programming complicates development and makes it difficult to understand and debug server applications. Consequently, the resulting implementations are often either lacking in performance, buggy or both. At the same time, the interleaving of multiple threads of server logic makes it difficult to identify performance bottlenecks or predict server performance prior to deployment.

This paper introduces *Flux*, a domain-specific language that addresses these problems in a declarative, flow-oriented language (Flux stems from the Latin word for "flow"). A Flux program describes two things: (1) the flow of data from client requests through nodes, typically off-the-shelf C or C++ functions, and (2) mutual exclusion requirements for these nodes, expressed as high-level *atomicity constraints*. Flux requires no other typical programming language constructs like variables or loops – a Flux program executes inside an implicit infinite loop. The Flux compiler combines the C/C++ components into a high performance server using just the flow connectivity and atomicity constraints.

Flux captures a programming pattern common to server applications: concurrent executions, each based on a client request from the network and a subsequent response. This focus enables numerous advantages over conventional server programming:

- **Ease of use.** Flux is a declarative, implicitly-parallel coordination language that eliminates the error-prone management of concurrency via threads or locks. A typical Flux server requires just tens of lines of code to combine off-the-shelf components written in sequential C or C++ into a server application.
- **Reuse.** By design, Flux directly supports the incorporation of unmodified existing code. There is no "Flux API" that a component must adhere to; as long as components follow the standard UNIX conventions, they can be incorporated unchanged. For example, we were able to add PHP support to our web server just by implementing a required PHP interface layer.
- **Runtime independence.** Because Flux is not tied to any particular runtime model, it is possible to deploy Flux programs on a wide variety of runtime systems. Section 3 describes three runtimes we have implemented: thread-based, thread pool, and event-driven.
- **Correctness.** Flux programs are type-checked to ensure their compositions make sense. The atomicity constraints eliminate deadlock by enforcing a canonical ordering for lock acquisitions.
- **Performance prediction.** The Flux compiler optionally outputs a discrete event simulator. As we

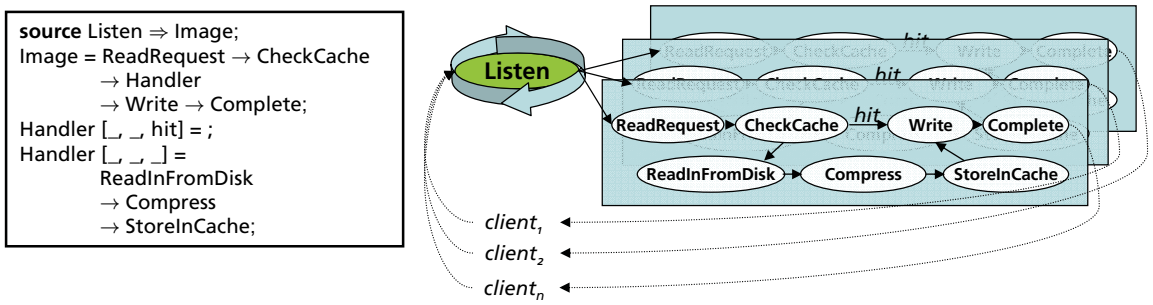

Figure 1: An example Flux program and a dynamic view of its execution.

show in Section 5, this simulator accurately predicts actual server performance.

- **Bottleneck analysis.** Flux servers include lightweight instrumentation that identifies the most-frequently executed or most expensive paths in a running Flux application.

Our experience with Flux has been positive. We have implemented a wide range of server applications in Flux: a web server with PHP support, a BitTorrent peer, an image server, and a multi-player online game server. The longest of these consists of fewer than 100 lines of code, with the majority of the code devoted to type signatures. In every case, the performance of these Flux servers matches or exceeds that of their hand-written counterparts.

The remainder of this paper is organized as follows. Section 2 presents the semantics and syntax of the Flux language. Section 3 describes the Flux compiler and runtime systems. Section 4 presents our experimental methodology and compares the performance of Flux servers to their hand-written counterparts. Section 5 demonstrates the use of path profiling and discrete-event simulation. Section 6 reports our experience using Flux to build several servers. Section 7 presents related work, and Section 8 concludes with a discussion of planned future work.

2 Language Description

To introduce Flux, we develop a sample application that exercises most of Flux's features. This sample application is an image server that receives HTTP requests for images that are stored in the PPM format and compresses them into JPEGs, using calls to an off-the-shelf JPEG library. Recently-compressed images are stored in a cache managed with a least-frequently used (LFU) replacement policy.

Figure 1 presents an abbreviated listing of the image server code with a schematic view of its dynamic execution (see Figure 2 for a more detailed listing). The `Listen` node (a "source node") executes in an infinite loop, handling client requests and transferring data and control to the server running the Flux program. A single Flux program represents an unbounded number of separate concurrent flows: each request executes along a separate flow through the Flux program, and eventually outputs results back to the client.

Notice that Flux programs are acyclic. The only loops exposed in Flux are the implicit infinite loops in which source nodes execute, and the round-trips between the Flux server and its clients. The lack of cycles in Flux allows it to enforce deadlock-free concurrency control. While theoretically limiting expressiveness, we have found cycles to be unnecessary for implementing in Flux the wide range of servers described in Section 4.

The Flux language consists of a minimal set of features, including **concrete nodes** that correspond to the C or C++ code implementing the server logic, **abstract nodes** that represent a flow through multiple nodes, **predicate types** that implement conditional data flow, **error handlers** that deal with exceptional conditions, and **atomicity constraints** that control simultaneous access to shared state.

2.1 Concrete Nodes

The first step in designing a Flux program is describing the *concrete nodes* that correspond to C and C++ functions. Flux requires type signatures for each node. The name of each node is followed by the input arguments in parentheses, followed by an arrow and the output arguments. Functions implementing concrete nodes require either one or two arguments. If the node is a source node, then it requires only one argument: a pointer to a `struct` that the function fills in with its outputs. Similarly, if the node is a sink node (without output), then its argument is a pointer to a `struct` that holds the function's inputs. Most concrete nodes have two arguments: first input, then output.

