# Fabric: A Platform for Secure Distributed Computation and Storage

Jed Liu Michael D. George K. Vikram
Xin Qi Lucas Waye Andrew C. Myers
{liujed,mdgeorge,kvikram,qixin,lrw,andru}@cs.cornell.edu
Department of Computer Science
Cornell University
4130 Upson Hall, Ithaca NY

## Abstract

Fabric is a new system and language for building secure distributed information systems. It is a decentralized system that allows heterogeneous network nodes to securely share both information and computation resources despite mutual distrust. Its high-level programming language makes distribution and persistence largely transparent to programmers. Fabric supports data-shipping and function-shipping styles of computation: both computation and information can move between nodes to meet security requirements or to improve performance. Fabric provides a rich, Java-like object model, but data resources are labeled with confidentiality and integrity policies that are enforced through a combination of compile-time and run-time mechanisms. Optimistic, nested transactions ensure consistency across all objects and nodes. A peer-to-peer dissemination layer helps to increase availability and to balance load. Results from applications built using Fabric suggest that Fabric has a clean, concise programming model, offers good performance, and enforces security.

## Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.4.6 [**Operating Systems**]: Security and Protection—*information flow controls*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

## General Terms

Security, Languages

## 1. Introduction

We rely on complex, distributed information systems for many important activities. Government agencies, banks, hospitals, schools, and many other enterprises use distributed information systems to manage information and interact with the public. Current practice does not offer general, principled techniques for implementing the functionality of these systems while also satisfying their security and privacy requirements. This lack motivates the creation of Fabric, a platform for building secure distributed information systems.

It is particularly difficult to build secure *federated* systems, which integrate information and computation from independent administrative domains—each domain has policies for security and privacy, but does not fully trust other domains to enforce them. Integrating information from different domains is important because it enables new services and capabilities.

To illustrate the challenges, consider the scenario of two medical institutions that want to securely and quickly share patient information. This goal is important: according to a 1999 Institute of Medicine study, at least 44,000 deaths annually result from medical errors, with incomplete patient information identified as a leading cause [25]. However, automated sharing of patient data poses difficulties. First, the security and privacy policies of the two institutions must be satisfied (as mandated by HIPAA [22] in the U.S.), restricting which information can be shared or modified by the two institutions. Second, a patient record may be updated by both institutions as treatment progresses, yet the record should be consistent and up to date when viewed from the two institutions. It is inadequate to simply transmit a copy of the record in a common format such as XML, because the copy and the original are likely to diverge over time. Instead, both institutions should have easy, secure, consistent, and efficient access to what is logically a single patient record.

Scenarios like this one inspire the development of Fabric, a federated system that supports secure, shared access to information and computation, despite distrust between cooperating entities. The goal of Fabric is to make secure distributed applications much easier to develop, and to enable the secure integration of information systems controlled by different organizations.

To achieve this goal, Fabric provides a shared computational and storage substrate implemented by an essentially unbounded number of Internet hosts. As with the Web, there is no notion of an "instance" of Fabric. Two previously noninteracting sets of Fabric nodes can interact and share information without prior arrangement. There is no centralized control over admission: new nodes, even untrustworthy nodes, can join the system freely.

Untrustworthy nodes pose a challenge for security. The guiding principle for security in Fabric is that one's security should never depend on components of the system that one does not trust. Fabric provides security assurance through a combination of mechanisms at the language and system levels.

Fabric gives programmers a high-level programming abstraction in which security policies and some distributed computing features


*SOSP'09,* October 11–14, 2009, Big Sky, Montana, USA.

are explicitly visible to the programmer. Programmers access Fabric objects in a uniform way, even though the objects may be local or remote, persistent or nonpersistent, and object references may cross between Fabric nodes.

The Fabric programming language is an extension to the Jif programming language [33, 36], in turn based on Java [50]; Fabric extends Jif with support for distributed programming and transactions. Like Jif, Fabric has several mechanisms, including access control and information flow control, to prevent untrusted nodes from violating confidentiality and integrity. All objects in Fabric are labeled with policies from the *decentralized label model* (DLM) [34], which expresses security requirements in terms of principals (e.g., users and organizations). Object labels prevent a node that is not trusted by a given principal from compromising the security policies of that principal. Therefore, Fabric has fine-grained trust management that allows principals to control to what extent other principals (and nodes) can learn about or affect their information.

To achieve good performance while enforcing security, Fabric supports both *data shipping*, in which data moves to where computation is happening, and *function shipping*, in which computations move to where data resides. Data shipping enables Fabric nodes to compute using cached copies of remote objects, with good performance when the cache is populated. Function shipping enables computations to span multiple nodes. Inconsistency is prevented by performing all object updates within transactions, which are exposed at the language level. The availability of information, and scalability of Fabric, are increased by replicating objects within a peer-to-peer *dissemination layer*.

Of course, there has been much previous work on making distributed systems both easier to build and more secure. Prior mechanisms for remotely executing code, such as CORBA [41], Java RMI [24], SOAP [52] and web services [5], generally offer only limited support for information security, consistency, and data shipping. J2EE persistence (EJB) [16] provides a limited form of transparent access to persistent objects, but does not address distrust or distributed computation. Peer-to-peer content-distribution and wide-area storage systems (e.g., [15, 19, 27, 44]) offer high data availability, but do little to ensure that data is neither leaked to nor damaged by untrusted users. Nor do they ensure consistency of mutable data. Prior distributed systems that enforce confidentiality and integrity in the presence of distrusted nodes (e.g., [57, 11, 56]) have not supported consistent computations over persistent data.

Fabric integrates many ideas from prior work, including compile-time and run-time information flow, access control, peer-to-peer replication, and optimistic transactions. This novel integration makes possible a higher-level programming model that simplifies reasoning about security and consistency. Indeed, it does not seem possible to provide a high-level programming model like that of Fabric by simply layering previous distributed systems abstractions. Several new ideas were also needed to make Fabric possible:

- A programming language that integrates information flow, persistence, transactions, and distributed computation.
- A *trust ordering* on information-flow labels, supporting reasoning about information flow in distributed systems.
- An integration of function shipping and data shipping that also enforces secure information flows within and among network nodes.
- A way to manage transactions distributed among mutually distrusting nodes, and to propagate object updates while enforcing confidentiality and integrity.

Fabric does not require application developers to abandon other standards and methodologies; it seems feasible for Fabric to interoperate with other standards. In fact, Fabric already interoperates with existing Java application code. It seems feasible to implement many existing abstractions (e.g., web services) using Fabric. Conversely, it seems feasible to implement Fabric nodes by encapsulating other services such as databases. We leave further work on interoperability to the future.

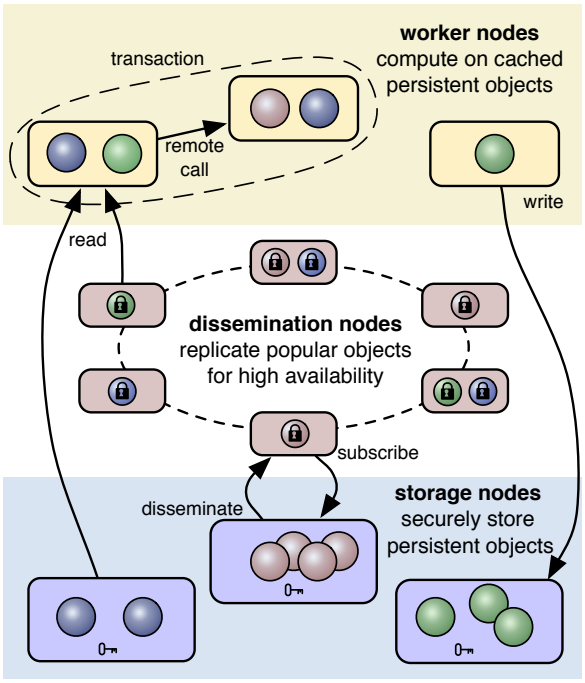


**Figure 1: Fabric architecture**

The rest of this paper describes the design and implementation of Fabric. Section 2 presents the Fabric architecture in more detail. Section 3 introduces the Fabric language. Section 4 covers cache and transaction management. Section 5 explains how multinode transactions are implemented. Some details of the Fabric implementation are given in Section 6; with the exception of certain explicitly identified features, the design described in this paper has been implemented in a prototype. Section 7 reports on our evaluation of this implementation, including results for the expressiveness of Fabric and the performance of a substantial application built using Fabric. Related work is covered in Section 8, and Section 9 concludes.

## 2. Architecture

Fabric nodes take on one of the three roles depicted in Figure 1:

- *Storage nodes* (or *stores*) store objects persistently and provide object data when requested.
- *Worker nodes* perform computation, using both their own objects and possibly copies of objects from storage nodes or other worker nodes.
- *Dissemination nodes* provide copies of objects, giving worker nodes lower-latency access and offloading work from storage nodes.

