Rethinking Application Networking as Transactional Scripting

Behram F. T. Mistree, Jay Thomason, Gabriel Kho, Harrison Ho, Edric Kyauk, and Philip Levis

Stanford University

Abstract

We describe Waldo, a scripting language for application networking. Waldo allows programmers to describe complex network interactions between many hosts as transactional operations with atomicity, consistency, and isolation. Waldo is able to provide these transactional semantics starvation-free without assuming global clocks, without centralized scheduling, and under arbitrary transaction conflicts. This allows programmers to write application networking as short transactional scripts that will never starve. Waldo achieves these results with a novel distributed transaction scheduling algorithm that combines the wound-wait algorithm and Lamport clocks with two transaction priority levels. Experimental results show that using the primary algorithm Waldo can perform up to 10,000 transctions per second between two endpoints connected across the wide area network.

1 Introduction

Driven by mobile devices and social networking, applications are increasingly networked. MagicPiano pairs users, allowing them to play music together [45]. Using Texas Hold Em, poker enthusiasts around the world can wager against each other [24]. Networking is arguably one of the hardest parts of an application, having to deal with concurrency, failure, latency, and overload. Despite this challenge, frameworks today provide little, if any, support beyond sockets and data transfer. iOS and Android, for example, provide APIs for HTTP operations, handling cookies and other details, but, in practice, this is just a data transport layer. The challenges of managing connections, concurrency, fairness, and correctness still lie with the programmer.

The success of web frameworks such as Ruby on Rails and Django have demonstrated that the right level of abstraction can greatly speed development. These and other frameworks automate the simple but time consuming parts of building web applications, by abstracting the database and simplifying query-dispatch for common queries. They make many services on the same machine, such as databases, caches, and a web server all appear like a single process.

There are also frameworks for distributed systems [41, 28, 13, 20], which help programmers build systems such as distributed hash tables and content distribution networks. To such programmers, and to systems researchers, application networking might seem comparatively simple or easy to implement. But, based on our experiences teaching an introductory networking course for several years as well as two more controlled user studies, discussed further in Section 2, we argue that networking code is extremely challenging to most developers.

Four challenges dominate application networking development time: handling failures, preventing or debugging deadlock, managing event ordering and data consistency, and handling data races. These challenges arise from the execution model common to most networking code: one or more endpoints exchange a series of messages, stepping through intermediate states in order to enact a larger, stable state transition on nodes across a network. For example, in a poker game, all players must agree to play before the game can begin; to transition from the closed to established states in TCP, an endpoint typically goes through two intermediate states as part of the three-way-handshake. In both cases, a failure in the midst of the higher-level state transition (no game to game initiated; closed to established) requires cleaning up a variety of temporary and intermediate state. Understanding when this might happen, how it might happen, and how to handle it is difficult.

This paper argues that application networking's common challenges can be avoided or eased with a better abstraction: transactions. Describing a multi-step state transition as a transaction that provides atomicity, consistency, and isolation[1] between two (or more endpoints) simplifies application logic. If a failure occurs mid-transaction, the system automatically restores to a consistent state. Similarly, transaction isolation ensures that executing concurrent transactions does not deadlock. Finally, writing an exchange as a single, atomic transaction defines the set of valid interleavings within a transaction and ensures that the inter-transaction interleavings maintain consistent state.

As we discuss in Section 4, standard centralized systems that provide these types of guarantees, such as databases, require a network round-trip time for any operation on guarded data, require setup and maintenance costs, and interface poorly with local resources. Further, existing distributed approaches such as [21] do not provide properties such as starvation prevention and/or fair access to resources, which are important in a application-networking context.

To address these issues, the key technical contribution of this paper is a novel distributed transaction scheduling algorithm. This algorithm enforces consistency, atomicity, and isolation while ensuring that no transaction starves and providing quality of service guarantees across network endpoints.

Programming transactional distributed shared memory over a wide area network may seem challenging. This paper's second contribution is Waldo, a high-level domain specific language for building application network components on top of this memory model.

Waldo provides three principal abstractions: endpoints, sequences, and services. An *endpoint* is one half of a reliable connection, coupled with application state and code. A *sequence* is a piece of linear code that controls switching execution between two endpoints. Finally, a *service* allows application logic to control, use, and manage multiple endpoints. As a simple example, to atomically send an update to a set of peers, a programmer connects an endpoint on each peer to a publisher. To send an update, a service on the publisher iterates through each of its endpoints, triggering a sequence on each. If any update sequence fails, Waldo automatically rolls back the transaction.

We evaluate Waldo by building several sample applications, focusing on two in Section 6: a networked game and a network configuration tool. Waldo is able to support 13,000 transactions per second across two endpoints connected by TCP on a single host and 10,000 transactions per second on the wide area. Increasing network latencies reduces transaction rates from this result when there are read-write conflicts. This performance is orders of magnitude greater than what most application networking requires. Using Waldo, a programmer can write application networking code using simple linear programs that run transactionally and can handle thousands of operations per second.

The rest of this paper is structured as follows. The next section describes our observations on the common problems programmers run into when writing networking code, and how those problems motivate the need for transactions. Section 3 describes the Waldo language and its abstractions, walking through a toy publish-subscribe system as an example. Section 4 describes how Waldo schedules its transactions using a novel algorithm that meets the needs of network protocols while remaining efficient and allowing parallelism. Section 5 describes a Waldo implementation that we evaluate in Section 6. Section 7 describes related work that Waldo is similar to and builds on, and Section 8 concludes.

2 Motivation

While teaching a quarter-long intermediate course on computer networking, we observed patterns of mistakes in students' programming assignments. To analyze these patterns in more controlled environments, we performed two need-finding user studies. In the first, we observed and worked with nine undergraduates for three months as they developed networked applications for a 3D virtual world [9] using a JavaScript derivative [33]. In the second, 17 graduate and upperclass undergraduate subjects wrote networked code for a simple bank-customer application in Python and an early antecedent to Waldo. Most subjects were unable to complete the application after struggling with the challenges described below.

Handling failures — Subjects in the second study were explicitly told they could ignore error conditions. In the first study, we never observed subjects' attempting to detect errors, for instance by catching exceptions, nor attempting to recover from them. In several cases, this led to incorrect behavior. For example, a bug in a virtual world bank left a flaw in which a transfer-like message from a customer would credit the transferee's account and cause an error before deducting from the transferer's. Colluding customers could invent money.

Deadlock — Almost half of the Python subjects in the second study encountered deadlock issues that they were unable to recover from. These subjects spent well over half their total study time (on average, over 30 minutes) trying to debug this problem before giving up.