Figure 2 starts with the signatures for three of the concrete nodes in the image server: `ReadRequest` parses client input, `Compress` compresses images, and `Write` outputs the compressed image to the client.

While most concrete nodes both receive input data

```
// concrete node signatures
Listen ()
  => (int socket);

ReadRequest (int socket)
  => (int socket, bool close,
      image_tag *request);

CheckCache (int socket, bool close,
            image_tag *request)
        => (int socket, bool close,
            image_tag *request);

// omitted for space:
//   ReadInFromDisk, StoreInCache

Compress (int socket, bool close,
          image_tag *request,
          __u8 *rgb_data)
  => (int socket, bool close,
      image_tag *request);

Write (int socket, bool close,
       image_tag *request)
  => (int socket, bool close,
      image_tag *request);

Complete (int socket, bool close,
      image_tag *request) => ();

// source node
source Listen => Image;

// abstract node
Image = ReadRequest -> CheckCache
  -> Handler -> Write -> Complete;

// predicate type & dispatch
typedef hit TestInCache;

Handler:[_, _, hit] = ;
Handler:[_, _, _] =
  ReadInFromDisk -> Compress
  -> StoreInCache;

// error handler
handle error ReadInFromDisk => FourOhFor;

// atomicity constraints
atomic CheckCache:{cache};
atomic StoreInCache:{cache};
atomic Complete:{cache};
```

Figure 2: An image compression server, written in Flux.

and produce output, *source* nodes only produce output to initiate a data flow. The statement below indicates that `Listen` is a source node, which Flux executes inside an infinite loop. Whenever `Listen` receives a connection, it transfers control to the `Image` node.

```
// source node
source Listen => Image;
```

2.2 Abstract Nodes

In Flux, concrete nodes can be composed to form *abstract nodes*. These abstract nodes represent a flow of data from concrete nodes to concrete nodes or other abstract nodes. Arrows connect nodes, and Flux checks to ensure that these connections make sense. The output type of the node on the left side of the arrow must match the input type of the node on the right side. For example, the abstract node `Image` in the image server corresponds to a flow from client input that checks the cache for the requested image, handles the result, writes the output, and completes.

```
// abstract node
Image = ReadRequest -> CheckCache
  -> Handler -> Write -> Complete;
```

2.3 Predicate Types

A client request for an image may result in either a cache hit or a cache miss. These need to be handled differently. Instead of exposing control flow directly, Flux lets programmers use the *predicate type* of a node's output to direct the flow of data to the appropriate subsequent node. A predicate type is an arbitrary Boolean function supplied by the Flux programmer that is applied to the node's output.

Using predicate types, a Flux programmer can express multiple possible paths for data through the server. Predicate type dispatch is processed in order of the tests in the Flux program. The `typedef` statement binds the type `hit` to the Boolean function `TestInCache`. The node `Handler` below checks to see if its first argument is of type `hit`; in other words, it applies the function `TestInCache` to the third argument. The underscores are wildcards that match any type. `Handler` does nothing for a hit, but if there is a miss in the cache, the image server fetches the PPM file, compresses it, and stores it in the cache.

```
// predicate type & dispatch
typedef hit TestInCache;

Handler:[_, _, hit] = ;
Handler:[_, _, _] =
  ReadInFromDisk -> Compress
  -> StoreInCache;
```

2.4 Error Handling

Any server must handle errors. Flux expects nodes to follow the standard UNIX convention of returning error codes. Whenever a node returns a non-zero value, Flux checks if an error handler has been declared for the node. If none exists, the current data flow is simply terminated.

In the image server, if the function that reads an image from disk discovers that the image does not exist, it returns an error. We handle this error by directing the flow to a node `FourOhFour` that outputs a 404 page:

```
// error handler
handle error ReadInFromDisk => FourOhFour;
```

2.5 Atomicity Constraints

All flows through the image server access a single shared image cache. Access to this shared resource must be controlled to ensure that two different data flows do not interfere with each other's operation.

The Flux programmer specifies such *atomicity constraints* in Flux rather than inside the component implementation. The programmer specifies atomicity constraints by using arbitrary symbolic names. These constraints can be thought of as locks, although this is not necessarily how they are implemented. A node only runs when it has "acquired" all of the constraints. This acquisition follows a two-phase locking protocol: the node acquires ("locks") all of the constraints in order, executes the node, and then releases them in reverse order.

Atomicity constraints can be specified as either *readers* or *writers*. Using these constraints allows multiple readers to execute a node at the same time, supporting greater efficiency when most nodes read shared data rather than update it. Reader constraints have a question mark appended to them ("?"). Although constraints are considered writers by default, a programmer can append an exclamation point ("!") for added documentation.

In the image server, the image compression cache can be updated by three nodes: `CheckCache`, which increments a reference count to the cached item, `StoreInCache`, which writes a new item into the cache, evicting the least-frequently used item with a zero reference count, and `Complete`, which decrements the cached image's reference count. Only one instance of each node may safely execute at a time; since all of them modify the cache, we label them with the same writer constraint (`cache`).

```
// atomicity constraints
atomic CheckCache:{cache};
atomic StoreInCache:{cache};
atomic Complete:{cache};
```

Note that programmers can apply atomicity constraints not only to concrete nodes but also to abstract nodes. In this way, programmers can specify that multiple nodes must be executed atomically. For example, the node `Handler` could also be annotated with an atomicity constraint, which would span the execution of the path `ReadInFromDisk` → `Compress` → `StoreInCache`. This freedom to apply atomicity constraints presents some complications for deadlock-free lock assignment, which we discuss in Section 3.1.1.