Although Fabric nodes serve these three distinct roles, a single host machine can have multiple Fabric nodes on it, typically colocated in the same Java VM. For example, a store can have a colo-

cated worker, allowing the store to invoke code at the worker with low overhead. This capability is useful, for example, when a store needs to evaluate a user-defined access control policy to decide whether an object update is allowed. It also gives the colocated worker the ability to efficiently execute queries against the store. Similarly, a worker node can be colocated with a dissemination node, making Fabric more scalable.

## 2.1 Object model

Information in Fabric is stored in objects. Fabric objects are similar to Java objects; they are typically small and can be manipulated directly at the language level. Fabric also has array objects, to support larger data aggregates. Like Java objects, Fabric objects are mutable and are equipped with a notion of identity.

**Naming.** Objects are named throughout Fabric by object identifiers (*oids*). An object identifier has two parts: a store identifier, which is a fully qualified DNS hostname, and a 64-bit object number (*onum*), which identifies the object on that host. An object identifier can be transmitted through channels external to Fabric, by writing it as a uniform resource locator (URL) with the form `fab://`$store$`/`$onum$, where $store$ is a fully qualified DNS hostname and $onum$ is the object number.

An object identifier is permanent in the sense that it continues to refer to the same object for the lifetime of that object, and Fabric nodes always can use the identifier to find the object. If an object moves to a different store, acquiring an additional oid, the original oid still works because the original store has a surrogate object containing a forwarding pointer. Path compression is used to prevent long forwarding chains.

Knowing the oid of an object gives the power to name that object, but not the power to access it: oids are not capabilities [18]. If object names were capabilities, then knowing the name of an object would confer the power to access any object reachable from it. To prevent covert channels that might arise because adversaries can see object identifiers, object numbers are generated by a cryptographically strong pseudorandom number generator. Therefore, an adversary cannot probe for the existence of a particular object, and an oid conveys no information other than the name of the node that persistently stores the object.

Fabric uses DNS to map hostnames to IP addresses, but relies on X.509 certificates to verify the identity of the named hosts and to establish secure SSL connections to them. Therefore, certificate authorities are the roots of trust and naming, as in the Web.

Fabric applications can implement their own naming schemes using Fabric objects. For example, a naming scheme based on directories and pathnames is easy to implement using a hash map.

**Labels.** Every object has an associated *label* that describes the confidentiality and integrity requirements associated with the object's data. It is used for information flow control and to control access to the object by Fabric nodes. This label is automatically computed by the Fabric run-time system based on programmer annotations and a combination of compile-time and run-time information flow analysis. Any program accepted by the Fabric type system is guaranteed to pass access control checks at stores, unless some revocation of trust has not yet propagated to the worker running it.

**Classes.** Every Fabric object, including array objects, contains the oid of its *class object*, a Fabric object representing its class in the Fabric language. The class object contains both the fully-qualified path to its class (which need not be unique across Fabric) and the SHA-256 hash of the class's bytecode (which should be globally unique). The class object creates an unforgeable binding between each object and the correct code for implementing that object. The class object can also include the actual class bytecode, or the class bytecode can be obtained through an out-of-band mechanism and then checked against the hash. When objects are received over the network, the actual hash is verified against the expected one.

**Versions.** Fabric objects can be mutable. Each object has a *current version number*, which is incremented when a transaction updates the object. The version number distinguishes current and old versions of objects. If worker nodes try to compute with out-of-date object versions, the transaction commit will fail and will be retried with the current versions. The version number is an information channel with the same confidentiality and integrity as the fields of the object; therefore, it is protected by the same mechanisms.

## 2.2 Security and assumptions

The design of Fabric is intended to allow secure sharing of computations and information, despite the presence of adversaries that control some Fabric nodes. The security of any system rests on assumptions about the threats it is designed to protect against. The assumptions that Fabric makes about adversaries are largely typical, and weak, which strengthens Fabric's security assurance.

Compromised nodes are assumed to be malicious, so they may appear to participate correctly in Fabric protocols, and to execute code correctly, while behaving maliciously. The Fabric run-time exposes some information to Fabric nodes that is not visible at the language level, such as object identifiers and version numbers; this information is visible to compromised nodes, which can attack below the language level of abstraction. Compromised nodes may misuse any information they receive, including cryptographic keys, and may supply corrupted information to other nodes. However, without the corresponding private keys, nodes are assumed not to be able to learn anything about encrypted content except its size, and cannot fake digital signatures. Fabric does not attempt to control read channels [55] at the language level. As with most work on distributed systems, timing and termination channels are ignored.

Network adversaries are assumed not to be able to read or fabricate messages. This assumption is justified in Fabric by using SSL for all network communication. However, an adversary might still learn information about what is happening in Fabric from the size, existence, or timing of network messages. As in other work on distributed systems, these covert channels are ignored. A network adversary can also prevent message delivery. The availability of services written using Fabric therefore depends on an assumption that the network delivers messages eventually.

Fabric users are able to express partial or complete trust in Fabric nodes. Therefore, statements of trust are security assumptions of the users expressing trust. If a user expresses trust in a node, the compromise of that node might harm the security of that user. In fact, the degree of trust expressed bounds the degree to which security might be violated from the perspective of that user.

## 2.3 Storage nodes

Storage nodes (stores) persistently store objects and provide copies of object data on request to both worker nodes and dissemination nodes. Access control prevents nodes from obtaining data they should not see. When a worker requests a copy of an object from a store, the store examines the confidentiality part of the object's label, and provides the object only if the requesting node is trusted enough to read it. Therefore the object can be sent securely in plaintext between the two nodes (though it is of course encrypted by SSL). This access control mechanism works by treating each Fabric node as a principal. Each principal in Fabric keeps track of how much it trusts the nodes that it interacts with. Trust relationships are created by the delegation mechanisms described in Section 3.1.

Objects on a store are associated with *object groups* containing a set of related objects. When an object is requested by a worker or dissemination node, the entire group is prefetched from the store, amortizing the cost of store operations over multiple objects. Every object in the object group is required to have the same security policy, so that the entire group can be treated uniformly with respect to access control, confidentiality and integrity. The binding between an object and its group is not permanent; the store constructs object groups as needed and discards infrequently used object groups. To improve locality, the store tries to create object groups from objects connected in the object graph.

After a worker fetches an object, it can perform computations using this cached copy, perhaps modifying its state. When the transaction containing these computations completes, the worker commits object updates to the stores that hold objects involved in the transaction. The transaction succeeds only if it is serializable with other transactions at those stores. As with object fetch requests, the store also enforces access control on update requests based upon the degree of trust in the worker and the integrity policies in these objects' labels.

## 2.4 Worker nodes

Workers execute Fabric programs. Fabric programs may be written in the Fabric language. Trusted Fabric programs—that is, trusted by the worker on which they run—may incorporate code written in other languages, such as the Fabric intermediate language, FabIL. However, workers will run code provided by other nodes only if the code is written in Fabric, and signed by a trusted node.

Fabric could, in principle, provide certifying compilation [38], allowing Fabric nodes to check that compiled code obeys the Fabric type system—and therefore that it correctly enforces access control and information flow control—without relying on trusting the compiler or the node that runs it. The design and implementation of this feature are left to future work.

Fabric programs modify objects only inside transactions, which the Fabric programming language exposes to the programmer as a simple `atomic` construct. Transactions can be nested, which is important for making code compositional. During transactions, object updates are logged in an undo/redo log, and are rolled back if the transaction fails either because of inconsistency, deadlock, or an application-defined failure.

A Fabric program may be run entirely on a single worker that issues requests to stores (or to dissemination nodes) for objects that it needs. This data-shipping approach makes sense if the cost of moving data is small compared to the cost of computation, and if the objects' security policies permit the worker to compute using them.

When data shipping does not make sense, function shipping may be used instead. Execution of a Fabric program may be distributed across multiple workers, by using *remote method calls* to transfer control to other workers. Remote method calls in Fabric differ from related mechanisms such as Java RMI [24] or CORBA [41]:

- The *receiver object* on which the method is invoked need not currently be located at the remote worker (more precisely, cached at it). In fact, the receiver object could be cached at the caller, at the callee, or at neither. Invocation causes the callee worker to cache a copy of the receiver object if it does not yet have a copy.

- The entire method call is executed in its own nested transaction, the effects of which are not visible to other code running on the remote node. These effects are not visible until the commit of the top-level transaction containing the nested transaction. The commit protocol (Section 4.3) causes all workers participating in the top-level transaction to commit the subtransactions that they executed as part of it.