Event and data management — Event and data management errors include incorrectly constructing and incorrectly handling edge cases in event orderings such that code enters an undefined state. Subjects in the second study generated their own message formats and

[1] We ignore durability in this context.

parsed messages in a large dispatch function using string-splitting and/or regular expressions. Generally, this strategy prevents type-checking when constructing and parsing messages; as a result, many errors that could have been caught by a type checker had to be debugged by intuition at runtime.

Data races — Networking code responds both to application calls and network events, and these often come from different threads of control. Subjects found lock-based concurrency management challenging. No subjects in the Python condition of the second study correctly completed the task. Although half the subjects passed all test cases, their code contained hidden, unexcercised data races.

The four challenges described above stem from a lack of atomicity, isolation, and consistency.

A lack of *atomicity* leaves systems in inconsistent states when programmers do not or incorrectly handle failure. Because the programmer above's transfer event was not atomic, a failure meant that users could defraud his application. More broadly, unless a programmer catches, handles, and forwards exceptions to all nodes processing an event, a system can get in an inconsistent state, potentially with some nodes holding locks for or awaiting responses from failed nodes.

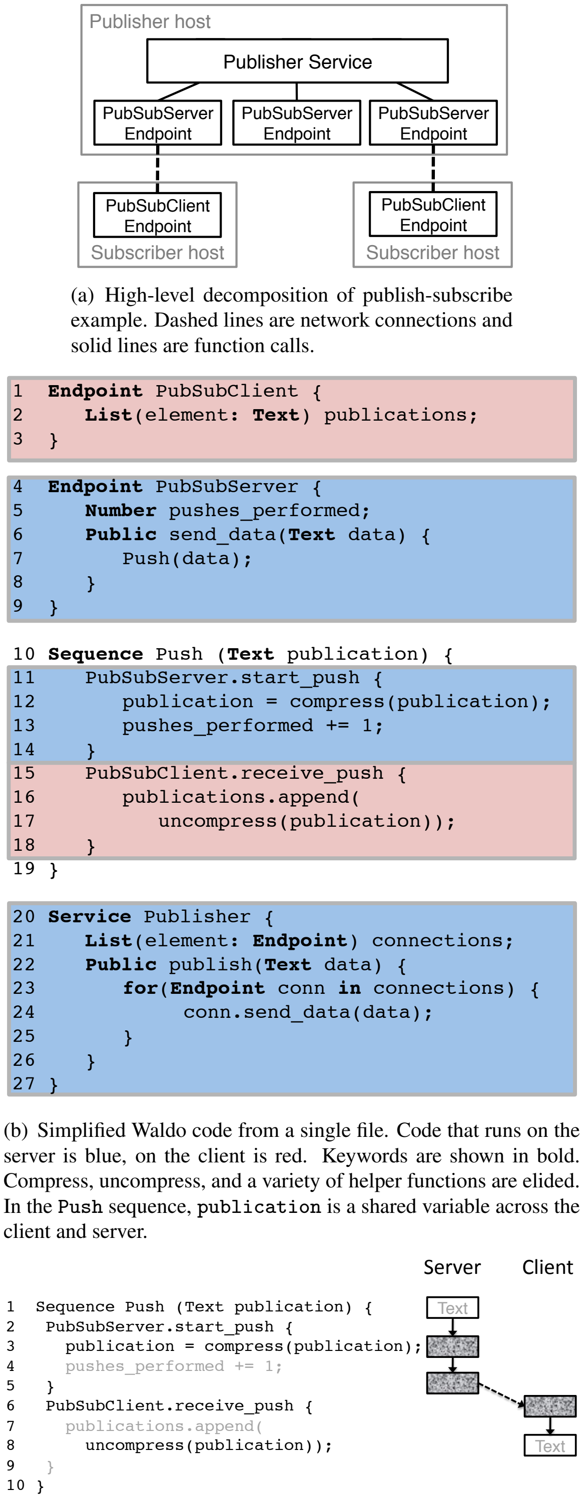
A lack of *isolation* causes deadlock. Subject code deadlocked because logically separate, but incorrectly isolated events ran concurrently.

A lack of *consistency* causes the event and data management problems described above. Instead of higher-level language support for composing sequences of messages across endpoints or for automatically constructing messages, programmers were left to perform these checks and operations themselves. Finally, a lack of isolation and consistency also cause data races.

Many networked applications may benefit from managing their own atomicity, isolation, and consistency requirements. However, as demonstrated by the user studies above, code that correctly provides atomicity, isolation, is difficult and error-prone to write and debug. And therefore there is a large group of programmers and applications that benefit from abstracting away these otherwise tricky-to-provide properties into reusable runtime components.

3 Waldo language overview

Waldo is a language for writing the networking components of applications. Programmers write the network code for their applications in Waldo, compile into whichever language their application uses (currently, Waldo emits to Java and Python) and then instantiate Waldo objects.



(a) High-level decomposition of publish-subscribe example. Dashed lines are network connections and solid lines are function calls.

```
Endpoint PubSubClient {
   List(element: Text) publications;
}

Endpoint PubSubServer {
   Number pushes_performed;
   Public send_data(Text data) {
      Push(data);
   }
}

Sequence Push (Text publication) {
   PubSubServer.start_push {
      publication = compress(publication);
      pushes_performed += 1;
   }
   PubSubClient.receive_push {
      publications.append(
         uncompress(publication));
   }
}

Service Publisher {
   List(element: Endpoint) connections;
   Public publish(Text data) {
      for(Endpoint conn in connections) {
           conn.send_data(data);
      }
   }
}
```

(b) Simplified Waldo code from a single file. Code that runs on the server is blue, on the client is red. Keywords are shown in bold. Compress, uncompress, and a variety of helper functions are elided. In the `Push` sequence, `publication` is a shared variable across the client and server.

```
Sequence Push (Text publication) {
 PubSubServer.start_push {
   publication = compress(publication);
   pushes_performed += 1;
 }
 PubSubClient.receive_push {
   publications.append(
     uncompress(publication));
 }
}
```

(c) Execution of the Push sequence. The Text publication starts on the server, where it is compressed (shaded box). The Waldo runtime automatically transfers the compressed Text to the client before invoking PubSubClient.receive_push().

Figure 1: Atomic publish/subscribe in Waldo.

Waldo provides three networking abstractions to programmers: endpoints, sequences, and services. Using these, programmers can compose transactions across state on one or many hosts in the network. An *endpoint* represents one side of a networking protocol between two hosts, and couples half of a networking connection with application state and logic. A *sequence* describes an interaction between a pair of endpoints. Finally, a *service* holds references to multiple endpoints and their shared state, allowing a Waldo programmer to control and use many connections in a single transaction.