2.5.1 Scoped Constraints

While flows generally represent independent clients, in some server applications, multiple flows may constitute a single *session*. For example, a file transfer to one client may take the form of multiple simultaneous flows. In this case, the state of the session (such as the status of transferred chunks) only needs to be protected from concurrent access in that session.

In addition to *program-wide constraints* that apply across the entire server (the default), Flux supports *per-session constraints* that apply only to particular sessions. Using session-scoped atomicity constraints increases concurrency by eliminating contention across sessions. Sessions are implemented as hash functions on the output of each source node. The Flux programmer implements a session id function that takes the source node's output as its parameter and returns a unique session identifier, and then adds `(session)` to a constraint name to indicate that it applies only per-session.

2.5.2 Discussion

Specifying atomicity constraints in Flux rather than placing locking operations inside implementation code has a number of advantages, beyond the fact that it allows the use of libraries whose source code is unavailable.

Safety. The Flux compiler imposes a canonical ordering on atomicity constraints (see Section 3.1.1). Combined with the fact that Flux flows are acyclic, this ordering prevents cycles from appearing in its lock graph. Programs that use Flux-level atomicity constraints exclusively (i.e., that do not themselves contain locking operations) are thus guaranteed to not deadlock.

Efficiency. Exposing atomicity constraints also enables the Flux compiler to generate more efficient code for particular environments. For example, while a multi-threaded runtime require locks, a single-threaded event-driven runtime does not. The Flux compiler generates locks or other mutual exclusion operations only when needed.

Granularity selection. Finally, atomicity constraints let programmers easily find the appropriate granularity of locking — they can apply fine-grained constraints to individual concrete nodes or coarse-grained constraints to abstract nodes that comprise many concrete nodes.

However, even when deadlock-freedom is guaranteed, grain selection can be difficult: too coarse a grain results in contention, while too fine a grain can impose excessive locking overhead. As we describe in Section 5.1, Flux can generate a discrete event simulator for the Flux program. This simulator can let a developer measure the effect of different granularity decisions and identify the appropriate locking granularity before actual server deployment.

3 Compiler and Runtime Systems

A Flux program is transformed into a working server by a multi-stage process. The compiler first reads in the Flux source and constructs a representation of the program graph. It then processes the internal representation to type-check the program. Once the code has been verified, the runtime code generator processes the graph and outputs C code that implements the server's data flow for a specific runtime. Finally, this code is linked with the implementation of the server logic into an operational server. We first describe the compilation process in detail. We then describe the three runtime systems that Flux currently supports.

3.1 The Flux Compiler

The Flux compiler is a three-pass compiler implemented in Java, and uses the JLex lexer [5] in conjunction with the CUP LALR parser generator [3].

The first pass parses the Flux program text and builds a graph-based internal representation. During this pass, the compiler links nodes referenced in the program's data flows. All of the conditional flows are merged, with an edge corresponding to each conditional flow.

The second pass decorates edges with types, connects error handlers to their respective nodes, and verifies that the program is correct. First, each node mentioned in a data flow is labelled with its input and output types. Each predicate type used by a conditional node is associated with its user-supplied predicate function. Finally, the error handlers and atomicity constraints are attached to each node. If any of the referenced nodes or predicate types are undefined, the compiler signals an error and exits. Otherwise, the program graph is completely instantiated. The final step of program graph construction checks that the output types of each node match the inputs of the nodes that they are connected to. If all type tests pass, then the compiler has a valid program graph.

The third pass generates the intermediate code that implements the data flow of the server. Flux supports generating code for arbitrary runtime systems. The compiler defines an object-oriented interface for code generation. New runtimes can easily be plugged into the Flux compiler by implementing this code generator interface.

The current Flux compiler supports several different runtimes, described below. In addition to the runtime-specific intermediate code, the Flux compiler generates a Makefile and stubs for all of the functions that provide the server logic. These stubs ensure that the programmer uses the appropriate signatures for these methods. When appropriate, the code generator outputs locks corresponding to the atomicity constraints.

3.1.1 Avoiding Deadlock

The Flux compiler generates locks in a canonical order. Our current implementation sorts them alphabetically by name. In other words, a node that has `y,x` as its atomicity constraints actually first acquires `x`, then `y`.

When applied only to concrete nodes, this approach straightforwardly combines with Flux's acyclic graphs to eliminate deadlock. However, when abstract nodes also require constraints, the constraints may become nested and preventing deadlock becomes more complicated. Nesting could itself cause deadlock by acquiring constraints in non-canonical order. Consider the following Flux program fragment:

```
A = B;
C = D;

atomic A: {x};
atomic B: {y};
atomic C: {y};
atomic D: {x};
```

In this example, a flow passing through `A` (which then invokes `B`) locks `x` and then `y`. However, a flow through `C` locks `y` and then `x`, which is a cycle in the locking graph.

To prevent deadlock, the Flux compiler detects such situations and moves up the atomicity constraints in the program, forcing earlier lock acquisition. For each abstract node with atomicity constraints, the Flux compiler computes a constraint list comprising the atomicity constraints the node transitively requires, in execution order. This list can easily be computed via a depth-first traversal of the relevant part of the program graph. If a constraint list is out of order, then the first constraint acquired in a non-canonical order is added to the parent of the node that requires the constraint. This process repeats until no out-of-order constraint lists remain.

For the above example, Flux will discover that node `C` has an out-of-order sequence (`y`, `x`). It then adds constraint `x` to node `C`. The algorithm then terminates with the following set of constraints:

```
atomic A: {x};
atomic B: {y};
atomic C: {x,y};
atomic D: {x};
```

Flux locks are reentrant, so multiple lock acquisitions do not present any problems. However, reader and writer locks require special treatment. After computing all constraint lists, the compiler performs a second pass to find any instances when a lock is acquired at least once as a reader and a writer. If it finds such a case, Flux changes the first acquisition of the lock to a writer if it is not one already. Reacquiring a constraint as a reader while possessing it as a writer is allowed because it does not cause the flow to give up the writer lock.