- Remote method calls are subject to compile-time and run-time access control checks. The caller side is checked at compile time to determine if the callee is trusted enough to enforce security for the method; the callee checks at run time that the calling node is trusted enough to invoke the method that is being called and to see the results of the method (Section 3.5).

Fabric workers are multithreaded and can concurrently serve requests from other workers. Pessimistic concurrency control (locking) is used to isolate transactions in different threads from each other.

One important use of remote calls is to invoke an operation on a worker colocated with a store. Since a colocated worker has low-cost access to persistent objects, this can improve performance substantially. This idea is analogous to a conventional application issuing a database query for low-cost access to persistent data. In Fabric, a remote call to a worker that is colocated with a store can be used to achieve this goal, with two advantages compared to database queries: the worker can run arbitrary Fabric code, and information-flow security is enforced.

## 2.5 Dissemination nodes

To improve the scalability of Fabric, a store can send copies of objects to *dissemination nodes*. Rather than requesting objects from remote or heavily loaded stores, workers can request objects from dissemination nodes. Dissemination nodes improve scalability because they help deal with popular objects that would otherwise turn the stores holding them into bottlenecks. Objects are disseminated at the granularity of object groups, to amortize the costs associated with fetching remote objects.

Stores provide object data in encrypted form on request to dissemination nodes. Receiving encrypted objects does not require as much trust, because the fields of the object are not visible without the object's encryption key, which dissemination nodes do not in general possess.

Fabric has no prescribed dissemination layer; workers may use any dissemination nodes they choose, and dissemination nodes may use whatever mechanism they want to find and provide objects. In the current Fabric implementation, the dissemination nodes form a peer-to-peer content distribution network based on FreePastry [47]. However, other dissemination architectures can be substituted if the interface to workers and stores remains the same.

To avoid placing trust in the dissemination layer, disseminated object groups are encrypted using a symmetric key and signed with the public key of the originating store. The symmetric encryption key is stored in a *key object* that is not disseminated and must be fetched directly from its store. When an object group is fetched, the dissemination node sends the oid of the key object and the random initialization vector needed for decryption. Key objects are ordinarily shared across many disseminated object groups, so workers should not need to fetch them often.

Disseminated object groups are identified by dissemination nodes based on the oid of a contained object called the *head object*. The oid of the head object is exposed in the object group, but other oids in the object group (and the contents of all objects) are hidden by encryption.

To help keep caches up to date, workers and dissemination nodes are implicitly subscribed to any object group they read from a store. When any object in the group is updated, the store sends the updated group to its subscribers. The dissemination layer is responsible for

pushing updated groups to workers that have read them. A transaction that has read out-of-date data can then be aborted and retried by its worker on receipt of the updated group.

The fetch requests that dissemination nodes receive may allow them to learn something about what workers are doing. To control this information channel, dissemination nodes are assigned a label representing the maximum confidentiality of information that may leak on this channel. Workers use dissemination nodes only for fetch requests that do not leak more than this maximum. Other requests go directly to stores.

# 3. The Fabric language

Fabric offers a high-level programming language for building distributed programs. This language permits code running at a given Fabric node to access objects or code residing at other nodes in the system.

The Fabric programming language is an extension to the Jif programming language [33, 36], which also enforces secure information flow and has been used to build a few significant systems (e.g., [21, 14]). To support distributed programming, Fabric adds two major features to Jif:

- Nested transactions ensure that computations observe and update objects consistently, and support clean recovery from failures.
- Remote method calls (remote procedure calls to methods) allow distributed computations that span many workers.

These features are unusual but not new (e.g., Argus [30] has both, though it lacks data shipping between workers). What *is* new is combining these features with information flow security, which requires new mechanisms so that, for example, transactions do not leak confidential information, and remote calls are properly authorized. To support compile-time and run-time security enforcement for secure distributed computation, Fabric adds a new trust ordering on information flow labels. Further, Fabric extends Jif with new support for trust management, integrated with a public-key infrastructure (PKI).

## 3.1 Principals

Principals capture authority, privilege, and trust in Fabric. Principals represent users, roles, groups, organizations, privileges, and Fabric nodes. As in Jif [33], they are manifested in the Fabric programming language as the built-in type `principal`.

When running, Fabric code can possess the *authority* of a principal, and may carry out actions permitted to that principal. The authority to act as principal $p$ can be delegated during a method call, if the method is annotated with a clause `where caller(`$p$`)`. This model of delegating authority has similarities to Java stack inspection [53]; it differs in that authority is statically checked except at remote method calls, where the receiver checks that the caller is sufficiently trusted.

Trust relationships between principals are represented by the *acts-for* relation [35]. If principal $p$ acts for principal $q$, any action by principal $p$ can be considered to come from principal $q$ as well. These actions include statements made by principal $p$. Thus, this acts-for relationship means $q$ trusts $p$ completely. We write this relationship more compactly as $p \succcurlyeq q$. The acts-for relation $\succcurlyeq$ is transitive and reflexive. There is a top principal $\top$ that acts for all other principals and a bottom principal $\bot$ that all principals act for. The operators $\wedge$ and $\vee$ can be used to form conjunctions and disjunctions of principals.

The acts-for relation can be used for authorization of an action. The idea is to create a principal that represents the privilege needed to perform the action. Any principal that can act for the privilege principal is then able to perform the action. In Fabric code, an access control check is expressed explicitly using `if`. To check if `user` $\succcurlyeq$ `priv`, we write:

```
principal user, priv;
   ...
   if (user actsfor priv) {
        ... do action ...
   }
```

The Fabric model of principals extends that of Jif 3.0 [13], which represents principals as objects inherited from the abstract class `fabric.lang.Principal`. Instances of any subclass can be used as principals, and the methods of these classes automatically possess the authority of the instance `this`—an object acts for at least itself. Principals control their acts-for relationships by implementing a method `p.delegatesTo(q)`, which tests whether `q` acts for `p`. This allows a principal to say who can directly act for it; the Fabric run-time system at each worker node automatically computes and caches the transitive closure of these direct acts-for relationships. The run-time system also exposes operations for notifying it that acts-for relationships have been added and removed. These operations cause the acts-for cache to be updated conservatively to remove any information that might be stale. In general, worker nodes may have different partial views of the acts-for relation; this is not a problem, because of the monotonicity of the label system [35].

Unlike in SIF, acts-for relationships can be used by principals to specify the degree to which they trust Fabric nodes. Fabric nodes are represented as first-class objects in Fabric, and they are also principals. For example, a storage node might be represented as a variable `s`. The test `s actsfor p` would then test whether `p` trusts `s`. This would always be the case if the principal `p` were stored at store `s`.

Principals implement their own authentication. Principal objects have an `authenticate` method that implements authentication and also optionally some authorization. This method takes as an argument an *authentication proof* that establishes the right of the caller to perform an action, provided to `authenticate` as a closure. Principals can implement authentication proofs in many ways—for example, using digital signatures or even passwords. Authentication proofs allow trust to be bootstrapped in Fabric. For example, if user `u` wants to start using Fabric from a new worker `w`, the user can establish that the worker is trusted by adding the acts-for relation `w` $\succcurlyeq$ `u`. To add the relation requires the authority of `u`, so code on the worker makes a remote call to `u.authenticate` on another, trusted worker, passing an authentication proof and a closure that invokes `u.addDelegatesTo(w)`.

Fabric has a built-in way to authenticate worker nodes as corresponding to their Fabric worker node objects. This is accomplished using X.509 certificates [23] that include the node's hostname and the oid of its principal object. Whether the certificates of a given certificate authority are accepted is decided by the Fabric node receiving them.

## 3.2 Labels

Information security is provided by information flow control. All information is labeled with policies for confidentiality and integrity. These labels are propagated through computation using compile-time type checking, but run-time checks are used for dynamic policies and to deal with untrusted nodes.

Information flow security policies are expressed in terms of principals, which is important because it enables the integration of access control and information flow control. A key use of this integra-

tion is for authorizing the downgrading of information flow policies through declassification (for confidentiality) and endorsement (for integrity).

For example, the confidentiality policy `alice→bob` says that principal `alice` owns the policy and that she permits principal `bob` to read it. Similarly, the integrity policy `alice←bob` means that `alice` permits `bob` to affect the labeled information. A label is simply a set of such policies, such as `{alice→bob; bob→alice}`.

These *decentralized labels* [35] keep track of *whose* security is being enforced, which is useful for Fabric, where principals need to cooperate despite mutual distrust. On a worker `w` trusted by `alice` (i.e., $\texttt{w} \succcurlyeq \texttt{alice}$), information labeled with the policy `alice→bob` can be explicitly declassified by code that is running with the authority of `alice`, removing that policy from its label.