Figure 1(b) shows a code listing for an expository atomic publish-subscribe system built using Waldo. Sections 3.1, 3.2, and 3.3 step through this example, describing the role of endpoints, sequences, and services, respectively. Section 3.5 follows with a more realistic example.

3.1 Endpoints

At a high level, a Waldo endpoint is like a basic socket in that it interfaces to a partner endpoint, presumably running on a different host. Unlike a socket, an endpoint wraps application state and specifies valid sequences of operations between partner endpoints. As shown by endpoints `PubSubServer` and `PubSubClient` in Figure 1(b), an endpoint is written as an object complete with methods (`send_data` for `PubSubServer`) and internal variables (`publications` for `PubSubClient`; `pushes_performed` for `PubSubServer`). Waldo endpoints use private data and methods by default and allow explicit specification of `Public` methods to encourage encapsulation.

When called from outside Waldo, invoking an endpoint's `Public` method automatically begins a new transaction. When called from within Waldo (*e.g.*, by a service or another endpoint), the method call is nested within the current transaction. In Figure 1(b), for example, the `Publisher` service calls `PubSubServer`'s `send_data` method.

Currently, both endpoints and services can only be created in external application code. The abstraction for creating endpoints is similar to that for creating sockets. A programmer requests the Waldo runtime to bind an endpoint factory to listen for incoming connections on an IP-port tuple. On receiving a new connection request, the factory automatically creates a new endpoint, initializing its state, and executing an optional callback. Similarly, a programmer can request an endpoint factory to connect directly to an IP-port tuple, where, presumably, another factory is listening.

3.2 Sequences

Waldo has no notion of messages, packets, packet formats, or packet parsing. Instead, endpoints implicitly exchange information through shared variables declared in a sequence. The developer focuses on specifying what the data is, which data should be private or shared, and how that data is modified. They do not need to worry about the details of getting the data over the network that do not affect the core application logic.

A sequence represents a protocol exchange between a pair of endpoints. Sequence syntax consists of three parts: a declaration, a data section, and two or more sequence steps. Like a function, a sequence declaration specifies arguments and a return type, and is invoked in a similar manner. On line 10, in Figure 1(b), the `Push` sequence for instance, takes a single argument, `publication`, and returns no data. Following the sequence declaration, a programmer can also declare variables in the sequence's data section. Not pictured in Figure 1(b), variables declared here, as well as the arguments to the sequence are scoped to the lifetime of the sequence and are shared by all sequence steps.

Finally sequence steps are program blocks that alternate execution between two endpoints. Execution "falls through" each step: when one step completes execution, the runtime updates sequence-local data and begins executing the next block, giving the impression of straight-line code. In Figure 1(b), `PubSubServer.start_push` executes on the server's host, followed by `ClientSubscriber.receive_push` on a client host. Figure 1(c) shows how this sequence behaves as it executes.

As mentioned previously, variables declared at the scope of the sequence, such as `publication` are shared by all sequence steps. For example, when `PubSubServer.start_push` assigns to `publication`. This change is visible to `ClientSubscriber.receive_push`. Furthermore, each sequence step can access the variables and methods of its corresponding endpoint. For example, `PubSubServer.start_push` increments `pushes_performed` and calls the elided `compress` method. Because `pushes_performed` is local to one endpoint, this change is not visible to the other endpoint. However, because the sequence runs atomically, if the client fails before completing the sequence the `PubSubServer` will not see its variable incremented.

The advantage of using a single structure to specify sequence logic is that it allows program structure to concretely mirror program execution. As observed in Section 2, programmers had difficulty reasoning about message exchanges. Waldo's sequences order message exchanges as straight-line code, interleaved between endpoints.

3.3 Services

Endpoints and sequences describe the relationship between a single pair of hosts. However, frequently, applications operate over many hosts: in the publish-subscribe example, many hosts all receive the same pushed update; on a file server many hosts should be able to read and modify a shared group of files; BitTorrent avoids requesting the same chunk from many peers; HTTP clients open multiple connections to reduce head-of-line blocking.

A Waldo service allows a programmer to group operations across multiple endpoints into a single transaction. In the code listing in Figure 1(b), `Publisher` (lines 20-27) is an example of a service. Similar to endpoints, non-Waldo code can call a service's `Public` methods. For instance, non-Waldo code can publish a message across all subscribers by calling `publish` (line 22).

Services and endpoints provide a well-defined interface between application protocol code and other application logic (*e.g.*, GUI code) that might otherwise be intermixed. This allows programmers to independently test the networked components of their system.

3.4 Error detection and handling

Waldo's error handling system allows programmers to recover from or propagate errors atomically. The system in Waldo is based on a simple exception handling model with termination semantics. Exceptions are detected, raised by the runtime, and propagated back up the call stack where the exceptions can be handled if the throwing code is nested within a try-catch block. Error handling in Waldo is similar to error handling in other languages with one key distinction: the call stack may be distributed over many hosts. When an exception is raised on one endpoint, Waldo automatically serializes and sends it to its calling endpoint or service, which may handle the exception or continue propagating it back.

In languages without atomicity guarantees, if an error is not explicitly handled by the programmer then it may result in the corruption of variables or state, making it difficult or impossible for the program to continue running correctly. When an error is left unhandled in Waldo, the runtime backs out of the transaction, and thus no state within Waldo is affected.

Waldo makes a distinction between two classes of exceptions: `ApplicationExceptions` and `NetworkExceptions`. `ApplicationExceptions` are automatically raised by the emitted Waldo code during the execution of an event. For example, division by zero results in an `ApplicationException`.

```
Service LogManager {
  List(element: Endpoint) all_endpoints;
  List (element: Text) log;

  Public Function root_add_log_entry(Text entry)
    returns TrueFalse {
      Number num_endpts_responded = 0;
      for (Endpoint endpt in all_endpoints) {
        try {
              endpt.add_log_entry(entry);
              num_endpts_responded += 1;
        } catch(NetworkException nex) {}
      }

      Number majority = len(all_endpoints)/2;
      if (num_endpts_responded >= majority) {
         log.append(entry);
         return True;
      }
      abort();
  }

  Public Function leaf_add_log_entry(Text entry) {
      log.append(entry);
  }
}
```

Figure 2: Service for replicated logging.

Using a configurable TCP heartbeat between remote partner endpoints, the Waldo runtime actively detects `NetworkExceptions`. Once an endpoint detects that its partner has become unreachable, it raises a `NetworkException` in any events currently waiting on a response from, or sending a message to, the partner endpoint. Further, once an endpoint's partner becomes unreachable, any new event which attempts to contact the partner will raise a `NetworkException`.

