Report Number: WUCSE-2005-61

2005-12-01

Architecture and Execution Model for a Survivable Workflow Transaction Infrastructure

Haraldur D. Thorvaldsson and Kenneth J. Goldman

We present a novel architecture and execution model for an infrastructure supporting fault-tolerant, long-running distributed applications spanning multiple administrative domains. Components for both transaction processing and persistent state are replicated across multiple servers, en-suring that applications continue to function correctly de-spite arbitrary (Byzantine) failure of a bounded number of servers. We give a formal model of application execution, based on atomic execution steps, linearizability and a sep-aration between data objects and transactions that act on them. The architecture is designed for robust interoperability across domains, in an open and shared Internet computing infrastructure. A notable feature supporting cross-domain applications... **Read complete abstract on page 2.**

Architecture and Execution Model for a Survivable Workflow Transaction Infrastructure

Haraldur D. Thorvaldsson and Kenneth J. Goldman

Complete Abstract:

We present a novel architecture and execution model for an infrastructure supporting fault-tolerant, long-running distributed applications spanning multiple administrative domains. Components for both transaction processing and persistent state are replicated across multiple servers, en-suring that applications continue to function correctly de-spite arbitrary (Byzantine) failure of a bounded number of servers. We give a formal model of application execution, based on atomic execution steps, linearizability and a sep-aration between data objects and transactions that act on them. The architecture is designed for robust interoperability across domains, in an open and shared Internet computing infrastructure. A notable feature supporting cross-domain applications is that they may declare invariant constraints between data objects and furthermore declare dependencies on constraints maintained by other applications, leading to flexible, incidental atomicity between applications. The ar-chitecture is highly evolvable, maintaining system availabil-ity and integrity during upgrades to both application com-ponents and the system software itself.

2005-61

Architecture and Execution Model for a Survivable Workflow Transaction Infrastructure

Authors: Haraldur D. Thorvaldsson, Kenneth J. Goldman

Corresponding Author: kjg@cse.wustl.edu

Abstract: We present a novel architecture and execution model for an infrastructure supporting fault-tolerant, long-running distributed applications spanning multiple administrative domains. Components for both transaction processing and persistent state are replicated across multiple servers, ensuring that applications continue to function correctly despite arbitrary (Byzantine) failure of a bounded number of servers. We give a formal model of application execution, based on atomic execution steps, linearizability and a separation between data objects and transactions that act on them.

The architecture is designed for robust interoperability across domains, in an open and shared Internet computing infrastructure. A notable feature supporting cross-domain applications is that they may declare invariant constraints between data objects and furthermore declare dependencies on constraints maintained by other applications, leading to flexible, incidental atomicity between applications. The architecture is highly evolvable, maintaining system availability and integrity during upgrades to both application components and the system software itself.

Type of Report: Other

Architecture and Execution Model for a Survivable Workflow Transaction Infrastructure

Haraldur D. Thorvaldsson Kenneth J. Goldman
Dept. of Computer Science and Engineering
Washington University in St. Louis, St. Louis, MO 63130
{ harri, kjg }@cse.wustl.edu

Abstract

We present a novel architecture and execution model for an infrastructure supporting fault-tolerant, long-running distributed applications spanning multiple administrative domains. Components for both transaction processing and persistent state are replicated across multiple servers, ensuring that applications continue to function correctly despite arbitrary (Byzantine) failure of a bounded number of servers. We give a formal model of application execution, based on atomic execution steps, linearizability and a separation between data objects and transactions that act on them.

The architecture is designed for robust interoperability across domains, in an open and shared Internet computing infrastructure. A notable feature supporting cross-domain applications is that they may declare invariant constraints between data objects and furthermore declare dependencies on constraints maintained by other applications, leading to flexible, incidental atomicity between applications. The architecture is highly evolvable, maintaining system availability and integrity during upgrades to both application components and the system software itself.

1. Introduction

The Internet revolutionized computer networking by providing a common, global communication infrastructure. However, building dependable and secure applications on top of it still poses a challenge, as witnessed by intermittent availability of Internet-based services and attacks successfully breaching their security. Meanwhile, demand for globally distributed yet interoperable information systems has long been growing, as has the realization that software architecture paradigms for traditional centralized systems do not easily carry over to distributed and decentralized ones.

This paper gives a formal model of a novel architecture for a globally distributed, survivable and evolvable *application infrastructure*, which we call Survivable Workflow Transaction Infrastructure (SWFTI). It features a transactional programming model, based on atomic execution steps by active *transaction* components that monitor and modify the state of passive *object* components. As an example, a transaction in a workflow application might observe an input queue object, process each item that is inserted into the queue and deposit the result into another queue object. This model is simple yet inherently concurrent, and is well suited to cross-domain application workflow processing. Software can be developed assuming a sequential, failure-free environment and the infrastructure ensures linearizability and failure recovery for all executions, within or across administrative boundaries.