**Information flow ordering.** The Fabric compiler checks information flows at compile time to ensure that both explicit and implicit [17] information flows are secure. The *information flow ordering* $L_1 \sqsubseteq L_2$ captures when information flow from $L_1$ to $L_2$ is secure. For example, we have $\{\texttt{alice} \rightarrow \texttt{bob}\} \sqsubseteq \{\texttt{charlie} \rightarrow \texttt{dora}\}$ exactly when $\texttt{charlie} \succcurlyeq \texttt{alice}$, and $\texttt{dora} \succcurlyeq \texttt{bob}$ or $\texttt{dora} \succcurlyeq \texttt{alice}$.[1] Integrity works the opposite way, because integrity policies allow flow from trusted sources to untrusted recipients: the relationship $\{\texttt{alice} \leftarrow \texttt{bob}\} \sqsubseteq \{\texttt{charlie} \leftarrow \texttt{dora}\}$ holds iff we have $\texttt{alice} \succcurlyeq \texttt{charlie}$, and $\texttt{bob} \succcurlyeq \texttt{dora}$ or $\texttt{bob} \succcurlyeq \texttt{charlie}$. See [34] for more justification of these rules.

The following code illustrates these rules. The assignment from `y` to `x` (line 3) is secure because the information in `y` can be learned by fewer readers (only `bob` rather than both `bob` and `charlie`). The assignment from `x` to `y` (line 4) is rejected by the compiler, because it permits `charlie` to read the information. However, the second assignment from `x` to `y` (line 6) is allowed because it occurs in a context where `charlie` is known to act for `bob`, and can therefore already read any information that `bob` can.

```
1 int {alice→bob} x;
2 int {alice→bob, charlie} y;
3 x = y; // OK: bob ≽ (bob ∨ charlie)
4 y = x; // Invalid
5 if (charlie actsfor bob) {
6     y = x; // OK: (bob ∨ charlie) ≽ bob
7 }
```

**Trust ordering.** Fabric extends the DLM by defining a second ordering on labels, the *trust ordering*, which is useful for reasoning about the enforcement of policies by a partially trusted platform. A label $L_1$ may require at least as much *trust* as a label $L_2$, which we write as $L_1 \succcurlyeq L_2$ by analogy with the trust ordering on principals. If $L_1$ requires at least as much trust as $L_2$, then any platform trusted to enforce $L_1$ is also trusted to enforce $L_2$. This happens when $L_1$ describes confidentiality and integrity policies that are at least as strong as those in $L_2$; unlike in the information flow ordering, integrity is not opposite to confidentiality in the trust ordering.

Therefore, both confidentiality and integrity use the same rules in the trust ordering: both $\{\texttt{alice} \rightarrow \texttt{bob}\} \succcurlyeq \{\texttt{charlie} \rightarrow \texttt{dora}\}$ and $\{\texttt{alice} \leftarrow \texttt{bob}\} \succcurlyeq \{\texttt{charlie} \leftarrow \texttt{dora}\}$ are true exactly when $\texttt{alice} \succcurlyeq \texttt{charlie}$ and $\texttt{bob} \succcurlyeq \texttt{dora} \vee \texttt{charlie}$.

Figure 2 depicts how the two label orderings relate. In the information flow ordering, the least label describes information that can be used everywhere, because it is public and completely trustworthy: $\{\bot \rightarrow \bot; \top \leftarrow \top\}$. The greatest label describes information that can be used nowhere, because it is completely secret and completely untrustworthy: $\{\top \rightarrow \top; \bot \leftarrow \bot\}$. In the trust ordering, the least label describes information that requires no trust to enforce its security, because it is public and untrusted: $\{\bot \rightarrow \bot; \bot \leftarrow \bot\}$. Because policies owned by $\bot$ can be dropped, this is the default label, which can be written as $\{\}$. The greatest label in the trust ordering is for information that is maximally secret and trusted: $\{\top \rightarrow \top; \top \leftarrow \top\}$.

[1] The final disjunct is there because `alice` is implicitly a reader in her own policy; the policy `alice→bob` is equivalent to `alice→bob∨alice`, also written as `alice→bob, alice`.

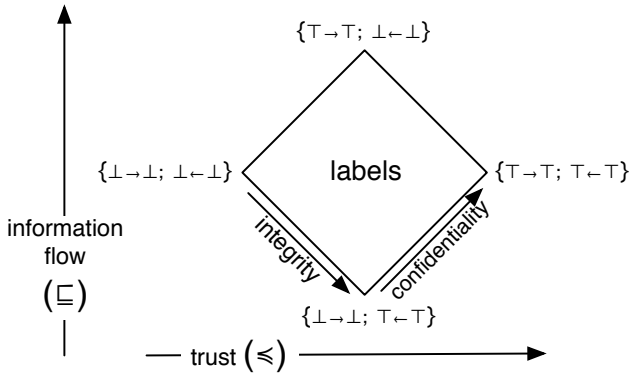


**Figure 2: Orderings on the space of labels**

## 3.3 Object labels

Every Fabric object has a single immutable label that governs the use of information in that object. The label determines which storage nodes can store the object persistently and which worker nodes can cache and compute directly on the object. It also controls which object groups an object may be part of and which key objects may be used to encrypt it. This is a simplification of Jif, which permits object fields to have different labels. Jif objects whose fields have different labels can be encoded as Fabric objects by introducing an additional level of indirection.

An object with label $L_o$ may be stored securely on a store $n$ if the store is trusted to enforce $L_o$. Recalling that $n$ can be used as a principal, this condition is captured formally using the trust ordering on labels:

$$\{\top \rightarrow n; \top \leftarrow n\} \succcurlyeq L_o \tag{1}$$

To see this, suppose $L_o$ has a confidentiality policy $\{p \rightarrow q\}$, which is equivalent to $\{p \rightarrow p \vee q\}$. Condition 1 implies $n \succcurlyeq p$ or $n \succcurlyeq q$—either $p$ must trust $n$, or $p$ must believe that $n$ is allowed to read things that $q$ is allowed to read. Conversely, if $L_o$ has an integrity policy $\{p \leftarrow q\}$, we require the same condition, $n \succcurlyeq p \vee q$—either $p$ trusts $n$, or $p$ believes that $n$ is allowed to affect things that $q$ is. Therefore we can write $L(n)$ to denote the label corresponding to node $n$, which is $\{\top \rightarrow n; \top \leftarrow n\}$, and express condition 1 simply as $L(n) \succcurlyeq L_o$.

Fabric classes may be parameterized with respect to labels or principals, so different instances of the same class may have different labels. This allows implementation of reusable classes, such as data structures that can hold information with different labels.

By design, Fabric does not provide *persistence by reachability* [3], because it can lead to unintended persistence. Therefore, constructors are annotated to indicate on which store the newly created object should be made persistent. The call `new C@s(...)` creates a new object of class `C` whose store is identified by the variable `s`. No communication with the store is needed until commit. If the store of an object is omitted, the new object is created at the same store as the object whose method calls `new`. Objects may have non-final fields

```
1 void m1{alice←} () {
2   Worker w = findWorker("bob.cs.cornell.edu");
3   if (w actsfor bob) {
4     int{alice→bob} data = 1;
5     int{alice→} y = m2@w(data);
6   }
7 }
8
9 int{alice→bob} m2{alice←} (int{alice→bob} x) {
10  return x+1;
11 }
```

**Figure 3: A remote call in Fabric**

that are marked `transient`. These transient fields are not saved persistently, which is similar to their treatment by Java serialization.

## 3.4 Tracking implicit flows

Information can be conveyed by program control flow. If not controlled, these *implicit flows* can allow adversaries to learn about confidential information from control flow, or to influence high-integrity information by affecting control flow.

Fabric controls implicit flows through the *program-counter label*, written $L_{\text{pc}}$, which captures the confidentiality and integrity of control flow. The program-counter label works by constraining side effects; to assign to a variable $x$ with label $L_x$, Fabric requires $L_{\text{pc}} \sqsubseteq L_x$. If this condition does not hold, either information with a stronger confidentiality policy could leak into $x$ or information with a weaker integrity policy could affect $x$.

Implicit flows cross method-call boundaries, both local and remote. To track these flows, object methods are annotated with a *begin label* that constrains the program counter label of the caller. The $L_{\text{pc}}$ of the caller must be lower than or equal to the begin label. Implicit flows via exceptions and other control flow mechanisms are also tracked [33].

Because implicit flows are controlled, untrusted code and untrusted data cannot affect high-integrity control flow unless an explicit downgrading action is taken, using the authority of the principals whose integrity policies are affected. Further, because Fabric enforces robustness [12], untrusted code and untrusted data cannot affect information release. Thus, Fabric provides general protection against a wide range of security vulnerabilities.