3.5 Log example

The previous publish-subscribe example illustrates Waldo's core features, but little of their motivation or utility. To do so, consider instead extending this simple example from a 1) *single* publisher writing to a log on 2) *all* its associated clients to an example in which 1) *multiple* publishers maintain a consistent, replicated log over 2) a *majority* of themselves.

Such an abstraction could be useful for an application-level game to track players' health or in-game account balances. This problem is considered challenging [35]. Well-known core systems, such as Chubby [7] rely on majority-endorsed, consistent replicated logs for fault tolerance, usually using Paxos [31] to implement them.

Figure 2 shows the core Waldo service code to build this. The core endpoint code is shorter, containing a single `Public` method that calls a sequence, which simply calls back into `LogManager`'s `leaf_add_log_entry`. An extended version of both code listings, which also handles bootstrapping and nodes' rejoining the network and fast-forwarding their logs is available online at `http://bcoli.stanford.edu/waldo_examples/`.

Each server holding a copy of the replicated log runs a `LogManager` service. To add an entry to a majority of all `LogManagers' logs`, a programmer calls `root_add_log_entry` (defined on line 5 of Figure 2) on a `LogManager` instance. This method iterates over all endpoints, requesting each to update its replicated version of the log via an endpoint call. If an endpoint's host has crashed, the endpoint call should throw a `NetworkException`, which is caught and ignored (line 12). If an endpoint crashes after its endpoint call has been invoked, but before the transaction completes, the Waldo runtime automatically backs out the transaction across all endpoints and passes a network exception back to the non-Waldo code calling `root_add_log_entry`. If a majority of nodes are able to append the update, the `LogManager` appends to its own log and returns `True`. Otherwise, it backs out all changes in its transaction.

3.6 TLS-like handshake example

Sequences are intended to encourage designing protocols composed of linear, non-branching exchanges, making endpoints' interactions easier to reason about. Figure 3 shows an example TLS-like handshake to negotiate and initialize a cipher between two endpoints to illustrate this. Instead of writing such an exchange as separate client and server handlers specified in distinct files, using sequences, such an exchange appears as straight-line code operating on shared data.

```
Sequence Negotiation ()
    returns TrueFalse cipher_initialized {
  List<Text> srvr_ciphers;
  Text sel_cipher, clnt_cipher_info;

  Server.get_available_ciphers {
    srvr_ciphers = available_ciphers.keys();
  }
  Client.select_cipher {
     cipher_initialized = False;
     Text accptbl_cipher;
     for (accptbl_cipher in cipher_priorities) {
        if (accptbl_cipher in srvr_ciphers) {
            cipher_initialized = True;
            sel_cipher = accptbl_cipher;
            break;
        }
     }
     if (not cipher_initialized)
        return;
  }
  Server.cipher_init_info {
     srvr_cipher_info = cipher_info(sel_cipher);
     local_cipher_info = srvr_cipher_info;
  }
  Client.recv_server_info_and_init {
     clnt_cipher_info = cipher_info(sel_cipher);
     local_cipher_info = client_cipher_info;
  }
  Server.recv_client_info {
     partner_cipher_info = clnt_cipher_info;
  }
}
```

Figure 3: Waldo source file for a cipher negotiation. Endpoints and their associated state (*e.g.*, available_ciphers) and methods (*e.g.*, cipher_info) are elided.

4 Transaction Scheduling Algorithm

Transactions are at the core of Waldo's event and memory model. A common way to provide transactions is through a centralized service, such as a database or a memory manager such as Sinfonia [1]. For Waldo to follow such a model, each endpoint would communicate with the centralized service to read, write, and commit state changes. There are three major problems with a centralized approach. First, one must set up and manage a centralized service, whose failure will halt all execution. Second, programs must interact with the service over the network to modify local transactional state, leading to an extra round-trip time. Third, scaling such an approach requires sharding or scaling this database, even if most transactions are just between pairs of connected hosts.

For these reasons, Waldo eschews a centralized service and instead executes distributed transactions. These transactions can modify state on multiple hosts that are only transitively connected (i.e., $A \leftrightarrow B$ and $B \leftrightarrow C$ but $A \not\leftrightarrow C$). Distributed transactions mean that a programmer does not have to set up a centralized database and a database failure will not halt the system. Second, an endpoint or service can modify its local state without needing to communicate those modifications across the network. Third, such an approach scales: the amount of processing any given host performs depends on the number of endpoints involved in its transactions, not the total number of endpoints in the system.

The major drawback of distributed transactions is that wide area latencies, combined with isolation and atomicity, can lead to transaction rates too small for low-level or high performance systems. Waldo, however, targets application-level networking. For many applications, such as updating a video game character heath or reconfiguring an edge network, even a few tens of transactions per second is sufficient.

4.1 Consistency and Execution Model

Waldo provides a strictly serializable consistency model. All reads and writes within a transaction appear as though they were completed in order. These reads and writes operate as if there were no other transactions executing at the same time. Like any other transaction processing system, the fundamental question the Waldo runtime must answer is: *when two or more transactions want to operate on the same piece of data, what happens?* To provide a useful abstraction for application networking, Waldo's transaction execution behavior focuses on three goals:

Meet networking assumptions — Networking protocols and systems generally assume some level of fairness, such that everyone competing for a shared resource receives a non-zero share of it. For example, many competing flows each receive some share of a link, and many competing web clients receive some share of its processing. While this might seem like an obvious design goal, it is not traditionally a consideration in transaction processing algorithms, which sacrifice fairness for performance. In this way, networking introduces a novel requirement for a transaction processing algorithm.

Support parallelism — A host might run multiple applications in parallel, or even multiple instances of an application (e.g. BitTorrent swarms). While an individual application might require only a few hundred transactions per second, Waldo should not limit its aggregate system performance to these expectations. Transactions that can run safely in parallel should do so.

Execute efficiently — On one hand, application networking often does not need high performance. However, as applications often run on battery-powered devices (laptops, phones, tablets), it is important that the code waste neither CPU cycles nor network capacity.

4.2 Candidate approaches

Transaction processing systems are generally structured in two parts: 1) some algorithm schedules running transactions on existing resources and 2) the system performs a two-phase commit for committing transactions. This section briefly overviews common scheduling approaches, describing why they do not meet one or more of the requirements above. Readers familiar with (or uninterested in) transaction processing implementations may wish to skip to Section 4.3, which describes Waldo's scheduling algorithm. Section 5.1 explains how this algorithm is integrated with two-phase commit in Waldo.

4.2.1 Static program analysis

Database query optimizers [26] and systems such as Periscope [22] examine program text to schedule distributed tasks. The basic drawback in such approaches is that their precision is limited to what static program analysis can provide. In the case of data dependent accesses (such as looking up in a hashtable with a key from a data load), the systems need to be conservative and so limit parallelism. In the case of Waldo, for example, a sequence that transfers a file passed as a parameter to a sequence would need to read-lock the accessible file system. It therefore greatly limits parallelism.