Because early lock acquisition can reduce concurrency, whenever the Flux compiler discovers and resolves potential deadlocks as described above, it generates a warning message.

3.2 Runtime Systems

The current Flux compiler supports three different runtime systems: one thread per connection, a thread-pool system, and an event-driven runtime.

3.2.1 Thread-based Runtimes

In the thread-based runtimes, each request handled by the server is dispatched to a thread function that handles all possible paths through the server's data flows. In the one-to-one thread server, a thread is created for every different data flow. In the thread-pool runtime, a fixed number of threads are allocated to service data flows. If all threads are occupied when a new data flow is created, the data flow is queued and handled in first-in first-out order.

3.2.2 Event-driven Runtime

Event-driven systems can provide robust performance with lower space overhead than thread-based systems [25]. In the event-driven runtime, every input to a functional node is treated as an event. Each event is placed into a queue and handled in turn by a single thread. Additionally, each source node (a node with no input) is repeatedly added to the queue to originate each new data flow. The transformation of input to output by a node generates a new event corresponding to the output data propagated to the subsequent node.

The implementation of the event-based runtime is complicated by the fact that node implementations may perform blocking function calls. If blocking function calls like `read` and `write` were allowed to run unmodified, the operation of the entire server would block until the function returned.

Instead, the event-based runtime intercepts all calls to blocking functions using a handler that is pre-loaded via the `LD_PRELOAD` environment variable. This handler captures the state of the node at the blocking call and moves to the next event in the queue. The formerly-blocking call is then executed asynchronously. When the event-based runtime receives a signal that the call has completed, the event is reactivated and re-queued for completion. Because the mainstream Linux kernel does not currently support callback-driven asynchronous I/O, the current Flux event-based runtime uses a separate thread to simulate callbacks for asynchronous I/O using the `select` function. A programmer is thus free to use synchronous I/O primitives without interfering with the operation of the event-based runtime.

3.2.3 Other Languages and Runtimes

Each of these runtimes was implemented in the C using POSIX threads and locks. Flux can also generate code for different programming languages. We have also implemented a prototype that targets Java, using both SEDA [25] and a custom runtime implementation, though we do not evaluate the Java systems here.

In addition to these runtimes, we have implemented a code generator that transforms a Flux program graph into code for the discrete event simulator CSIM [18]. This simulator can predict the performance of the server under varying conditions, even prior to the implementation of the core server logic. Section 5.1 describes this process in greater detail.

4 Experimental Evaluation

To demonstrate its effectiveness for building high-performance server applications, we implemented a number of servers in Flux. We summarize these in Table 1. We chose these servers specifically to span the space of possible server applications. Most server applications can be broadly classified into one of the following categories, based on how they interact with clients: request-response client/server, "heartbeat" client/server and peer-to-peer.

We implemented a server in Flux for each of these categories and compared its performance under load with existing hand-tuned server applications written in conventional programming languages. The Flux servers rely on single-threaded C and C++ code that we either borrowed from existing implementations or wrote ourselves. The most significant inclusions of existing code were in the web server, which uses the PHP interpreter, and in the image server, which relies on calls to the `libjpeg` library to compress JPEG images.

4.1 Methodology

We evaluate all server applications by measuring their throughput and latency in response to realistic workloads.

All testing was performed with a server and client machine, both running Linux version 2.4.20. The server machine was a Pentium 4 (2.4Ghz, 1GB RAM), connected via gigabit Ethernet on a dedicated switched

Server	Style	Description	Lines of Flux code	Lines of C/C++ code
Web server	request-response	a basic HTTP/1.1 server with PHP	36	386 (+ PHP)
Image server	request-response	image compression server	23	551 (+ `libjpeg`)
BitTorrent	peer-to-peer	a file-sharing server	84	878
Game server	heartbeat client-server	multiplayer game of "Tag"	54	257

Table 1: Servers implemented using Flux, described in Section 4.

network to the client machine, a Xeon-based machine (2.4Ghz, 1GB RAM). All server and client applications were compiled using GCC version 3.2.2. During testing, both machines were running in multi-user mode with only standard services running. All results are for a run of two minutes, ignoring the first twenty seconds to allow the cache to warm up.

4.2 Request-Response: Web Server

Request-response based client/server applications are among the most common examples of network servers. This style of server includes most major Internet protocols including FTP, SMTP, POP, IMAP and HTTP. As an example of this application class, we implemented a web server in Flux. The Flux web server implements the HTTP/1.1 protocol and can serve both static and dynamic PHP web pages.

We implemented a benchmark to load test the Flux webserver that is similar to SPECweb99 [21]. The benchmark simulates a number of clients requesting files from the server. Each simulated client sends five requests over a single HTTP/1.1 TCP connection using keep-alives. When one file is retrieved, the next file is immediately requested. After the five files are retrieved, the client disconnects and reconnects over a new TCP connection. The files requested by each simulated client follow the static portion of the SPECweb benchmark and each file is selected using the Zipf distribution. The working set for this benchmark is approximately 32MB, which fits into RAM, so this benchmark primarily stresses CPU performance.

We compare the performance of the Flux webserver against the latest versions of the *knot* webserver distributed with Capriccio [24] and the *Haboob* webserver distributed with the SEDA runtime system [25]. Figure 3 presents the throughput and latency for a range of simultaneous clients. These graphs represent the average of five different runs for each number of clients.

The results show that the Flux web server provides comparable performance to the fastest webserver (*knot*), regardless of whether the event-based or thread-based runtime is used. All three of these servers (knot, flux-threadpool and flux-event-based) significantly outperform *Haboob*, the event-based server distributed with SEDA. As expected, the naïve one-thread, one-client server generated by Flux has significantly worse performance due to the overhead of creating and destroying threads.