## 3.5 Remote calls

Distributed control transfers are always explicit in Fabric. Fabric introduces the syntax `o.m@w(`$\mathtt{a}_1$`,...,`$\mathtt{a}_n$`)` to signify a remote method call to the worker node identified by variable `w`, invoking the method `m` of object `o`. Figure 3 shows example code in which at line 5, a method `m1` calls a method `m2` on the same object, but at a remote worker that is dynamically looked up using its hostname. If the syntax `@w` is omitted, the method call is always local, even if the object `o` is not cached on the current node (in this case the object will be fetched and the method invoked locally).

Remote method calls are subject to both compile-time and run-time checking. The compiler permits a call to a remote method only if it can statically determine that the call is secure. Information sent to a worker `w` can be read by the worker, so all information sent in the call (the object, the arguments, and the implicit flow) must have labels $L_s$ where $L_s \sqsubseteq \{\top \rightarrow \mathtt{w}\}$. For example, in Figure 3, the variable `data`, with label $\{\mathtt{alice} \rightarrow \mathtt{bob}\}$, can be passed to method `m2` only because the call happens in a context where it is known that $\mathtt{w} \succcurlyeq \mathtt{bob}$, and hence $\{\mathtt{alice} \rightarrow \mathtt{bob}\} \sqsubseteq \{\top \rightarrow \mathtt{w}\}$.

Information received from `w` can be affected by it, so by a similar argument, all returned information must have labels $L_r$ where $\{\top \leftarrow \mathtt{w}\} \sqsubseteq L_r$.

The recipient of a remote method call has no *a priori* knowledge that the caller is to be trusted, so run-time checking is needed. When a call occurs from sender worker `sw` to receiver worker `rw`, the receiver checks all information sent or received at label $L$ (including implicit flows), to ensure that $L(\mathtt{sw}) \succcurlyeq L$. For example, when `bob.cs.cornell.edu` receives the remote call to `m2`, purporting to provide integrity $\{\mathtt{alice} \leftarrow\}$, it will check that the calling worker has the authority of `alice`. Additional compile-time checks prevent these run-time checks from leaking information themselves.

For example, when the code of Figure 3 invokes method `m1`, the node `w` will check that the calling node acts for `alice`, because the initial integrity of the method is $\{\mathtt{alice} \leftarrow \mathtt{alice}\}$ (written in the code using the shorthand $\{\mathtt{alice} \leftarrow\}$).

## 3.6 Transactions

All changes to Fabric objects take place inside transactions, to provide concurrency control and ensure consistency of reads and writes. A transaction is indicated in Fabric code by the construct `atomic { `$S$` }`, where $S$ is a sequence of statements. The semantics is that the statement $S$ is executed atomically and in isolation from all other computations in Fabric. In other words, Fabric enforces serializability of transactions.

Accesses to mutable fields of Fabric objects are not permitted outside transactions. Reads from objects are permitted outside transactions, but each read is treated as its own transaction.

If $S$ throws an exception, it is considered to have failed, and is aborted. If $S$ terminates successfully, its side effects become visible outside its transaction. Failure due to conflict with other transactions causes the atomic block to be retried automatically. If the maximum number of retries is exceeded, the transaction is terminated.

Transactions may also be explicitly retried or aborted by the programmer. A `retry` statement rolls back the enclosing atomic block and restarts it from the beginning; an `abort` statement also rolls back the enclosing atomic block, but results in throwing the exception `UserAbortException`. Aborting a transaction creates an implicit flow; therefore, Fabric statically enforces that the $L_{\text{pc}}$ of the `abort` is lower than or equal to the $L_{\text{pc}}$ of the atomic block: $L_{\text{pc}}^{\texttt{abort}} \sqsubseteq L_{\text{pc}}^{\texttt{atomic}}$. Exceptions generated by $S$ are checked similarly.

Atomic blocks may be used even during a transaction, because Fabric allows nested transactions. This allows programmers to enforce atomicity without worrying about whether their abstractions are at "top level" or not. Atomic blocks can also be used as a way to cleanly recover from application-defined failures, via `abort`.

Multi-worker computations take place in atomic, isolated transactions that span all the workers involved. The Fabric runtime system ensures that when multiple workers use the same object within a transaction, updates to the object are propagated between them as necessary (Section 5.1).

Transactions are single-threaded; new threads cannot be started inside a transaction, though a worker may run multiple transactions concurrently. This choice was made largely to simplify the implementation, though it maps well onto many of the applications for which Fabric is intended.

Fabric uses a mix of optimistic and pessimistic concurrency control. In the distributed setting, it is optimistic, because worker nodes are allowed to compute on objects that are out of date. However, to coordinate threads running on the same worker, Fabric uses pessimistic concurrency control in which threads acquire locks on objects. Edge chasing [10] allows distributed deadlocks to be detected in Fabric.

## 3.7 Java interoperability

Fabric programs can be written with a mixture of Java, Fabric, and FabIL (the Fabric intermediate language). FabIL is an extension to Java that supports transactions and remote calls, but not information flow labels or static information flow control. More concretely, FabIL supports the `atomic` construct and gives the ability to invoke methods and constructors with annotations `@w` and `@s` respectively. Transaction management is performed on Fabric and FabIL objects but not on Java objects, so the effects of failed transactions on Java objects are not rolled back. The use of FabIL or Java code in Fabric programs offers lower assurance to principals who trust the nodes running this code, but it does not undermine the security assurance offered by the system. FabIL can be convenient for code whose security properties are not accurately captured by static information flow analysis, making the labels of the full Fabric language counterproductive. An example is code implementing cryptography.

# 4. Caches and transactions

A Fabric worker node holds versions of some subset of all Fabric objects. This subset includes nonpersistent objects allocated by the worker node itself, as well as cached versions of objects from stores. During computation on a worker, references to objects not yet cached at the worker may be followed. The worker then issues read requests for the missing objects, either to the dissemination layer or directly to stores. The dissemination layer or store then responds with an object group that includes the requested object.

## 4.1 Transaction bookkeeping

During computation on a worker, reads and writes to objects are logged. The first write to an object during a transaction also logs the prior state of the object so that it can be restored in case the transaction aborts. Because transactions can be nested, transaction logs are hierarchical. When a local subtransaction commits, its log is merged with the parent transaction log.

To reduce logging overhead, the copy of each object at a worker is stamped with a reference to the last transaction that accessed the object. No logging needs to be done for an access if the current transaction matches the stamp.

## 4.2 Versions and transaction management

Each object contains a version number that is incremented when the object is updated by a top-level transaction. At commit time, the version numbers of read objects are compared against the authoritative versions at the store, to determine whether the transaction used up-to-date information.

To conserve memory, cached objects may be *evicted* if they have no uncommitted changes. In the current implementation, eviction is accomplished automatically by the Java run-time system, because cached objects are referenced using a Java `SoftReference` object. The worker records the version numbers of read objects for use at commit time.

When worker `w` commits to store `s`, the commit includes the versions of objects read and written during the transaction and the new data for the written objects. For security, the store checks the label $L_o$ of each updated object to ensure that `w` is trusted to modify the object; the test is $L(\texttt{w}) \succcurlyeq L_o$. This check also ensures that the version number reported by the worker is meaningful.

## 4.3 Hierarchical commit protocol

In general, a transaction may span worker nodes that do not trust each other. This creates both integrity and confidentiality concerns. An untrusted node cannot be relied to commit its part of a transaction correctly. More subtly, the commit protocol might also cause an untrusted node to learn information it should not. Just learning the identities of other nodes that participated in a transaction could allow sensitive information to be inferred. Fabric's hierarchical two-phase commit protocol avoids these problems.

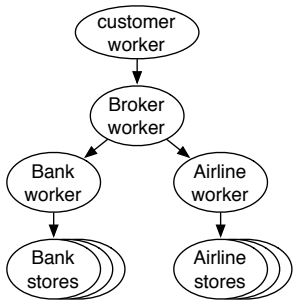


**Figure 4: A hierarchical, distributed transaction**

For example, consider a transaction that updates objects owned by a bank and other objects owned by an airline, perhaps as part of a transaction in which a ticket is purchased (see Figure 4). The bank and the airline do not necessarily trust each other; nor do they trust the customer purchasing the ticket. Therefore some computation is run on workers managed respectively by the bank and the airline. When the transaction is to be committed, some updates to persistent objects are recorded on these different workers.

Because the airline and the bank do not trust the customer, their workers will reject remote calls from the customer—the customer's worker lacks sufficient integrity. Therefore, this scenario requires the customer to find a trusted third party. As shown in the figure, a third-party broker can receive requests from the customer, and then invoke operations on the bank and airline. Because the broker runs at a higher integrity level than the customer that calls it, Fabric's endorsement mechanism must be used to boost integrity. This reflects a security policy that anyone is allowed to make requests of the broker. It is the responsibility of the broker to sanitize and check the customer request before endorsing it and proceeding with the transaction.