4.2.2 Optimistic Concurrency Control

Instead of acquiring object read and write locks *before* executing a transaction, optimistic concurrency control and commit-time validation execute transactions and take locks only at commit time. If the system detects that conflicting writes have been committed, it rolls back the operation and tries again. Haskell's atomic statements, for example, use this approach by having each object maintain a monotonically-increasing version number [23]. The key drawback of these approaches is that they permit starvation. Repeated short transactions can starve a long-running transaction which always sees values have changed when it tries to commit. In the presence of high contention, these approaches do not provide the semantics and fairness that networking systems and protocols expect.

4.2.3 Wound/wait

Static program analysis locks objects before a transaction. Optimistic concurrency control locks them at the end of a transaction. A final alternative acquires read and write locks on objects as a transaction progresses. The primary challenge with this approach is deadlock. This third approach acquires locks in the order in which a transaction encounters them, and so deadlock can occur. In these cases, the system detects the deadlock and breaks it by aborting one of the transactions.

Wound/wait [42] is a standard database algorithm for breaking deadlocks.[2] Wound/wait assigns every transaction a timestamp. If an older transaction tries to take a lock held by a younger one, the younger transaction rolls back (it is wounded). If a younger transaction tries to take a lock held by an older one, it waits. Because time stamps are transitive, wound/wait breaks waits-for cycles and therefore prevents deadlock. Furthermore, because the oldest transaction in the system will always roll back other transactions, no transaction will starve indefinite.

Wound/wait gives no fairness guarantees. If one client issues one million transactions at time t, then another client issuing a transaction at time $t+1$ must wait for those million transactions to complete.

4.2.4 Fair queueing

Can a transaction scheduler provide fairness by applying fair queueing algorithms [16]? For example, one could assign timestamps in wound/wait based not only on the arrival time of a transaction but also when the last transaction from that client completed. A modified version of wound/wait using these timestamps would then allow

[2] Wait/die is another common algorithm. Wait/die may rollback more transactions, but because it does so earlier than wound/wait, these rollbacks may be less expensive [46].

transactions from endpoints with fewer transactions executed to rollback transactions from other endpoints.

Unfortunately, fair queueing algorithms do not fit this problem well. Fair queueing is designed to fairly share a serialized resource, such as a communication channel. It therefore does not satisfy the parallelism requirement. For example, suppose that an endpoint submits 50 transactions that can execute in parallel: how should timestamps be assigned to them? Assigning them the identical timestamp can result in poor fairness as other transactions have to wait for all 50 to commit. Furthermore, the timestamp of each transaction needs to be consistent across all endpoints that might execute it.

4.3 Waldo algorithm

To provide fairness while simultaneously allowing parallelism, Waldo uses a novel transaction scheduling algorithm consisting of four parts:

1. A "standard" transaction's timestamp is the time it was created in terms of a root endpoint's local Lamport clock [30]. Potential deadlocks between standard transactions are avoided with wound/wait, such that an older standard transaction preempts a younger standard transaction.

2. Each endpoint assigns its oldest outstanding transaction to be its "primary" transaction. The timestamp of a primary transaction is the last local time a primary transaction rooted at the endpoint committed.

3. A primary transaction always preempts a standard transaction, regardless of timestamp.

4. After a primary transaction completes, a Waldo endpoint promotes its oldest standard transaction to be its primary transaction. This may involve sending messages across the network.

The end result is that Waldo uses a fairness algorithm (in this case, round-robin) to resolve conflicts between primary transactions, but allows other transactions to resolve conflicts using wound/wait. This provides each endpoint a minimum quality of service, satisfying the fairness requirement. It also allows many non-conflicting transactions to execute in parallel, meeting the parallelism requirement. Finally, as the evaluation in Section 6 shows, the algorithm itself is efficient, such that it does not consume many CPU cycles or send much additional state.

4.3.1 Algorithm analysis

Because endpoints take turns having the oldest primary transaction, Waldo's primary transactions provide a minimum quality of service for each endpoint. However, commits across standard transactions can still be unfair. This section examines the worst case fairness bounds for Waldo's scheduling algorithm and shows it is starvation-free.

Consider an idealized network of E endpoints with O outstanding transactions per endpoint at some time t_0, and define T as the longest start-to-finish time for a transaction, absent lock contention. After an endpoint commits a primary transaction, its subsequent primary transaction is given the lowest priority of primary transactions. It must therefore wait at most $(E-1)T$ to start executing. Therefore, each endpoint with an outstanding transaction will commit at least every ET seconds.

Because transactions are made primary by age, no transaction in the system submitted at t_0 waits longer than $(O-1)ET$ to execute. This bound is guaranteed regardless of how many additional transactions are added after t_0. Further, because transactions that do not conflict with any other transactions are neither blocked nor preempted, this algorithm ensures parallelism (when possible).

A fairness bound in the absence of lock contention is not meaningful: if endpoint A submits a series of transactions that must run serially, but endpoint B submits 100,000 transactions that do not conflict with each other or A's transactions, B should commit more transactions than A. Limiting the ratio of their commit rates to ensure fairness needlessly decreases utilization. In some ways, this problem is similar to recent work on multi-resource fairness [18], except that those algorithms require predeclaring all necessary resources.

The worst case fairness across conflicting transactions, in terms of transaction execution time, occurs when one endpoint dominates all standard transactions: all endpoints still receive a fair share of primary transaction stream, but only one commits standard transactions. In this case, if the fair share of transaction execution time is $\frac{1}{E}$, the worst case fairness is that one endpoint receive $\frac{E+1}{2E}$ and all other endpoints receive $\frac{1}{2E}$.

5 Implementation and extensions

Waldo compiles to both Python and Java. Excluding automatically generated code, unit tests, and external libraries, the Waldo compiler and runtime libraries consist of 22,500 lines of code, measured using SLOCcount [47]. This section describes several features and details of their implementation.

5.1 Two-phase commit

Waldo provides consistency, atomicity, and isolation for its transactions using two mechanisms. The first is through the algorithm described in Section 4.3, which schedules lock acquisition across *executing* transactions. The second is a standard two-phase commit protocol, across all state touched by a *committing* transaction. Between the first and second phases of its commit, regardless of priority, no other transaction may preempt it.

This adds some complexity to the implementation: when a primary transaction attempts to write to an object read locked by many standard transactions, it must first atomically ensure that all readers have not yet reached their commit phase before preempting them.

5.2 Message format and compression

Waldo's sequences abstract away the notion of messages. However, to provide this abstraction, the underlying runtime still exchanges a host of messages between communicating endpoints.