The results for the event-based server highlight one drawback of running on a system without true asynchronous I/O. With small numbers of clients, the event-based server suffers from increased latency that initially decreases and then follows the behavior of the other servers. This hiccup is an artifact of the interaction between the webserver's implementation and the event-driven runtime, which must simulate asynchronous I/O. The first node in the webserver uses the `select` function with a timeout to wait for network activity. In the absence of other network activity, this node will block for a relatively long period of time. Because the event-based runtime only reactivates nodes that make blocking I/O calls after the completion of the currently-operating node, in the absence of other network activity, the call to `select` imposes a minimum latency on all blocking I/O. As the number of clients increases, there is sufficient network activity that `select` never reaches its timeout and frozen nodes are reactivated at the appropriate time. In the absence of true asynchronous I/O, the only solution to this problem would be to decrease the timeout call to `select`, which would increase the CPU usage of an otherwise idle server.

4.3 Peer-to-Peer: BitTorrent

Peer-to-peer applications act as both a server and a client. Unlike a request-response server, they both receive and initiate requests.

We implemented a BitTorrent server in Flux as a representative peer-to-peer application. BitTorrent uses a scatter-gather protocol for file sharing. BitTorrent peers exchange pieces of a shared file until all participants have a complete copy. Network load is balanced by randomly requesting different pieces of the file from different peers.

To facilitate benchmarking, we changed the behavior of both of the BitTorrent peers we test here (the Flux version and CTorrent). First, all client peers are *unchoked* by default. Choking is an internal BitTorrent state that blocks certain clients from downloading data. This protocol restriction prevents real-world servers from being overwhelmed by too many client requests. We also allow

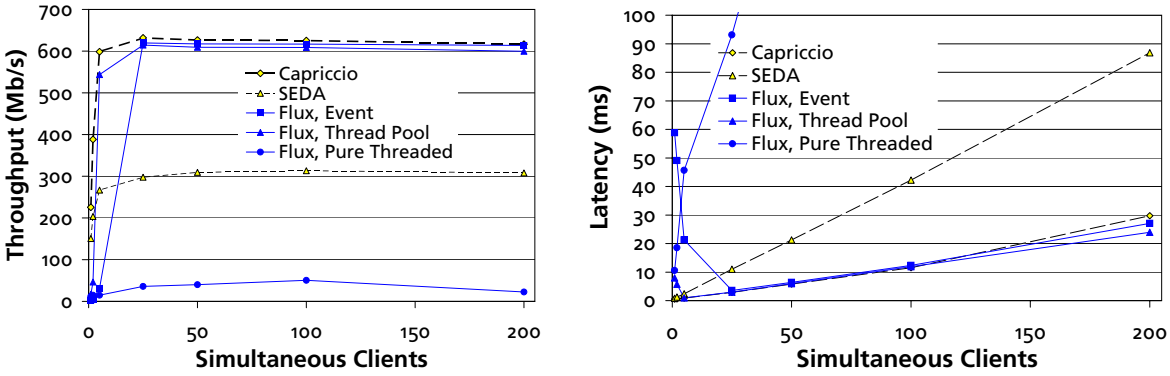

Figure 3: Comparison of Flux web servers with other high-performance implementations (see Section 4.2).

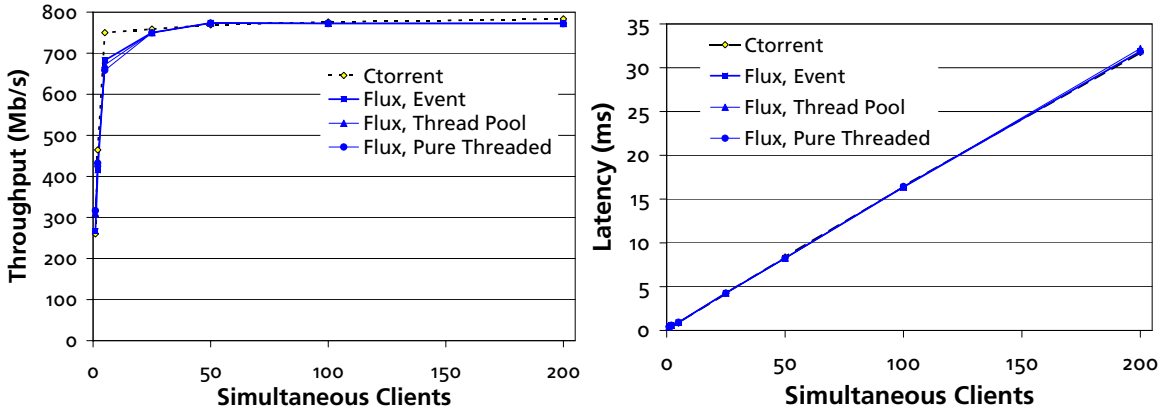

Figure 4: Comparison of Flux BitTorrent servers with CTorrent (see Section 4.3).

an unlimited number of unchoked client peers to operate simultaneously, while the real BitTorrent server only unchokes clients who upload content.

We are unaware of any existing BitTorrent benchmarks, so we developed our own. Our BitTorrent benchmark mimics the traffic encountered by a busy BitTorrent peer and stresses server performance. It simulates a series of clients continuously sending requests for randomly distributed pieces of a 54MB test file to a BitTorrent peer with a complete copy of the file. When a peer finishes downloading a piece of the file, it immediately requests another random piece of the file from those still missing. Once a client has obtained the entire file, it disconnects. This benchmark does not simulate the "scatter-gather" nature of the BitTorrent protocol; instead, all requests go to a single peer. Using single peers has the effect of maximizing load, since obtaining data from a different source would lessen the load on the peer being tested.

Figure 4 compares the latency, throughput in completions per second and network throughput to CTorrent, an implementation of the BitTorrent protocol written in C. The goal of any BitTorrent system is to maximize network utilization (thus saturating the network), and both the CTorrent and Flux implementations achieve this goal. However, prior to saturating the network, all of the Flux servers perform slightly worse than the CTorrent server. We are investigating the cause of this small performance gap.