The hierarchical commit protocol begins with the worker that started the top-level transaction. It initiates commit by contacting all the stores for whose objects it is the current writer, and all the other workers to which it has issued remote calls. These other workers then recursively do the same, constructing a *commit tree*. This process allows all the stores involved in a transaction to be informed about the transaction commit, without relying on untrusted workers to choose which workers and stores to contact and without revealing to workers which other workers and stores are involved in the transaction lower down in the commit tree. The two-phase commit protocol then proceeds as usual, except that messages are passed up and down the commit tree rather than directly between a single coordinator and the stores.

Of course, a worker in this tree could be compromised and fail to correctly carry out the protocol, causing some stores to be updated in a way that is inconsistent with other stores. However, a worker that could do this could already have introduced this inconsistency by simply failing to update some objects or by failing to issue some remote method calls. In our example above, the broker could cause payment to be rendered without a ticket being issued, but only by violating the trust that was placed in it by the bank and airline. The customer's power over the transaction is merely to prevent it from happening at all, which is not a security violation.

Once a transaction is prepared, it is important for the availabil-

ity of the stores involved that the transaction is committed quickly. The transaction coordinator should remain available, and if it fails after the prepare phase, it must recover in a timely way. An unavailable transaction coordinator could become an availability problem for Fabric, and the availability of the coordinator is therefore a trust assumption. To prevent denial-of-service attacks, prepared transactions are timed out and aborted if the coordinator is unresponsive. In the example given, the broker can cause inconsistent commits by permanently failing after telling only the airline to commit, in which case the bank will abort its part of the transaction. This failure is considered a violation of trust, but in keeping with the security principles of Fabric, the failing coordinator can only affect the consistency of objects whose integrity it is trusted to enforce. This design weakens Fabric's safety guarantees in a circumscribed way, in exchange for stronger availability guarantees.

## 4.4 Handling failures of optimism

Computations on workers run transactions optimistically, which means that a transaction can fail in various ways. The worker has enough information to roll the transaction back safely in each case. At commit time the system can detect inconsistencies that have arisen because another worker has updated an object accessed during the transaction. The stores inform the workers which objects involved in the transaction were out of date; the workers then flush their caches of the stale objects before retrying.

Another possible failure is that the objects read by the transaction are already inconsistent, breaking invariants on which the code relies. Broken invariants can lead to errors in the execution of the program. Incorrectly computed results are not an issue because they will be detected and rolled back at commit time. Exceptions may also result, but as discussed earlier, exceptions also cause transaction failure and rollback. Finally, a computation might diverge rather than terminate. Fabric handles divergence by retrying transactions that are running too long. On retry, the transaction is given more time in case it is genuinely a long-running transaction. By geometrically growing the retry timeout, the expected run time is inflated by only a constant factor.

Because Fabric has subscription mechanisms for refreshing workers and dissemination nodes with updated objects, the object cache at a worker should tend to be up to date, and inconsistent computations can be detected before a transaction completes.

# 5. Distributed computation

Fabric transactions can be distributed across multiple workers by executing remote calls within a transaction. The whole transaction runs in isolation from other Fabric transactions, and its side effects are committed atomically. The ability to distribute transactions is crucial for reconciling expressiveness with security. Although some workers are not trusted enough to read or write some objects, it is secure for them to perform these updates by calling code on a sufficiently trusted worker.

## 5.1 Writer maps

An object can be accessed and updated by multiple workers within a distributed transaction, each of which may be caching the object. This is challenging. For consistency, workers need to compute on the latest versions of the shared object as they are updated. For performance, workers should be able to locally cache objects that are shared but not updated. For security, updates to an object with confidentiality $L$ should not be learned by a worker `c` unless $L \sqsubseteq \{\top \to \texttt{c}\}$. To allow workers to efficiently check for updates to objects they are caching, without revealing information to workers not trusted to learn about updates, Fabric introduces *writer maps*.

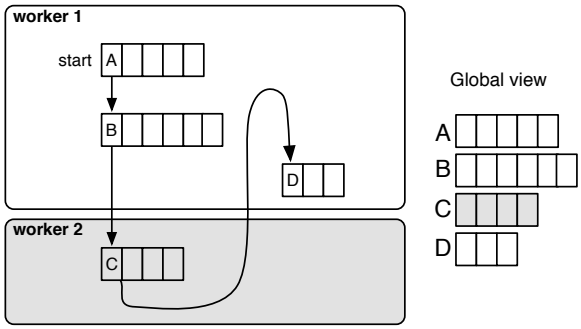


**Figure 5: Logs of nested distributed transactions**

If an object is updated during a distributed transaction, the node performing the update becomes the object's *writer* and stores the definitive copy of the object. If the object already has a writer, it is notified and relinquishes the role (this notification is not a covert channel because the write must be at a level lower than the object's label, which the current writer is already trusted to read). The change of object writer is also recorded in the writer map, which is passed through the distributed computation along with control flow.

The writer map contains two kinds of mappings: writer mappings and label mappings. An update to object `o` at worker `w` adds a writer mapping with the form $hash(\texttt{oid}, \texttt{tid}, \texttt{key}) \mapsto \{w\}_{\texttt{key}}$, where `oid` is the oid of object `o`, `tid` is the transaction identifier, and `key` is the object's encryption key, stored in its key object. This mapping permits a worker that has the right to read or write `o`—and therefore has the encryption key for `o`—to learn whether there is a corresponding entry in the writer map, and to determine which node is currently the object's writer. Nodes lacking the key cannot exploit the writer mapping because without the key, they cannot verify the hash. Because the transaction id is included in the hash, they also cannot watch for the appearance of the same writer mapping across multiple transactions.

Label mappings support object creation. The creation of a new object with oid `oid` adds an entry with the form $hash(\texttt{oid}) \mapsto \texttt{oid}_{\texttt{label}}$, where $\texttt{oid}_{\texttt{label}}$ is the oid of the object's label, which contains the object's encryption key. This second kind of mapping allows a worker to find the encryption key for newly created objects, and then to check the writer map for a mapping of the first kind.

The writer map is an append-only structure, so if an untrusted worker fails to maintain a mapping, it can be restored. The size of the writer map is a covert channel, but the capacity of this channel is bounded by always padding out the number of writer map entries added by each worker to the next largest power of 2, introducing dummy entries containing random data as needed. Therefore a computation that modifies $n$ objects leaks at most $\lg \lg n$ bits of information.

## 5.2 Distributed transaction management

To maintain consistency, transaction management must in general span multiple workers. A worker maintains transaction logs for each top-level transaction it is involved in. These transaction logs must be stored on the workers where the logged actions occurred, because the logs may contain confidential information that other workers may not see. Figure 5 illustrates the log structures that could result in a distributed transaction involving two workers. In the figure, a transaction (A) starts on worker 1, then starts a nested subtransaction (B), then calls code on worker 2, which starts another subtransaction (C) there. That code then calls back to worker

1, starting a third subtransaction (D). Conceptually, all the transaction logs together form a single log that is distributed among the participating workers, as shown on the right-hand side. When D commits, its log is conceptually merged with the log of C, though no data is actually sent. When C commits, its log, including the log of D, is conceptually merged with that of B. In actuality, this causes the log of D to be merged with that of B, but the log for C remains on worker 2. When the top-level transaction commits, workers 1 and 2 communicate with the stores that they have interacted with, using their respective parts of the logs.

# 6. Implementation

The Fabric implementation uses a mixture of Java, FabIL, and Fabric code. Not counting code ported to FabIL from earlier Java and Jif libraries, the implementation includes a total of 33k lines of code.

In addition to a common base of 6.5k lines of code supporting the worker, store, and dissemination nodes, the worker is implemented as 6.4k lines of Java code and 4.6k lines of FabIL code; the store is 2.8k lines of Java; and the dissemination layer is 1.5k lines of Java code. In addition, some of the GNU Classpath collection libraries have been ported to FabIL for use by Fabric programs (another 12k lines of code),

The Fabric compiler, supporting both Fabric and FabIL source files, is a 14k-line extension to the Jif 3.3 compiler [36], itself a 13k-line extension to the Polyglot compiler framework [39].

Implementing Fabric in Java has the advantage that it supports integration with and porting of legacy Java applications, and access to functionality available in Java libraries. However, it limits control over memory layout and prevents the use of many implementation techniques. In an ideal implementation, the virtual machine and JIT would be extended to support Fabric directly. For example, the Java `SoftReference` capability that is used for eviction could be implemented with fewer indirections. We leave VM extensions to future work.