Waldo's messages are specified using protocol buffers [19]. Both the Java and Python Waldo implementations use the code autogenerated from these specifications to serialize and deserialize messages between endpoints. Unlike the ad-hoc string-based solutions we observed programmers employing in the studies in Section 2, protocol buffers' serialization methods automatically compresses data sent between endpoints, potentially saving bandwidth. Additionally, Waldo only transmits dirty data: if a programmer does not write to a shared variable, the runtime does not transmit it. Finally, Waldo only transmits variable deltas: if a programmer writes to a single element in a shared map, the runtime only sends data relevant for that single element.

5.3 Security

Unlike applications running on a single host, distributed applications may transmit data over networks where adversaries can listen and inject packets. There is a substantial body of work devoted to providing high-level confidentiality, integrity, and authentication guarantees in such environments. Waldo's goal is not to provide any new primitives on top of this work, but rather to incorporate it in a way that makes it as *usable* as possible.

By default, Waldo applications run over SSL with self-signed certificates. Further, just a single Waldo API call allows a developer to instantiate a certificate authority, which they can interface to as if it were a general Waldo service to add authentication to their projects.

6 Evaluation

The core claims of this paper are:

1. Waldo and its consistency provide atomicity, consistency, and isolation at a reasonable transaction rate;
2. Leveraging transactional semantics may simplify many networked applications; and
3. The abstractions that Waldo provides are flexible enough to support such applications.

We demonstrate the first claim through a series of microbenchmarks and the following two by building several applications using Waldo, and describing them.

6.1 Microbenchmarks

Waldo is not targeted for all networked applications. Applications that require ultra-low latency or ultra-high throughput should use alternate tools. The goal of this section is to demonstrate that Waldo performs reasonably well enough to make it usable for many basic desktop applications, mobile games, etc. To this end, we measure the latency, throughput, and bandwidth consumption for a pair of endpoints connected over TCP, with its no delay flag set. All numbers are from the Java implementation of Waldo, running with encryption off. To simulate the effect of a long-running connection and reduce the bottleneck of slow start, we take our measurements after a warmup period.

Recognizing the diversity of networked applications, we present benchmarks for four target environments: a Nexus 4 mobile phone, an Amazon Kindle Fire HD, a Macbook Pro, and a Dell Studio XPS connected over a wide area network to an EC2 node with an average 74.7 ms round trip ping time. The specifications for each are shown in Table 1.

For all tests, endpoints run a simple two step sequence that touches no endpoint data and contains no sequence data. We turn runtime optimizations off to ensure that, despite not touching any state, both endpoints still go through a full two-phase commit. To characterize implementation overhead and provide a best-case throughput and latency bound, we benchmark the phone, Kindle, and Macbook by running each endpoint locally (connected via TCP). The last condition, between the Dell desktop and EC2 node runs over the wide area network.

6.1.1 Latency

To measure latency, we run 3000 transactions serially for each condition, except the local Macbook (because transactions complete more quickly on the Macbook, we

Machine	OS	Processor	RAM
Nexus 4	Android 4.3	Krait Quad Core 1500 MHz	2GB
Kindle Fire HD	Android 4.0.3	Dual Core TI OMAP4 1.5GHz	1GB
Macbook Pro	OSX 10.8.4	Intel Core i7 2.4GHz	16GB
Dell XPS 8100	Ubuntu 12.04	Intel Core i7 2.8GHz	8GB
EC2 m1.xlarge	Ubuntu 12.04	Intel Xeon E5-2650 2.00GHz	15GB

Table 1: Hardware specifications for conditions.

Machine	Min (ms)	Max (ms)	Avg (ms)
Phone	6.8264	7.8196	7.5118
Kindle	4.7254	5.0296	4.8982
Macbook	0.17309	0.17520	0.17437
WAN	150.23	150.68	150.47

Table 2: Latency microbenchmarks, across ten runs for each condition.

run 30,000 transactions serially), and compute the time it takes to complete a single transaction as the average of the total time it takes to complete a single transaction.

Table 2 shows the results of these experiments. Transactions in all conditions take fractions of a second to process. Recalling the 74.7 ms ping round trip time mentioned above, as expected, network latency dominates the transaction execution time for the WAN condition.

The phone and Kindle take more than an order of magnitude longer to process a transaction than the Macbook. Running Waldo in a profiler on the Macbook demonstrates that this slowdown is likely not due to memory backpressure: across 30,000 transactions, resident memory never exceeds 95MB. This suggests that the slowdown is likely attributable to 1) different JVM and OS characteristics and/or 2) fundamental processor differences. We cannot interchange the JVM and OS of a Kindle and a Macbook to directly quantify these differences. However, running an older Macbook with OSX 10.6.8 and Java 1.6.033 installed through OSX's default software update mechanism ran approximately 30% faster than running the identical benchmark on the same hardware running Ubuntu 12.04 and openjdk-6.

6.1.2 Throughput

Table 3 shows throughput results for each condition, across ten runs. For the phone, laptop, and WAN cases, in order to saturate throughput, application code shares a Waldo endpoint object between many threads that invoke it. In the Kindle's case, running additional threads decreases aggregate throughput, perhaps from the overhead of context switching, and we only run one invoking thread. The WAN condition shows slightly higher variation than the other conditions, likely due to network effects.

Machine	Min (tps)	Max (tps)	Avg (tps)
Phone	183.54	190.03	185.93
Kindle	198.82	211.62	204.24
Macbook	12,938	13,325	13,105
WAN	9,430.5	10,420	10,047

Table 3: Throughput benchmarks, across ten runs for each condition. Results are in transactions per second (tps).

6.1.3 Bandwidth

To perform 3000 transactions across the WAN, Waldo sends approximately 25,000 packets with a cumulative payload of 1.5MB, including ACKs and heartbeat messages. Amortized, this corresponds to roughly 500B of data per transaction. The vast majority of these data (70%) are from transmitting long event UUID strings between endpoints, which are used to map incoming messages to the transaction that they are intended for. We are exploring alternate encodings for these data.

6.2 Applications

The replicated server log example in Section 3.5 provides a focused example of Waldo's potential utility. To evaluate Waldo's effectiveness in complete end-to-end applications, we built several network applications. These include a distributed hash table (DHT), document server, distributed bank, leader election system, highscore scoreboard for a mobile game, a cloud image processing system for biomedical data, a network configuration system, and a game with game lobby. For brevity, we only focus on the last two of these, but note that for very simple client-server oriented applications, such as the highscore scoreboard, Waldo's primary utility was in providing an RPC frontend on a finely-grained synchronized data structure.