4.4 Heartbeat Client-Server: Game Server

Unlike request-response client/server applications and most peer-to-peer applications, certain server applications are subject to deadlines. An example of such a server is an online multi-player game. In these applications, the server maintains the shared state of the game and distributes this state to all of the players at "heartbeat" intervals. There are two important conditions that must be met by this communication: the state possessed by all clients must be the same at each instant in time, and the inter-arrival time between states can not be too great. If either of these conditions is violated, the game will be unplayable or susceptible to cheating. These requirements place an important delay-sensitive constraint on the server's performance.

We have implemented an online multi-player game of Tag in Flux. The Flux game server enforces the rules of Tag. Players can not move beyond the boundaries of the game world. When a player is tagged by the player who is "it", that player becomes the new "it" and is teleported to a new random location on the board. All communication between clients and server occurs over UDP at 10Hz, a rate comparable to other real-world online games. While simple, this game has all of the important characteristics of servers for first person shooter or real-time strategy games.

Benchmarking the gameserver is significantly different than load-testing either the webserver or BitTorrent peer. Throughput is not a consideration since only small pieces of data are transmitted. The primary concern is the latency of the server as the number of clients increases. The server must receive every player's move, compute the new game state, and broadcast it within a fixed window of time.

To load-test the game server, we measured the effect of increasing the number of players. The performance of the gameserver is largely based upon the length of time it takes the server to update the game state given the moves received from all of the players, and this computation time is identical across the servers. The latency of the gameserver is largely a product of the rate of game turns, which stays constant at 10Hz. We found no appreciable differences between a traditional implementation of the gameserver and the various Flux versions. These results show that Flux is capable of producing a server with sufficient performance for multi-player online gaming.

5 Performance

In addition to its programming language support for writing server applications, Flux provides support for predicting and measuring the performance of server applications. The Flux system can generate **discrete-event simulators** that predict server performance for synthetic workloads and on different hardware. It can also perform **path profiling** to identify server performance bottlenecks on a deployed system.

5.1 Performance Prediction

Predicting the performance of a server prior to deployment is important but often difficult. For example, performance bottlenecks due to contention may not appear during testing because the load placed on the system is insufficient. In addition, system testing on a small-scale system may not reveal problems that arise when the system is deployed on an enterprise-scale multiprocessor.

In addition to generating executable server code, the Flux code generator can automatically transform a Flux program directly into a discrete-event simulator that models the performance of the server. The implementation language for the simulator is CSim [18], a C-based, process-oriented simulator.

```
void Image (){
  rw_write_lock(lock);
  processor->reserve();
  hold(exponential(CMP_TIME_CPU_IMAGE));
  processor->release();
  rw_write_unlock(lock);
  // Call the next Node
  ReadRequest();
}
```

Figure 5: Compiler-generated discrete-event simulation code for a Flux node.

In the simulator, CPUs are modeled as resources that each Flux node acquires for a given amount of time. The simulator can either use observed parameters from a running system (per-node execution times, source node inter-arrival times, and observed branching probabilities), or the Flux programmer can supply estimates for these parameters. The simulator can model an arbitrary number of processors by increasing the number of nodes that may simultaneously acquire the CPU resource. When a node uses a given atomicity constraint, it treats it as a lock and acquires it for the duration of the node's execution. While the simulator accurately models both reader and writer constraints, it conservatively treats session-level constraints as globals.

The code in Figure 5 is a simplified version of the CSim code that Flux generates. Here, the node `Image` has a writer lock associated with it. Once CSim schedules this node onto a CPU, it models its execution time using an exponential distribution based on the observed CPU time collected from a profiling run. Finally, this node unlocks its reader-writer lock and executes the next node in the flow.

It is important to note that this simulation does not model disk or network resources. While this is a realistic assumption for CPU-bound servers (such as dynamic web-servers), other servers may require more complete modeling.

To demonstrate that the generated simulations accurately predict actual performance, we tested the image server described in Section 2. To simulate load on the machine, we made requests at increasingly small inter-arrival times. The image server had 5 images, and our load tester randomly requests one of eight sizes (between 1/8th scale and full-size) of a randomly-chosen image. When configured to run with n "clients", the load tester issues requests at a rate of one every $1/n$ seconds. The image server is CPU-bound, with each image taking on average 0.5 seconds to compress.

We first measured the performance of this server on a 16-processor SunFire 6800, but with only a single CPU enabled. We then used the observed node runtime and branching probabilities to parameterize the generated CSIM simulator. We compare the predicted and actual performance of the server by making more processors available to the system. As Figure 6 shows, the predicted results (dotted lines) and actual results (solid lines) match closely, demonstrating the effectiveness of the simulator at predicting performance.

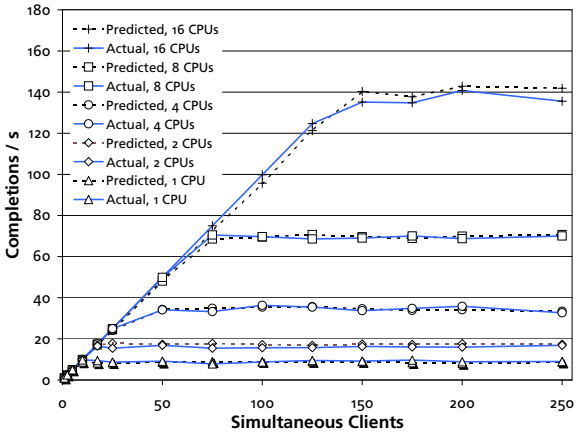

Figure 6: Predicted performance of the image server (derived from a single-processor run) versus observed performance for varying numbers of processors and load.

5.2 Path Profiling

The Flux compiler optionally instruments generated servers to simplify the identification of performance bottlenecks. This profiling information takes the form of "hot paths", the most frequent or most time-consuming paths in the server. Flux identifies these hot paths using the Ball-Larus path profiling algorithm [4]. Because Flux graphs are acyclic, the Ball-Larus algorithm identifies each unique path through the server's data-flow graph.