## 6.1 Store

The current store implementation uses Berkeley DB [40] as a backing store in a simple way: each object is entered individually with its oid as its key and its serialized representation as the corresponding value. Because stores cache both object groups and object versions in memory, and because workers are able to aggressively cache objects, the performance of this simple implementation is reasonable for the applications we have studied. For write-intensive workloads, object clustering at the backing store is likely to improve performance; we leave this to future work.

It is important for performance to keep the representation of an object at a store and on the wire compact. Therefore, references from one object to another are stored as onums rather than as full oids. A reference to an object located at another Fabric node is stored as an onum that is bound at that store to the full oid of the referenced object. This works well assuming most references are to an object in the same store.

## 6.2 Dissemination layer

The current dissemination layer is built using FreePastry [47], extended with proactive popularity-based replication based on Beehive [43] and with propagation of object updates. The popularity-based replication algorithm replicates objects according to their popularity, with the aim of achieving a constant expected number of hops per lookup.

The Fabric dissemination layer also propagates updates to object groups. Requests to stores fetch encrypted object groups and establish subscriptions for those groups. When a store notifies a dissemination node about an update, that node propagates the update through the dissemination layer, invalidating old versions of the group.

One standard configuration of Fabric worker nodes includes a colocated dissemination node to which dissemination layer requests are directed; with this configuration, the size of the dissemination layer scales in the number of worker nodes.

## 6.3 Unimplemented features

Most of the Fabric design described in this paper has been implemented in the current prototype. A few features are not, though no difficulties are foreseen in implementing them: distributed deadlock detection via edge chasing [10], timeout-based abort of possibly divergent computations, early detection of inconsistent transactions based on updates from subscriptions, path compression for pointer chains created by mobile objects, and avoidance of read channels at dissemination nodes.

# 7. Evaluation

## 7.1 Course Management System

To examine whether Fabric can be used to build real-world programs, and how its performance compares to common alternatives, we ported a portion of a course management system (CMS) [6] to FabIL. CMS is a 54k line J2EE web application written using EJB 2.0 [16], backed by an Oracle database. It has been used for course management at Cornell University since 2005; at present, it is used by more than 40 courses and more than 2000 students.

**Implementation.**

CMS uses the model/view/controller design pattern; the model is implemented with Enterprise JavaBeans using Bean-Managed Persistence. For performance, hand-written SQL queries are used to implement lookup and update methods, while generated code manages object caches and database connections. The model contains 35 Bean classes encapsulating students, assignments, courses, and other abstractions. The view is implemented using Java Server Pages.

We ported CMS to FabIL in two phases. First, we replaced the Enterprise JavaBean infrastructure with a simple, non-persistent Java implementation based on the Collections API. We ported the entire data schema and partially implemented the query functionality of the model, focusing on the key application features. Of the 35 Bean classes, 5 have been fully ported. By replacing complex queries with object-oriented code, we were able to simplify the model code a great deal: the five fully ported classes were reduced from 3100 lines of code to 740 lines, while keeping the view and controller mostly unchanged. This intermediate version, which we will call the Java implementation, took one developer a month to complete and contains 23k lines of code.

Porting the Java implementation to FabIL required only superficial changes, such as replacing references to the Java Collections Framework with references to the corresponding Fabric classes, and adding label and store annotations. The FabIL version adds fewer than 50 lines of code to the Java implementation, and differs in fewer than 400 lines. The port was done in less than two weeks by an undergraduate initially unfamiliar with Fabric. These results suggest that porting web applications to Fabric is not difficult and results in shorter, simpler code.

A complete port of CMS to Fabric would have the benefit of federated, secure sharing of CMS data across different administrative domains, such as different universities, assuming that information is assigned labels in a fine-grained way. It would also permit secure

| | Page Latency (ms) | | |
|---|---|---|---|
| | Course | Students | Update |
| EJB | 305 | 485 | 473 |
| Hilda | 432 | 309 | 431 |
| FabIL | 35 | 91 | 191 |
| FabIL/memory | 35 | 57 | 87 |
| Java | 19 | 21 | 21 |

**Table 1: CMS page load times (ms) under continuous load.**

| | total | app | tx | log | fetch | store |
|---|---|---|---|---|---|---|
| Cold | 9153 | 10% | 2% | 12% | 74% | 2% |
| Warm | 6043 | 27% | 3% | 6% | 61% | 3% |
| Hot | 840 | 46% | 14% | 24% | 0% | 17% |

**Table 2: Breakdown of OO7 traversal time (times in ms)**

access to CMS data from applications other than CMS. We leave this to future work.

**Performance.**

The performance of Fabric was evaluated by comparing five different implementations of CMS: the production CMS system based on EJB 2.0, the in-memory Java implementation (a best case), the FabIL implementation, the FabIL implementation running with an in-memory store (FabIL/memory), and a fifth implementation developed earlier using the Hilda language [54]. Comparing against the Hilda implementation is useful because it is the best-performing prior version of CMS. The performance of each of these systems was measured on some representative user actions on a course containing 55 students: viewing the course overview page, viewing information about all students enrolled in the course, and updating the final grades for all students in the course. All three of these actions are compute- and data-intensive.

All Fabric and Java results were acquired with the app server on a 2.6GHz single-core Intel Pentium 4 machine with 2GB RAM. The Hilda and EJB results were acquired on slightly better hardware: the Hilda machine had the same CPU and 4GB of memory; EJB results were acquired on the production configuration, a 3GHz dual-core Intel Xeon with 8GB RAM.

Table 1 shows the median load times for the three user actions under continuous load. The first three measurements in Table 1 show that the Fabric implementation of CMS runs faster than the previous implementations of CMS. The comparison between the Java and nonpersistent FabIL implementations illustrate that much of the run-time overhead of Fabric comes from transaction management and from communication with the remote store.

## 7.2 Multiuser SIF calendar

Fabric labels are intended to enforce the construction of secure distributed applications, even in an environment of mutual distrust. To evaluate whether this goal is achieved, we ported the multiuser calendar application originally written for SIF [13] to Fabric. This application allows users to create shared events and to control the visibility of their events using information flow policies.

The application is structured as a standard web application server running on a Fabric worker node. Persistent data is kept on one or more storage nodes, but the worker and the stores do not necessarily trust each other. The design allows users to maintain their calendar events on a store they trust, and application servers can run on any worker the user trusts. This design is in contrast to current distributed calendars where all events are maintained on a single globally trusted domain.

In the SIF version, security is enforced by explicitly labeling application data with confidentiality and integrity policies, but persistence is achieved with a MySQL backing store. The Fabric version straightforwardly removes the use of MySQL by making existing objects persistent.

The original SIF framework has about 4000 non-comment, non-blank lines of Java code, 1000 lines of Jif signatures, and a 900-line Jif library that implements user management. The Calendar application is another 1800 lines of Jif code.

Porting SIF to Fabric required changes only to the User library, because users are persistent objects. These changes involved refactoring so all fields shared the same label, and removing parametric labels with no run-time representation.

Porting the Calendar application required similar changes. Using the persistence features of Fabric simplifies its code by eliminating 414 lines of code for encoding objects into MySQL. The application was distributed by introducing remote calls to perform queries on a store-colocated worker. Static checks performed by the Fabric compiler force the insertion of additional dynamic label and principal tests, to ensure that persistent object creation and remote calls are secure.

## 7.3 Run-time overhead

To evaluate the overhead of Fabric computation at the worker when compared to ordinary computation on nonpersistent objects, and to understand the effectiveness of object caching at both the store and the worker, we used the OO7 object-oriented database benchmark [8]. We measured the performance of a read-only (T1) traversal on an OO7 small database, which contains 153k objects totaling 24Mb.

The results of these measurements are summarized in Table 2. Performance was measured in three configurations: (1) cold, (2) warm, with stores caching object groups, and (3) hot, with both the store and worker caches warmed up.

The results show that caching is effective at both the worker and the store. However the plain in-memory Java implementation of OO7 runs in 66ms, which is about 10 times faster than the worker-side part of the hot traversal. Because Fabric is designed for computing on persistent data, this is an acceptable overhead for many, though not all, applications. For computations that require lower overhead, Fabric applications can always incorporate ordinary Java code, though that code must implement its own failure recovery.

# 8. Related work

Fabric provides a higher-level abstraction for programming distributed systems. Because it aims to help with many different issues, including persistence, consistency, security, and distributed computation, it overlaps with many systems that address a subset of these issues. However, none of these prior systems addresses all the issues tackled by Fabric.

OceanStore [45] shares the goal with Fabric of a federated, distributed object store. OceanStore is more focused on storage than on computation. It provides consistency only at the granularity of single objects, and does not help with consistent distributed computation. OceanStore focuses on achieving durability via replication. Fabric stores could be replicated but currently are not. Unlike OceanStore, Fabric provides a principled model for declaring and enforcing strong security properties in the presence of distrusted worker and storage nodes.