6.2.1 WaldoConf

In an effort to test Waldo on real-world problems, we asked a network administrator at our host institution about his day-to-day challenges. He reported that scripting an update for virtual LAN settings across dozens of switches took over forty hours of work. Much of this time was spent physically restarting nodes that got into strange states when scripts failed [14].

To explore these problems, we built WaldoConf, a network administrative application. Without the ability to modify network device firmware directly, WaldoConf instead runs a single service per device on separate hosts and extend the service's internal state so that on commit, WaldoConf pushes changes via SNMP [8] to the associated network device. Using Waldo's distributed transactions, either all network devices transition to updated state, or none do. Additionally, we extended WaldoConf to perform trial transactions. Using trial transactions, each WaldoConf service pushes changes to network devices and automatically roll them back after a pre-specified period of time. Because configuration changes can affect the connection between a device and its controlling WaldoConf service, WaldoConf does not eradicate the need to manually reset devices, but may reduce it.

6.2.2 Anagrams and game lobby

Game play for Anagrams is simple and mimics popular mobile word scramble games [4, 6]. Several players are presented with the same set of letters. During a set time interval, each player constructs as many words as possible from the set of letters, and is assigned a score for each word based on 1) the length of the word and 2) whether the player is the first, second, third, etc. to have found the word in the scrambled letters.

Anagrams leverages the fairness and starvation guarantees of Waldo's scheduling algorithm. In Anagrams, even if all a player's opponents collude, attaching automated robots that perform literal dictionary attacks (submitting guesses for every word in an English dictionary to the game server), the player still receives a meaningful quality of service to submit his/her own found words.

As initially structured, Anagrams focused on steady-state operation: how to receive, process, and score guesses. But it ignored edge cases associated with starting and stopping the game. Players could join after a game began and submit guesses after a game ended; once a game service started, it persisted in memory, even after the game ended.

Extending Anagrams with a game lobby[3] was simple using Waldo's distributed transactions. A lobby service runs on a server, holding references to endpoints connected to players. When a new player joins, the player registers with the lobby service. As part of this registration, the lobby service checks if there are enough waiting players to begin a new game. If there are, the lobby creates a new Anagram service, composed of the new players.

Either all players begin a game or none do. On error when starting a new game, Waldo automatically resets affected state (*e.g.*, each player's list of opponents) as well as cleaning up state associated with the new Anagram service on the game server. Similarly using transactions, the game is either atomically, across all players, in a started state or in a completed state.

7 Related Work

This paper touches on two substantial bodies of work, network programming and transaction processing. We start by highlighting several canonical systems and languages at the intersection of these topics and then expand outwards.

7.1 Programming with transactions

Recent experiments demonstrate that software transactional memory on single node systems can speed development time for applications that do not require high performance and anecdotally reinforce many of the observations presented in Section 2 [37].

Languages such as Haskell [23] and Clojure [10] provide explicit linguistic support for software transactions on single nodes. Additionally, experimental compilers that support software transactional memory on a single node for other languages such as Python [38] and C++ [34] are also being considered or complete.

Argus first proposed a limited form of *distributed* transactions over 30 years ago [32], which provided atomicity, consistency, and durability, but required users to manually lock state for isolation. Similarly, Orca [5] allowed programmers to make transactional changes to shared, synchronized data objects, but disallowed nested transactions across two or more such objects.

More modern languages also are incorporating or have incorporated some form of distributed transactions. Chapel, a language developed by Cray, Inc. for the high-performance computing community, is considering adding support for distributed transactions [15], however, its current reference manual does not include any information about them [12].

Immutant [25] an application server framework for Clojure recently incorporated the Open XA standard [21] to perform distributed transactions. Java's transaction API [36] also uses Open XA to commit across multiple hosts. Unlike Waldo's scheduling algorithm, Open XA provides no client starvation or fairness guarantees.

[3] Game lobbies assemble a quorum of players and automatically spawn a new game populated with them.

7.2 Transaction and resource scheduling

Section 4.2 already described directly related approaches for scheduling transactions. Waldo's use of Lamport clocks for its primary transactions mirrors Rajwar and Goodman's similar use with wound/wait for transaction processing on a single multi-processor node [39].

Ghodsi *et al.* propose the Dominant Resource Fairness algorithm (DRF) [18] to schedule competing cluster jobs while providing attractive system-wide guarantees (*e.g.*, Pareto optimality, strategy-proofness, etc.). Like DRF, Waldo's scheduling algorithm mediates access to many different resources (individual Waldo objects to operate on) across multiple actors. Unlike DRF, the Waldo's algorithm is distributed: no node has a view of all resources, nor of all jobs in the system. Additionally, primary transactions are not required to pre-declare all resources they require to execute.

Waldo provides transactional access across distributed memory. Most work from the distributed software transaction community (*e.g.*, [11, 29]) focuses on distributed transactional memory replicated at each node in the network, and attempts to manage consistency across replicas.

7.3 Network programming

Thrift [44], CORBA's object request brokers [43], and Java's RMI API [17] are all examples of systems that allow a local program to execute a procedure on a remote host. Like Waldo's sequences, these simplify data serialization and message dispatch. However, RPC frameworks are isolated from application data.

Actor-model based languages, such as Erlang [2] and E [40], map naturally into a distributed programming paradigm. They enforce message passing share-nothing architectures, simplifying error recovery [3]. However, without higher-level primitives for coordination programmers must build their own state consistency protocols and reason about edge cases caused by event interleaving on their own.

MACE [28] addresses the challenge of edge cases by making a protocol's state space more visible. A MACE programmer explicitly specifies a state machine for his/her protocol, and may use additional tools to validate it [27]. A MACE programmer still explicitly manages state him/herself and audit it to ensure it will not lead to read-write conflicts. Like MACE, Waldo transcompiles the networking components of an application to that application's target language.

8 Discussion and conclusion

The challenges identified in Section 2 arise because the tools that subjects used were designed to support an execution model that strictly isolates data and logic between separate hosts. Such a programming model can create systems that are powerful, scalable, and fast. But it also requires a level of discipline that can be a tremendous impediment.

This paper argues instead that such problems can be avoided or eased with a better abstraction — transactions. Its core contributions supporting this thesis are a novel distributed scheduling algorithm and Waldo, a domain specific language composed of services, endpoints, and sequences.

Because sequences are always between a pair of endpoints, more complex control structures across a single sequence are almost never needed. The one exception is that very occasionally a protocol may want a form of conditional loop. We have found that in practice, the common needs for such control flows, such as send-N-packets, do not appear in Waldo as it automatically handles such fragmentation. In the case of sending a large block of data, a sequence simply assigns the data to a shared variable. When the next sequence step executes on the other endpoint, the data will have arrived. More complex cases, such as tree iterations (e.g., DNS) or conditional operations (e.g., HTTP redirects) can be handled as multiple sequences through a service.