A novel feature of SWFTI is that applications can declare and depend on invariants between data objects, even across domains, while providing non-interference guarantees for independent applications in the shared infrastructure.

The architecture is designed to support continued, correct execution of long-running, distributed applications in spite of failures and attacks. To achieve this, multiple copies of both active and passive component are hosted on server replicas running on independent physical host servers. Replicas engage in an *agreement protocol*, ensuring that all non-faulty replicas in the group take identical steps. SWFTI tolerates the *Byzantine failure* of a bounded number of replicas in a group, which includes both crashes and erroneous behavior such as may result from software bugs or intrusions by external attackers and malicious insiders. This is the strongest fault model possible, since faulty components may behave arbitrarily.

The remainder of the paper is organized as follows. Sections 2 and 3 discuss related work. Section 4 presents a serial (centralized) model of SWFTI. Section 5 gives a model of distributed SWFTI systems, with correctness formulated in terms of the serial systems.

2. Related Work

This section discusses a Byzantine fault-tolerant algorithm on which SWFTI builds, as well as related work in Grid and peer-to-peer systems.

2.1. Fault-tolerant state machines

The Byzantine fault-tolerance capability of SWFTI is built upon CLBFT [6], a practical Byzantine agreement algorithm for for deterministic replicated state machines in asynchronous, distributed systems. Each replica group has $3f + 1$ replicas, where at most f can be faulty [16, 5].

A replica r is *correct* with respect to a component v if it correctly executes its (necessarily deterministic) implementation of v and sends and receives correct agreement protocol messages from enough correct replicas to reach agreement. If a replica is not correct then we say it is *faulty*. CLBFT makes the relatively weak assumption that the delay for message delivery does not grow faster than real time. This assumption is necessary since distributed consensus is impossible with arbitrary message delays [10].

The CLBFT algorithm works roughly as follows: a client sends a request to a designated *primary* replica server. The primary sends a *pre-prepare* message with the request and a sequence number to all replicas, who send a corresponding *prepare* message to one another to ensure agreement on the sequence number. Upon receiving $2f$ prepare messages matching its pre-prepare message, a replica broadcasts a *commit* message to all replicas. Upon receiving commit messages from $2f + 1$ replicas (possibly including its own) a replica executes the request and sends the result to the client. Upon receiving $f + 1$ matching results, the client proceeds using that result value. Authenticity and integrity of messages is protected using cryptographic signatures.

If a client times out waiting for a reply (possibly due to a faulty primary) it broadcasts the request to all replicas, which either reply with the return value (if the operation was already committed) or forward the request to the primary and set a progress timer. If $2f$ replicas time out, all correct replicas switch to another primary in a *view change* operation and process the request using the new primary.

2.2. Grids and peer-to-peer systems

Grid computing [12] and peer-to-peer systems [2] have the common objective of sharing and coordinating use of resources within virtual communities [17, 11]. Grid research has historically been driven by the need of the scientific community to share instruments and distribute large computations and data sets, while research in peer systems was spurred by decentralized file sharing networks.

Grids have traditionally been cooperatively managed by scientific communities of modest size, with implicit trust among participants and only partially automated infrastructure and application configuration. Efforts are under way to scale up and commercialize grids using emerging web services standards [1].

Peer-to-peer systems have mainly been aimed at file sharing and publication, with high scalability, decentralized resource naming and in some cases, availability despite node failures and user anonymity. Many systems use distributed hash table (DHT) algorithms such as CAN [21], Chord [23], Pastry [20] and Tapestry [25], for resource distribution and discovery. These enable efficient decentralized lookup of an object by a key value (e.g., a file name).

Like Grids, SWFTI is designed to support arbitrary application and service types, and like peer-to-peer systems, it is designed to work on global scales with full decentralization. The main objective of SWFTI though, is the provision of a *shared infrastructure for survivable execution of complex, long-running, distributed applications*, that interact with each other through persistent data objects. SWFTI achieves this through Byzantine fault-tolerant replication of both processing and data, and by specifying a robust, transaction-oriented execution model as an integral part of its architecture (in lieu of e.g. communicating processes). Grids, in contrast, are fairly batch-oriented, with limited or ad-hoc application-level support for restarting failed processes and no tolerance of Byzantine faults. DHT-based peer-to-peer networks are decentralized and scalable by design but, to date, mainly support storage and routing of static data and generally tolerate only fail-stop failures, i.e. nodes halting or leaving the network.