Hot paths not only aid understanding of server performance characteristics but also identify places where optimization would be most effective. Because profiling information can be obtained from an operating server and is linked directly to paths in the program graph, a performance analyst can easily understand the performance characteristics of deployed servers.

The overhead of path profiling is low enough that hot path information can be maintained even in a production server. Profiling adds just one arithmetic operation and two high-resolution timer calls to each node. A performance analyst can obtain path profiles from a running Flux server by connecting to a dedicated socket.

To demonstrate the use of path profiling, we compiled a version of the BitTorrent peer with profiling enabled. For the experiments, we used a patched version of Linux that supports per-thread time gathering. The BitTorrent peer was load-tested with the same tester as in the performance experiments. For profiling, we used loads of 25, 50, and 100 clients. All profiling information was automatically generated from a running Flux server.

In BitTorrent, the most time-consuming path identified by Flux was, unsurprisingly, the file transfer path (`Listen` → `GetClients` → `SelectSockets` → `CheckSockets` → `Message` → `ReadMessage` → `HandleMessage` → `Request` → `MessageDone`, 0.295 ms). However, the second most expensive path was the path that finds no outstanding chunk requests (`Listen` → `GetClients` → `SelectSockets` → `CheckSockets` → `ERROR`, 0.016ms). While this path is relatively cheap compared to the file transfer path, it also turns out to be the most frequently executed path (780,510 times, compared to 313,994 for the file transfer path). Since this path accounts for 13% of BitTorrent's execution time, it is a reasonable candidate for optimization efforts.

6 Developer Experience

In this section, we examine the experience of programmers implementing Flux applications. In particular, we focus on the implementation of the Flux BitTorrent peer.

The Flux BitTorrent peer was implemented by two undergraduate students in less than one week. The students began with no knowledge of the technical details of the BitTorrent protocol or the Flux language. The design of the Flux program for the BitTorrent peer was entirely their original work. The implementation of the functional nodes in BitTorrent is loosely derived from the CTorrent source code. The program graph for the BitTorrent server is shown in Figure 7 at the end of this document.

The students had a generally positive reaction to programming in Flux. Primarily, they felt that organizing the application into a Flux program graph prior to implementation helped modularize their application design and debug server data flow prior to programming. They also found that the exposure of atomicity constraints at the Flux language level allowed for easy identification of the appropriate locations for mutual exclusion. Flux's immunity to deadlock and the simplicity of the atomicity constraints increased their confidence in the correctness of the resulting server.

Though this is only anecdotal evidence, this experience suggests that programmers can quickly gain enough expertise in Flux to build reasonably complex server applications.

7 Related Work

This section discusses related work to Flux in the areas of coordination and data flow languages, programming language constructs, domain-specific languages, and runtime systems.

Coordination and data flow languages. Flux is an example of a *coordination language* [11] that combines existing code into a larger program in a data flow setting. There have been numerous data flow languages proposed in the literature, see Johnston et al. for a recent survey [13]. Data flow languages generally operate at the level of fundamental operations rather than at a functional granularity. One exception is CODE 2, which permits incorporation of sequential code into a dynamic flow graph, but restricts shared state to a special node type [7, 8]. Data flow languages also typically prohibit global state. For example, languages that support *streaming* applications like StreamIt [22] express all data dependencies in the data flow graph. Flux departs from these languages by supporting safe access to global state via atomicity constraints. Most importantly, these languages focus on extracting parallelism from individual programs, while Flux describes parallelism across multiple clients or event streams.

Programming language constructs. Flux shares certain linguistic concepts with previous and current work in other programming languages. Flux's predicate matching syntax is deliberately based on the pattern-matching syntax used by functional languages like ML, Miranda, and Haskell [12, 19, 23]. The PADS data description language also allows programmers to specify predicate types, although these must be written in PADS itself rather than in an external language like C [9]. Flanagan and Freund present a type inference system that computes "atomicity constraints" for Java programs that correspond to Lipton's theory of reduction [10, 16]; Flux's atomicity constraints operate at a higher level of abstraction. The Autolocker tool [17], developed independently and concurrently with this work, automatically assigns locks in a deadlock-free manner to manually-annotated C programs. It shares Flux's enforcement of an acyclic locking order and its use of two-phase lock acquisition and release.

Related domain-specific languages. Several previous domain-specific languages allow the integration of off-the-shelf code into data flow graphs, though for different domains. The Click modular router is a domain-specific language for building network routers out of existing C components [14]. Knit is a domain-specific language for building operating systems, with rich support for integrating code implementing COM interfaces [20]. In addition to its linguistic and tool support for programming server applications, Flux ensures deadlock-freedom by enforcing a canonical lock ordering; this is not possible in Click and Knit because they permit cyclic program graphs.

Runtime systems. Researchers have proposed a wide variety of runtime systems for high-concurrency applications, including SEDA [25], Hood [1, 6], Capriccio [24], Fibers [2], cohort scheduling [15], and libasync/mp [26], whose per-callback *colors* could be used to implement Flux's atomicity constraints. Users of these runtimes are forced to implement a server using a particular API. Once implemented, the server logic is generally inextricably linked to the runtime. By contrast, Flux programs are independent of any particular choice of runtime system, so advanced runtime systems can be integrated directly into Flux's code generation pass.

8 Future Work

We plan to build on this work in several directions. First, we are actively porting Flux to other architectures, especially multicore systems. We are also planning to extend Flux to operate on clusters. Because concurrency constraints identify nodes that share state, we plan to use these constraints to guide the placement of nodes across a cluster to minimize communication.

To gain more experience with Flux, we are adding further functionality to the web server. In particular, we plan to build an Apache compatibility layer so we can easily incorporate Apache modules. We also plan to enhance the simulator framework to support per-session constraints.