Prior distributed systems that use language-based security to en-

force strong confidentiality and integrity in the presence of distrusted participating nodes, such as Jif/split [55], SIF [13], Swift [11], and have had more limited goals. They do not allow new nodes to join the system, and they do not support consistent, distributed computations over shared persistent data. They do use program analysis to control read channels [55], which Fabric does not.

DStar [56] controls information flow in a distributed system using run-time taint tracking at the OS level, with Flume-style decentralized labels [26]. Like Fabric, DStar is a decentralized system that allows new nodes to join, but does not require certificate authorities. DStar has the advantage that it does not require language support, but controls information flow more coarsely. DStar does not support consistent distributed computations or data shipping.

Some previous distributed storage systems have used transactions to implement strong consistency guarantees, including Mneme [32], Thor [29] and Sinfonia [1]. Cache management in Fabric is inspired by that in Thor [9]. Fabric is also related to other systems that provide transparent access to persistent objects, such as ObjectStore [28] and GemStone [7] These prior systems do not focus on security enforcement in the presence of distrusted nodes, and do not support consistent computations spanning multiple compute nodes.

Distributed computation systems with support for consistency, such as Argus [30] and Avalon [20], usually do not have a single-system view of persistent data and do not enforce information security. Emerald [4] gives a single-system view of a universe of objects while exposing location and mobility, but does not support transactions, data shipping or secure federation. InterWeave [51] synthesizes data- and function-shipping in a manner similar to Fabric, and allows multiple remote calls to be bound within a transaction, remaining atomic and isolated with respect to other transactions. However, InterWeave has no support for information security. The work of Shrira et al. [49] on exo-leases supports nested optimistic transactions in a client–server system with disconnected, multi-client transactions, but does not consider information security. MapJAX [37] provides an abstraction for sharing data structures between the client and server in web applications, but does not consider security. Other recent language-based abstractions for distributed computing such as X10 [48] and Live Objects [42] also raise the abstraction level of distributed computing but do not support persistence or information flow security.

Some distributed storage systems such as PAST [46], Shark [2], CFS [15], and Boxwood [31] use distributed data structures to provide scalable file systems, but offer weak consistency and security guarantees for distributed computation.

## 9. Conclusions

We have explored the design and implementation of Fabric, a new distributed platform for general secure sharing of information and computation resources. Fabric provides a high-level abstraction for secure, consistent, distributed general-purpose computations using distributed, persistent information. Persistent information is conveniently presented as language-level objects connected by pointers. Fabric exposes security assumptions and policies explicitly and declaratively. It flexibly supports both data-shipping and function-shipping styles of computation. Results from implementing complex, realistic systems in Fabric, such as CMS and SIF, suggest it has the expressive power and performance to be useful in practice.

Fabric led to some technical contributions. Fabric extends the Jif programming language with new features for distributed programming, while showing how to integrate those features with secure information flow. This integration requires a new trust ordering on information flow labels, and new implementation mechanisms such as writer maps and hierarchical two-phase commit.

While Fabric perhaps goes farther toward the goal of securely and transparently sharing distributed resources than prior systems, there are many hard problems left to solve. For example, persistent objects introduce the difficult problem of schema evolution; recent work on object adaptation may help. Also, Fabric does not guarantee availability in the way that it does confidentiality and integrity; this remains an interesting topic. The performance of Fabric is limited by its strong consistency guarantees; a principled way to weaken these guarantees would also be valuable.

## Acknowledgments

Nate Nystrom and Xin Zheng were involved in early stages of the Fabric project, and Xin started the dissemination layer. Steve Chong provided guidance on extending Jif and quickly fixed Jif bugs. Dora Abdullah helped set up experiments. We thank Hakim Weatherspoon, Aslan Askarov, Barbara Liskov, Nickolai Zeldovich, and especially Michael Clarkson for discussions on Fabric and this paper. The SOSP reviewers also gave much helpful feedback.

This work was supported in part by the National Science Foundation under grants 0430161 and 0627649; by a grant from Microsoft Corporation; by AF-TRUST, which receives support from the DAF Air Force Office of Scientific Research (FA9550-06-1-0244) and the NSF (0424422); by NICECAP Grant FA8750-08-2-0079, monitored by Air Force Research Laboratories; and by the Office of Naval Research under award N000140910652. This work does not necessarily represent the opinions, expressed or implied, of any of these sponsors.

## 10. References

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, pages 159–174, October 2007.

[2] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[3] M. Atkinson et al. The object-oriented database manifesto. In *Proc. International Conference on Deductive Object Oriented Databases*, Kyoto, Japan, December 1989.

[4] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proc. 1st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–86, November 1986.

[5] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, 2004.

[6] Chavdar Botev et al. Supporting workflow in a course management system. In *Proc. 36th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 262–266, February 2005.

[7] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone Object Database Management System. *Comm. of the ACM*, 34(10):64–77, October 1991.

[8] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

[9] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, October 1997.

[10] K. Mani Chandy, J. Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2), 1983.

[11] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, October 2007.

[12] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proc. 19th IEEE Computer Security Foundations Workshop*, pages 242–253, July 2006.

[13] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security Symposium*, August 2007.

[14] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proc. IEEE Symposium on Security and Privacy*, pages 354–368, May 2008.

[15] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symp. on Operating Systems Principles (SOSP)*, October 2001.

[16] Linda G. DeMichiel. *Enterprise JavaBeans Specifications, Version 2.1*. Sun Microsystems.

[17] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[18] J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3):143–155, March 1966.

[19] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *In Proc. IEEE Workshop on Hot Topics in Operating Systems*, Schoss Elmau, Germany, May 2001.

[20] M. Herlihy and J. Wing. Avalon: Language support for reliable distributed systems. In *Proc. 17th International Symposium on Fault-Tolerant Computing*, pages 89–94. IEEE, July 1987.

[21] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In *22nd Annual Computer Security Applications Conference (ACSAC)*, December 2006.

[22] Health insurance portability and privacy act of 1996. Public Law 104–191, 1996.

[23] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Internet RFC-3280, April 2002.

[24] JavaSoft. Java Remote Method Invocation . `http://java.sun.com/products/jdk/rmi`, 1999.

[25] Linda T. Kohn, Janet M. Corrigan, and Molla S. Donaldson, editors. *To Err is Human: Building a Safer Health System*. The National Academies Press, Washington, D.C., April 2000.

[26] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[27] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. 9th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[28] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Comm. of the ACM*, 34(10):50–63, October 1991.

[29] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

[30] Barbara H. Liskov. The Argus language and system. In *Distributed Systems: Methods and Tools for Specification*, volume 150 of *Lecture Notes in Computer Science*, pages 343–430. Springer-Verlag Berlin, 1985.

[31] John MacCormick, Nick Murph, Marc Najor, Chandramohan A. Thekkat, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, December 2004.

[32] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Office Information Systems*, 8(2):103–139, March 1990.

[33] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[34] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, January 1999. Ph.D. thesis.

[35] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[36] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[37] Daniel Myers, Jennifer Carlisle, James Cowling, and Barbara Liskov. Mapjax: Data structure abstractions for asynchronous web applications. In *Proc. 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[38] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[39] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Compiler Construction Conference (CC'03)*, pages 138–152, April 2003. LNCS 2622.

[40] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proc. USENIX Annual Technical Conference*, 1999.

[41] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[42] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. Programming with live distributed objects. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP)*, 2008.

[43] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: $O(1)$ lookup performance for power-law query distributions in peer-to-peer overlays. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[44] Sean Rhea, Brighten Dodfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of ACM SIGCOMM '05 Symposium*, 2005.

[45] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *2nd USENIX Conference on File and Storage Technologies*, pages 1–14, 2003.

[46] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, October 2001.

[47] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale

peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[48] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *Proc. 12th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[49] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proc. ACM/IFIP/Usenix International Middleware Conference (Middleware 2008)*, December 2008.

[50] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

[51] Chunqiang Tang, DeQing Chen, Sandhya Dwarjadas, and Michael L. Scott. Integrating remote invocation and distributed shared state. In *Proc. 18th International Parallel and Distributed Processing Symposium*, April 2004.

[52] W3C. SOAP version 1.2, June 2003. W3C Recommendation, at `http://www.w3.org/TR/soap12`.

[53] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, California, USA, May 1998.

[54] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applictions with automatic client-server partitioning. In *Proc. 16th International World Wide Web Conference (WWW'07)*, pages 341–350, 2007.

[55] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[56] Nickolai Zeldovich, Silas Boyd, and David Mazières. Securing distributed systems with information flow control. In *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

[57] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, California, May 2003.