The goals of SWFTI also differ significantly from those of traditional distributed operating systems, such as Amoeba [24], Chorus [22], Clouds [9], and the V distributed system [8]. These support a traditional programming model by making a distributed system appear to the user as if it were a uniprocessor, offering a full complement of operating system services. SWFTI in contrast, does not hide the distributed nature of applications. Applications are written in terms of transactions accessing distributed objects. Objects rely on their local hosts for whatever traditional operating system services they require for their implementation.

3. Background: I/O Automata

Since SWFTI targets high-availability and safety-critical applications, we formally model its correct executions. This provides an unambiguous specification of the execution model, facilitating rigorously tested or verified system implementations with high design and implementation diversity. It also permits us to construct formal proofs that implementations satisfy specific properties of the system, increasing our confidence in them. Finally, a formal execution model enables rigorous reasoning about the applications running within the infrastructure, including system services such as security and upgrade installations, etc.

Our model is based on I/O automata [18], so we provide a short review. An I/O automaton is an (infinite) state machine whose state transitions are *actions*. An I/O automaton *signature* S consists of a set of actions, denoted $acts(S)$, partitioned into *input actions*, *output actions* and *internal*

actions, denoted $in(S)$, $out(S)$ and $int(S)$, respectively. Let $ext(S) = in(S) \cup out(S)$ be the *external actions* of S. An automaton a is a tuple (*sig*, *states*, *start*, *trans*, *tasks*), with *sig* an automaton signature, *states* a (potentially infinite) set of states, *start* a non-empty subset of *states*, *trans* a *state-transition relation*, with *trans* $\subseteq$ *states* $\times$ *acts*(*sig*) $\times$ *states* and *tasks* an equivalence relation on $ext(S)$. We abbreviate $acts(sig(a))$ as $acts(a)$, and similarly for *in*, *out* and so forth.

An *execution fragment* of a is a finite sequence s_0, π_1, s_1, π_2, ..., π_r, s_r or an infinite sequence s_0, π_1, s_1, π_2, ... of alternating states and actions such that $(s_k, \pi_{k+1}, s_{k+1}) \in trans(a)$ for every $k \geq 0$. An *execution* is an execution fragment beginning in a start state. An execution α is *fair* if for each task partition C, α is finite and all actions in C are disabled in α's final state or α is infinite and there are either infinitely many occurrences of actions from C in α or infinitely many occurrences of states in which all actions in C are disabled. Let $execs(a)$ and $fairexecs(a)$ be the set of all executions and fair executions of a, respectively.

The *trace* of an execution α of a, denoted $trace(\alpha)$, is the subsequence of α consisting of all the occurrences of actions from $ext(a)$. Any two finite execution fragments α, α' of a where α' begins with the last state of α may be concatenated (less the last state of α) to yield another execution fragment of a, denoted $\alpha \cdot \alpha'$. The occurrence of an action π in an execution or trace is called a π *event*.

An action $\pi \in int(a) \cup out(a)$ is *enabled* in state $s \in states$ if there exists transition $(s, \pi, s') \in trans$, for some state $s' \in states$. Input actions are always enabled by definition, so for every $\pi \in in(a)$ and state $s \in$ *states* there is a tuple (s, π, s') for some $s' \in$ *states*. The actions in $in(a) \cup out(a)$ are called the *local* actions of a, and a is said to be *quiescent* in state s if none of its local actions are enabled in s.

A collection $\{a_i\}_{i \in I}$ of automata may be *composed* to form a new automaton a if the signatures of each pair $a_i \neq a_j$ are compatible, meaning that each internal or output action is under the control of a single automaton. Formally, a collection $\{S_i\}_{i \in I}$ of signatures (indexed by some countable set I) is *compatible* if for each pair S_i and S_j with $i \neq j$ we have $int(S_i) \cap acts(S_j) = \emptyset$, $out(S_i) \cap out(S_j) = \emptyset$ and each action is contained in finitely many sets $acts(S_i)$. The signature of the composed automaton a has $out(a) = \bigcup_{i \in I} out(a_i)$, $int(a) = \bigcup_{i \in I} int(a_i)$ and $in(a) = \bigcup_{i \in I} in(a_i) - \bigcup_{i \in I} out(a_i)$. The states of automaton a are defined as the Cartesian product of the states of its component automata, that is $states(a) = \prod_{i \in I} states(a_i)$. Similarly, $start(a) = \prod_{i \in I} start(a_i)$. $trans(a)$ is the set of triples (s, π, s') such that for all $i \in I$, if $\pi \in acts(a_i)$ then $(s_i, \pi, s'_i) \in trans(a_i)$ otherwise $s_i = s'_i$, with s_i denoting the part of state s "belonging" to a_i. The task equivalence classes of the component automata become the equivalence classes of a, that is: $\bigcup_{i \in I} tasks(a_i)$.