The entire Flux system is available for download at `flux.cs.umass.edu` via the Flux-based BitTorrent and web servers described in this paper.

9 Acknowledgments

The authors thank Gene Novark for helping to design the discrete event simulation generator, and Vitaliy Lvin for assisting in experimental setup and data gathering.

This material is based upon work supported by the National Science Foundation under CAREER Awards CNS-0347339 and CNS-0447877. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management

without manual stack management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[3] A. W. Appel, F. Flannery, and S. E. Hudson. CUP parser generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[5] E. Berk and C. S. Ananian. JLex: A lexical analyzer generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[6] R. D. Blumofe and D. Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract). In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 266–267, New York, NY, USA, 1998. ACM Press.

[7] J. C. Browne, M. Azam, and S. Sobeck. CODE: A unified approach to parallel programming. *IEEE Software*, 6(4):10–18, 1989.

[8] J. C. Browne, E. D. Berger, and A. Dube. Compositional development of performance models in POEMS. *The International Journal of High Performance Computing Applications*, 14(4):283–291, Winter 2000.

[9] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 295–304, New York, NY, USA, 2005. ACM Press.

[10] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press.

[11] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.

[12] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[13] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

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

[15] J. R. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 103–114, Berkeley, CA, USA, 2002. USENIX Association.

[16] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[17] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 346–358. ACM, Jan. 2006.

[18] Mesquite Software. The CSIM Simulator. `http://www.mesquite.com`.

[19] R. Milner. A proposal for standard ml. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 184–197, New York, NY, USA, 1984. ACM Press.

[20] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the 4th ACM Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–360, Oct. 2000.

[21] Standard Performance Evaluation Corporation. SPECweb99. http://www.spec.org/osg/web99/.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[23] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[24] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.

[25] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the*

eighteenth ACM symposium on Operating systems principles, pages 230–243, New York, NY, USA, 2001. ACM Press.

[26] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

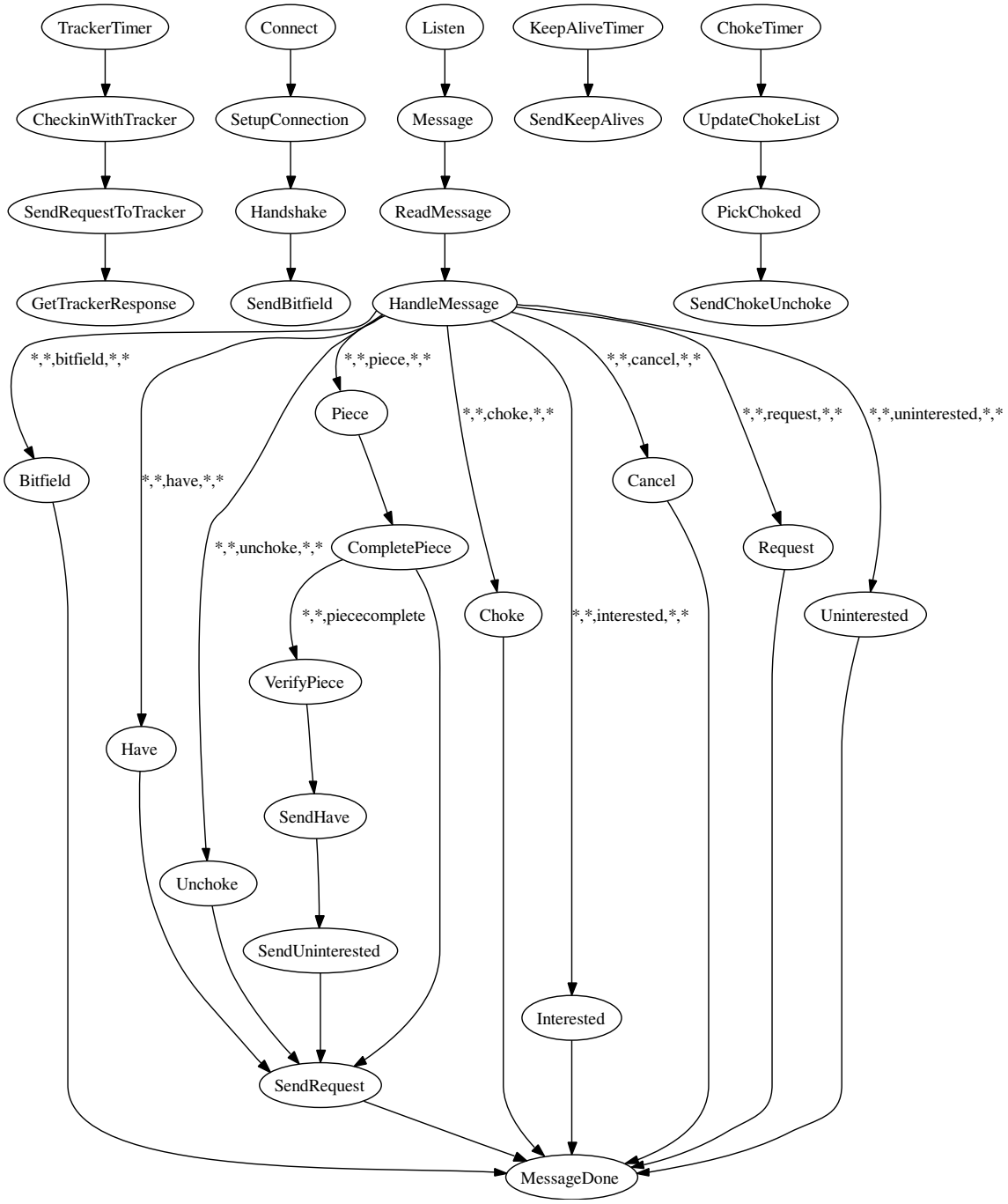

Figure 7: The Flux program graph for the example BitTorrent server.