Such approaches work reasonably when each sequence depends on the result of a previous sequence, but can needlessly slow a transaction in cases where a programmer initiates sequences across otherwise isolated endpoint pairs. For instance, in the publish-subscribe example described in Section 3, it is unnecessary for the publisher service to wait for one endpoint's `Push` sequence to complete before starting another's.

Ongoing and future work on Waldo explores this issue: we are currently considering adding runtime support for optimistic parallelization within a single transaction. Using such a mechanism, the Waldo runtime decides which code within a transaction to attempt to execute in parallel based on program analysis and current resource utilization. Upon execution, if the runtime detects that its schedule violates causality, it automatically rollsback the change, thereby preserving Waldo's otherwise linear interface for the programmer.

In addition to optimistic parallelization, we are also actively trying to improve Waldo's interface and tools for writing secure applications. This branch of research asks such basic questions as, *should a programmer need to know what a key is to maintain integrity in his/her application?* and *what would the Waldo runtime look like if two, connected endpoints do not trust each other?*

References

[1] Aguilera, M., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 159–174.

[2] Armstrong, J. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] Armstrong, J. Erlang. *Commun. ACM 53*, 9 (Sept. 2010), 68–75.

[4] Arts, E. Boggle. https://play.google.com/store/apps/details?id=com.ea.boggle_na&hl=en.

[5] Bal, H., Kaashoek, M., and Tanenbaum, A. Orca: A language for parallel programming of distributed systems. *Software Engineering, IEEE Transactions on 18*, 3 (1992), 190–205.

[6] BoYan_Soft. Anagram. https://play.google.com/store/apps/details?id=com.bogmix.anagramen&hl=en.

[7] Burrows, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 335–350.

[8] Case, J., Fedor, M., Schoffstall, M., and Davin, J. Rfc 1157: Simple network management protocol (snmp). *IETF, April* (1990).

[9] Cheslack-Postava, E., Azim, T., Mistree, B., Horn, D., Terrace, J., Levis, P., and Freedman, M. A scalable server for 3d metaverses. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX* (2012), vol. 12.

[10] http://clojure.org/refs.

[11] Couceiro, M., Romano, P., Carvalho, N., and Rodrigues, L. D2stm: Dependable distributed software transactional memory. In *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on* (2009), IEEE.

[12] Cray, I. Chapel specification version 0.93. http://chapel.cray.com/language.html.

[13] Dabek, F., Brunskill, E., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I., and Balakrishnan, H. Building peer-to-peer systems with chord, a distributed lookup service. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on* (2001), IEEE, pp. 81–86.

[14] Davis, M. Personal interview.

[15] Deitz, S., Choi, S., and Iten, D. Five powerful chapel idioms. *CUG2010* (2010).

[16] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review* (1989), vol. 19, ACM, pp. 1–12.

[17] Downing, T. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc., 1998.

[18] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: fair allocation of multiple resource types. In *USENIX NSDI* (2011).

[19] Google. https://developers.google.com/protocol-buffers/.

[20] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: portable parallel programming with the message passing interface*, vol. 1. MIT press, 1999.

[21] Group, T. O. Distributed transaction processing: the xa specification. http://www.opengroup.org/onlinepubs/009680699/toc.pdf.

[22] Guo, Z., Fan, X., Chen, R., Zhang, J., Zhou, H., McDirmid, S., Liu, C., Lin, W., Zhou, J., and Zhou, L. Spotting code optimizations in data-parallel pipelines through periscope. In *Proc. of the 10th USENIX conference on Operating systems design and implementation, OSDI* (2012).

[23] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2005), PPoPP '05, ACM, pp. 48–60.

[24] IGG.COM. Texas holdem poker deluxe. https://play.google.com/store/apps/details?id=com.igg.pokerdeluxe.

[25] Immutant.org. Immutant. http://immutant.org/.

[26] Jarke, M., and Koch, J. Query optimization in database systems. *ACM Computing Surveys 16* (1984), 111–152.

[27] Killian, C., Anderson, J., Jhala, R., and Vahdat, A. Life, death, and the critical transition: Finding liveness bugs in systems code. *Networked Systems Design and Implementation* (2007).

[28] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 179–188.

[29] Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., and Watson, I. Distm: A software transactional memory framework for clusters. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on* (2008), IEEE, pp. 51–58.

[30] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[31] Lamport, L. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[32] Liskov, B. Distributed programming in argus. *Commun. ACM 31*, 3 (Mar. 1988), 300–312.

[33] Mistree, B., Chandra, B., Cheslack-Postava, E., Levis, P., and Gay, D. Emerson: Accessible scripting for applications in an extensible virtual world. In *Proc. of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (2011), ACM.

[34] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., et al. Design and implementation of transactional constructs for c/c++. In *ACM Sigplan Notices* (2008), vol. 43, ACM, pp. 195–212.

[35] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm.

[36] Oracle, I. Java transaction api (jta). http://www.oracle.com/technetwork/java/javaee/jta/index.html.

[37] Pankratius, V., and Adl-Tabatabai, A.-R. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures* (2011), ACM, pp. 43–52.

[38] Pypy. Call for donations - transactional memory / automatic mutual exclusion in pypy. http://pypy.org/tmdonate.html.

[39] Rajwar, R., and Goodman, J. R. Transactional lock-free execution of lock-based programs. In *ACM SIGOPS Operating Systems Review* (2002), vol. 36, ACM, pp. 5–17.

[40] Richardson, J., Carey, M., and Schuh, D. The design of the e programming language. *ACM Transactions on Programming Languages and Systems (TOPLAS) 15*, 3 (1993), 494–534.

[41] Rodriguez, A., Killian, C., Bhat, S., Kostić, D., and Vahdat, A. Macedon: methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), USENIX Association, pp. 20–20.

[42] Rosenkrantz, D. J., Stearns, R. E., and Lewis, II, P. M. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.

[43] Schmidt, D. C., and Kuhns, F. An overview of the real-time CORBA specification. *Computer 33*, 6 (2000).

[44] Slee, M., Agarwal, A., and Kwiatkowski, M. Thrift: Scalable cross-language services implementation. *Facebook, Jan* (2007).

[45] Smule. Magic piano. `https://play.google.com/store/apps/details?id=com.smule.magicpiano&hl=en`.

[46] Ullman, J. D., Garcia-Molina, H., and Widom, J. *Database systems: the complete book*. Prentice Hall Upper Saddle River, 2001.

[47] Wheeler, D. SLOCcount. `http://www.dwheeler.com/sloccount/`, 2009.