Given an execution fragment α and some set of actions A we define the *projection* of α on A, denoted $\alpha|A$, as the subsequence of α comprised of all adjacent states and transitions π_r, s_r where $\pi_r \in A$. Similarly, for a trace β we define $\beta|A$ as the subsequence of β comprised of all actions in A. The *projection* $\alpha|a_i$ of an execution α of a composition automata a on one of its component automata a_i is defined as $\alpha|acts(a_i)$, with each state s_r replaced by the the state of a_i in s_r. Similarly, the projection $\beta|a_i$ of a trace β of a is defined as $\beta|ext(a_i)$. It can be shown that executions and traces of a yield executions and traces of a_i when projected on a_i, for each $i \in I$. Conversely, given an execution α_i for each $i \in I$ and a sequence β of actions in $ext(a)$ such that $\beta|a_i = trace(\alpha_i)$ for each $i \in I$, there is an execution α of a such that $trace(\alpha) = \beta$ and $\alpha|a_i = \alpha_i$ for each $i \in I$. Furthermore, if β is a sequence of actions in $ext(a)$ such that $\beta|a_i \in traces(a_i)$ for each $i \in I$, then $\beta \in traces(a)$. These theorems enable modular reasoning about executions and traces of composite automata.

4. Serial SWFTI systems

This section describes two formal models of SWFTI in a sequential execution environment, to lay the groundwork for defining correctness for distributed systems. Our base model is a serial system with a fixed set of components. Most information systems are not static, though, but are continually upgraded and and evolved to meet the needs of their users. The second model, therefore, extends the first one to an evolvable system, whose set of components may vary over time. Section 5 will describe the distributed and the replicated SWFTI systems, with correctness defined in terms of the evolvable and distributed system, respectively.

4.1. Components and operations

Objects and transactions are modeled with object and transaction automata, respectively. Let $\mathcal{V}_O$ and $\mathcal{V}_T$ be disjoint, infinite sets of *object identifiers* and *transaction identifiers*, respectively. Let $\mathcal{V}$ denote $\mathcal{V}_O \cup \mathcal{V}_T$. Let $A(v)$ denote the automaton corresponding to v, for any $v \in \mathcal{V}$. We use the letters o, t and v to denote elements of $\mathcal{V}_O$, $\mathcal{V}_T$, and $\mathcal{V}$, respectively, and to denote corresponding automata $A(o)$, $A(t)$ and $A(v)$, when clear from context.

An *object automaton* o is a deterministic I/O automaton that defines one or more *operations* that model, for example, object methods. Each operation has an *operation signature* $f(\vec{P}) : W$, consisting of an *operation name* f, a tuple $\vec{P} = (P_1, P_2, \ldots, P_n)$ of *parameters types* and a *return type* W. We use the dot notation $o.f(\vec{P})$, to denote an operation f and its object o.

An object o may have multiple requests pending. To ensure that request and responses are correctly matched up, o has an infinite set of *request* input actions for each operation f in o and each tuple $\vec{p}$ of parameters for f, denote by $q_o f_i(\vec{p})$ for each $i \in \mathbb{N}$. Similarly, for each request action $q_o f_i$ in o and each possible return value w of f, o has an infinite set of *response* output actions, denoted $r_o f_i(\vec{p}) : w$.

The external actions of each object o are comprised solely of request actions (requests, for short) and response actions (responses, for short). The operations of each object automaton o are partitioned into *accessors* and *mutators*. Intuitively, mutators can affect the future externally observable behavior of o whereas accessors cannot.

Since the internal state of object automata is opaque, we let $\mathcal{D}_o$ denote the domain of *user-view states* of any object $o \in \mathcal{V}_O$. A user-view state is some representation of an object's state that can be stored and passed around in the system. All objects have the accessor $getData() : \mathcal{D}_o$ and mutator $setData(\mathcal{D}_o) : \emptyset$. The $getData$ operation returns the user-view state corresponding to the current state of an object and the $setData$ operation sets the current state of an object to one corresponding to the given user-view state. We say that object o is *in* user-view state d if the $getData$ operation of o would return d. We define that all states resulting from a $setData$ response action are start states of that object. As a corollary, for any reachable user-view state of an object o, there is a reachable state of o which is a start state.

We define a set ∇ of *exceptional return values*, that indicate run-time failures, deprecated operations or invalid parameters and other unexpected conditions that an object cannot handle. An operation whose signature has return type W is allowed to return any value in $W \cup \nabla$.

A *transaction automaton* t is a deterministic I/O automaton that invokes object operations through *operation calls*, consisting of a pair of request and response actions. For each operation f that t might call on some object o and each tuple $\vec{p}$ of parameters of f, t has request output actions, denoted $q_o^t f_i(\vec{p})$, for each $i \in \mathbb{N}$. For each request output action $q_o^t f_i$ in t and each possible return value w of f, t has a response input action, denoted $r_o^t f_i(\vec{p}) : w$.

The external actions of each transaction automata t are comprised of requests and responses, in addition to the input action $create_t$ and the two output actions $requestCommit_t$ and $requestAbort_t$. The $create_t$ action starts the execution of t while the other two terminate it, signalling success or failure, respectively. Transaction t is quiescent before a $create_t$ action and after an $requestCommit_t$ or $requestAbort_t$ action.

A transaction may evaluate a Boolean expression over object accessor return values to determine whether mutating operations should be called. Our definition is general enough to permit various styles of transaction implementations, e.g. using explicit guards and effects or conventional sequential programming constructs.

For each object or transaction automaton $v \in V$, let $requests(v)$ and $responses(v)$ denote the set of request and response actions, respectively, in $acts(v)$.

For any transaction $t \in \mathcal{V}_T$, we say that the set of objects on which t calls accessors only is the *read set* of t, denoted $R(t)$, and that the set of objects on which t calls at least one mutator is the *write set* of t, denoted $W(t)$. Note that $R(t)$ and $W(t)$ may have elements in common.

4.2. Serial, static system

We define correct executions of the serial system, which serve as a correctness condition for the executions of the evolvable system. Since the modeling of the latter requires components to be dynamically created, destroyed and evolved, we proceed as if all possible component automata were a part of the model from the beginning. Only the (finite) set of automata that "exist" in a SWFTI system appear in its executions. Automata that have not yet been added or that have been removed are in a quiescent state, so their actions never appear in system executions. This is more convenient than trying to model the creation or modification of I/O automata, and is the approach adopted in [14, 19], for example.

4.2.1 Serial scheduler

We define for any $U \subseteq \mathcal{V}$ a *serial scheduler* automaton $sched_U$, that *schedules* transactions in U for execution, through their $create$ actions. Essentially, $sched_U$ treats all components in $\mathcal{V} \setminus U$ as non-existent. We wish to support concurrent scheduling of multiple "instances" of a particular transaction automaton. Therefore, when $sched_U$ intends to invoke a transaction $t \in U$, it actually schedules a functionally equivalent *invocation automaton* t_j instead, but the result is the same as if t had been scheduled directly. This also simplifies our definition of execution correctness, since we can project traces directly onto transaction invocations. Scheduler $sched_U$ picks a fresh t_j each time, e.g. by using some j greater than any which $sched_U$ has already used.

For each $t \in \mathcal{V}_T$ and each $j \in \mathbb{N}$ let t_j denote the invocation automaton that is identical to t except each of its actions has been subscripted with j. For example, $create_t$ is renamed to $create_{tj}$. Let $\mathcal{V}_I$ denote the set of all such automata for all $t \in \mathcal{V}_T$.

Operations on objects in U may be requested by the *environment*, which represents operation calls issued by clients (e.g. end-user principals) external to the system. We denote a request or response action of a client e for operation f of o by $r_o^e f_i$ and $q_o^e f_i$, respectively, for any integer $i \geq 1$.

Since our model captures external operation calls, neither invocation automata nor clients share actions with object automata directly. Instead, for each $o \in \mathcal{V}$ there is a

request handler rh_o that mediates operation requests and responses between o and transaction invocations and clients.

For each request action $q_o f_i \in requests(o)$ and response action $r_o f_i \in responses(o)$, rh_o has output actions $q_o f_i$ and input actions $r_o f_i$, respectively, for each integer $i \geq 1$. It also has input actions $q_o^t f_{ij}$ and an output action $r_o^t f_{ij}$, respectively, for each transaction invocation $t_j \in \mathcal{V}_I$. Finally, it has input actions $q_o^e f_i$ and output actions $r_o^e f_i$, respectively, for each external principal e and each integer $i \geq 1$.

When a request action $q_o^t f_{ij}$ or $q_o^e f_i$ occurs in a request handler, it eventually forwards it to o as a $q_o f_i$ event. When it gets a corresponding response $r_o f_i$ it eventually forwards it back to t_j or e as an $r_o^t f_{ij}$ or $r_o^e f_i$ event, respectively.

4.2.2 Serial execution

We use serial, sequential executions as the base model for correctness due to their intuitiveness, simplicity and tractability in formal reasoning.

For any subset $U \subseteq \mathcal{V}$ let U_O denote the objects in U, and let U_I denote the set of transaction invocation automata for the transactions in U. For any finite $U \subseteq \mathcal{V}$ we define the *serial system automaton* AS_U as the composition of the automata in $U_O \cup U_I$, the request handlers of the objects in U_O and the $sched_U$ scheduler automaton, using standard I/O automata composition [18]. To prevent transactions from calling non-existing objects we require that each transaction automata $t \in U$ only refers to objects in U, that is: $R(t) \cup W(t) \subseteq U$.

An execution α_t of a transaction invocation t_j is *well-formed* if $trace(\alpha_t) = \pi \cdot \beta_1 \cdot \beta_2 \cdot \ldots \cdot \beta_n \cdot \pi'$, where $\pi = create_{tj}$, for each $1 \leq k \leq n$ we have that β_k is a trace of the form $q_o^t f_{ij}(p)$, $q_o f_i(p)$, $r_o f_i(p)$, $r_o^t f_{ij}(p)$, for some operation f on some $o \in R(t) \cup W(t)$, parameter list p of f and some integer $i \geq 1$, and $\pi' = requestCommit_{tj}$ or $\pi' = requestAbort_{tj}$.

A *well-formed client call fragment* is an execution fragment α_e such that e is a principal and $trace(\alpha_e)$ has the form $q_o^e f_i(p)$, $q_o f_i(p)$, $r_o f_i(p)$, $r_o^e f_i(p)$, for some operation f on some $o \in R(t) \cup W(t)$, parameter list p and some $i \in \mathbb{N}$.

The steps of different transactions and client calls are not interleaved in a serial system execution. More formally, let a well-formed *call fragment* be a well-formed client call fragment or a well-formed transaction invocation execution. An execution α of AS_U is a *serial execution* if $trace(\alpha)$ is a concatenation of the traces of well-formed call fragments, with action $create_{tj}$ appearing at most once for any $j \geq 1$.

A transaction t is *terminating* if each execution $\alpha_t \in execs(t)$ is finite. Similarly, an object o is terminating if each execution of an object operation is finite, i.e. each execution fragment $q_o f_i \ldots r_o f_i$ appearing in $execs(o)$ for an operation f on o is finite. In practice, we expect all correct components to be terminating.

An execution of a transaction invocation t_j is partitioned into two execution fragments, which we call the *guard* and the *effect*, respectively. We define the guard as consisting of zero or more accessor operations on objects in $R(t)$. We define the effect as begin empty or a sequence of accessor operations on objects in $R(t)$ and mutator operations on objects in $W(t)$, beginning with a mutator operation.

4.2.3 Linearizability and fairness

Although SWFTI admits non-serial *concurrent* executions, we want all executions to be linearizable [15]. An execution $\alpha \in execs(AS_U)$ is *linearizable* if there exists a serial execution $\alpha' \in execs(AS_U)$ such that for each transaction invocation $t_j \in \mathcal{V}_I$ we have $trace(\alpha)|t_j = trace(\alpha')|t_j$. Said another way, the two executions "look the same" to all transaction invocations, so they cannot tell the difference between the concurrent execution and the serial one.

We also want our specification to capture the notion that all transactions get fair turns to perform execution steps and that all pending object operations are eventually performed. We therefore let each serial system automaton AS_U have for each transaction invocation $t_j \in U_I$ an internal action π_{t_j}, which is always enabled. A π_{t_j} event schedules t_j for execution, and we put all the scheduling actions for a particular transaction $t \in U$ in the same task partition of AS_U. Furthermore, we put each request handler rh_o for each object $o \in U$ in a task partition of its own. This ensures that in any $\alpha \in fairexecs(AS_U)$, all transactions and objects get fair turns to execute.

4.3. The computation graph

Intuitively, the connections between transactions and objects correspond to the flow of information and causality in the system; the behavior of a transaction is influenced by the data it accesses and its mutating operations on objects influence the future behavior of those objects.

We say that the components of V induce a *computation graph* $G = (V_O + V_T, E)$; a directed, bipartite graph over V, with its vertices partitioned into V_O and V_T. Each edge $(v_1, v_2) \in E$ represents potential operation calls between v_1 and v_2. For any $o \in V_O$ and $t \in V_T$, E has an edge from t to o iff $o \in W(t)$ and an edge from o to t iff $o \in R(t)$. For any $v \in V$, Let $E_{in}(v)$ and $E_{out}(v)$ be the incoming and outgoing edges of v, respectively. Let $E(v) = E_{in} \cup E_{out}$, the incident edges of v.

Properties of the flow of information in G may be statically analyzed, to prevent information "leaks" and reason about the impact of changes to components, etc. Formally, for any path $p \subseteq E$ connecting two objects $o_1, o_2 \in V_O$, let the corresponding *node path* be the set of components in V incident to any edge in p. Let $paths_T(o_1, o_2)$ denote the

union of all node paths from o_1 to o_2 using only edges incident to transactions in T, for any $T \subseteq V_T$ and let $span_T(o)$ be the union of all such node paths beginning at an object o. A simple property we can state is that if $o_2 \notin span_{V_T}(o_1)$ then information cannot leak within G from object o_1 to o_2.

4.4. Constraints and dependencies

A SWFTI *application* is some subgraph of G, and each object or transaction in V belongs to exactly one application. Let $\mathcal{A}$ be an infinite set of *application identifiers* and for any component $v \in \mathcal{V}$, let $app(v)$ denote its application.

Applications may require invariants to be enforced across multiple data objects, e.g. to ensure that each member of one collection has a corresponding member in another collection. We want the transactions that maintain an invariant to appear to run atomically with transactions that mutate objects involved in the invariant. To this end, we allow such transactions to be designated as *constraints*. Let $constr \subseteq V_T$ be the set of constraints.

We augment our definition of correct serial executions to support constraints. Roughly, if an application $A \in \mathcal{A}$ includes a constraint t and some other transaction or sequence of constraints in A updates an object $o \in R(t)$, then we require that no other transaction of A can witness a state in which o has been modified and t has not yet executed. Moreover, we allow an application to declare the fact that its correct execution depends upon invariants enforced by the constraints of *other* applications. We say that *incidental atomicity* occurs when an application A uses objects updated by constraints in applications that A depends on.

Constraints and dependencies are a powerful mechanism, since the developer of an application can define such constraints without having any control over the implementation of the objects involved and without any knowledge about other applications that may mutate them. Implementations have options for how and when a particular constraint is enforced, i.e. by running it as a sub-transaction of transactions writing to its read set or more lazily, as other transactions later access objects in its write set.

We formally state the correctness condition for serial executions with constraints and dependencies. The acyclic *application dependency relation* includes pair (a_1, a_2) only if application a_1 *depends on* application a_2, which we write as $a_1 \prec a_2$. Let $\prec^*$ denote the transitive, reflexive closure of $\prec$ and let $depends(A)$ denote the set of applications on which A (transitively) depends, the set $\{B \in \mathcal{A} \mid A \prec^* B\}$. Let $constr_A$ denote the set of constraints *relevant* to A, the set $\{t \in constr \mid app(t) \in depends(A)\}$.

Let β be the trace of any serial execution of serial system AS_U, for some $U \subseteq \mathcal{V}$. Let β_s be any subsequence of β with $\beta_s = q_o f_i \cdot \beta' \cdot q_p^t g_j$, where f is a mutator of some object o and g is an operation on some object p and $i, j \in \mathbb{N}$. Then, if $t \in constr_A$ for some application A and $p \in span_{constr_A}(o)$ then either there is no request from t to o in β' or β' contains a $create_{sk}$ action, for each constraint $s \in paths_{constr_A}(o, p)$ and some $k \in \mathbb{N}$.

4.5. Application non-interference

Although SWFTI applications execute in a shared infrastructure, the correct execution of an application should not have to rely on the correctness (safety and liveness) of other, unrelated applications. Yet, certain undesirable interactions may result when linearizability is realized through concurrency control, e.g. when an object needed by one application is locked by a transaction in another.

We want to provide non-interference guarantees for independent applications. Specifically, we wish to guarantee that an application A cannot be *blocked* by an application $B \notin depends(A)$. The idea is that by declaring a dependency on the applications in $depends(A)$, the developer of A has expressed a measure of trust in these applications, whereas other applications remain untrusted.

We say that a SWFTI system guarantees non-interference if for any transaction $t \in A$, if t is terminating and all transactions in $\bigcup_{v_1, v_2 \in R(t) \cup W(t)} paths_{constr_A}(v_1, v_2)$ are terminating and all objects $\{o \in R(t) \cup W(t) \mid app(o) \in depends(A)\}$ are terminating, then all invocations of t will terminate.

4.6. Evolvable, serial system

A serial system automaton AS_U models a static SWFTI system. We now define the *evolvable serial system automaton* AE, that allows system components to be added, removed and changed during the course of an execution.

We define the *evolvable serial scheduler* $sched_\Delta$ as sharing $create$ actions with all transaction invocation automata in $\mathcal{V}_I$. Scheduler $sched_\Delta$ keeps in its state the set of automata that it considers to currently exist and be candidate for scheduling. A component is "created" or "destroyed" by adding or removing it from that set, respectively.

We introduce persistent names for components, so we can "modify" an automaton by mapping an existing component name x to a different automaton. Formally, let $\mathcal{N}$ be an infinite set of *component names* (names, for short). Scheduler $sched_\Delta$ stores a finite, injective *system components* function $\mathcal{N} \mapsto \mathcal{V}$ in its state, relating names of existing component to their automata. Let N and V denote respectively the names and components appearing in the domain and range of that function and let $V_O = V \cap \mathcal{V}_O$ and $V_T = V \cap \mathcal{V}_T$. Scheduler $sched_\Delta$ only schedules transactions in V_T for execution.

We define AE as the composition of all of the automata in $\mathcal{V}_O \cup \mathcal{V}_I$, the request handlers of the objects in $\mathcal{V}_O$ and the $sched_\Delta$ scheduler automaton, using standard I/O automata composition.

Let α be an execution of AE. For any state $s \in \alpha$ let $syscomp(s)$ denote the components function of $sched_\Delta$ in state s. We call any execution fragment $s_i, \pi_{i+1}, s_{i+1}, \pi_{i+2}, \ldots, \pi_{i+n}, s_{i+n}$ appearing in α where for all $i \leq k \leq i+n$ we have $syscomp(s_k) = syscomp(s_i)$ a *stable interval* of α. A similar definition applies to any infinite execution fragment $s_i, \pi_{i+1}, s_{i+1}, \pi_{i+2}, \ldots$ and $i \leq k$. If a finite or infinite stable interval is not a subsequence of any other stable interval of α then it is a *maximal stable interval* of α.

The following claim captures the idea that the executions of AE consist of sequences of normal, linearizable application execution steps interspersed by evolution steps that modify the system components function.

Claim 1: Let $\alpha \in execs(AE) = \alpha_0 \cdot \pi_1 \cdot \alpha_1 \cdot \pi_2 \cdot \ldots \cdot \pi_n \cdot \alpha_n$, where for each $0 \leq k \leq n$ execution fragment α_k is a maximal stable interval of α. (note that each π_k is a *system evolution step* of α, transitioning between states with non-identical component functions). Let s_{k0} denote the starting state of interval α_k of α.

If for each $0 \leq k \leq n$, $V_k = syscomp(s_{k0})$ comprises a valid serial system and action π_k follows a call fragment then α_k is a linearizable execution of AS_{V_k}[1]. The proof is based on concatenation of (serial) executions and the fact that there exists by definition an execution starting in any user-view state of a serial system (Section 4.1).

5. Distributed SWFTI System

This section describes and formally models a possible implementation of an evolvable SWFTI system where components are assigned to replicated servers, whose replicas each execute on a physical host.

There are two main aspects of a distributed SWFTI system. The *logical system* consists of components, applications, constraints and application dependencies. The *configuration* determines how that logical system is partitioned into logical servers and how the servers are replicated and mapped to physical hosts.

More formally, a *distributed SWFTI system* is a tuple $Y = (logsys, config)$, where $logsys$ is the *logical system* and $config$ the physical system. We let $logsys$ be a tuple $L = (comp, app, constr, \prec)$, where $comp$ maps names to components, app a function mapping components to applications, $\prec$ an application dependency relation and $constr$ the subset of transactions in V_T that are constraints. We let $config$ be a tuple $C = (server, reps, host, impl)$, where $server$ is a relation partitioning the components in V, $reps$ maps the partitions to (logical) replica servers, $host$ maps replica servers to host server machines and $impl$ maps components and replicas to software implementations.

[1] All but the last fragment are trivially fair, since they are serial and end in a quiescent state after a response action.

After presenting the system model we describe how its structure is encoded in the state of data objects within the system itself.

5.1. Servers, Replicas and Hosts

We assume a domain $\mathcal{S}$ of named *logical servers* (servers, for short), that we use to partition the elements of V. The $server$ relation of C is a finite function $\mathcal{N} \mapsto \mathcal{S}$, mapping each component name $x \in N$ to the one server in $\mathcal{S}$ that *hosts* component $comp(x)$. Let $comps(s)$ be the set $\{x \in N \mid server(x) = s\}$. Due to non-interference requirements, all the components on a particular server must belong to the same application. That is, for any server s and all $v \in comps(s)$ we require that $app(v) = A$, with A some application identifier in $\mathcal{A}$. The partitioning into servers is thus a subpartition of the partition into applications.

Each server is *implemented* by a non-empty set of *server replicas* (replicas, for short), drawn from a domain $\mathcal{R}$ of named replicas, that can execute software and communicate with other replicas over a network. The $reps$ relation of C is a finite injection $\mathcal{S} \times \mathcal{R}$, mapping each s in $server$ to one or more replicas that implement s, called the *replica group* of s. Since each replica is (a part of) the implementation of at most one server, we let $server(r)$ denote s, for any replica $r \in reps(s)$. Also, let $comps(r)$ be a shorthand for $comps(server(r))$ and $reps(x)$ a shorthand for $reps(server(x))$, for any $x \in N$.

Let the *neighboring servers* $neigh(s)$ of a server s be the set of servers hosting a component adjacent to some component hosted on s, in G. Let the *neighboring replicas* $neigh(r)$ of a replica r be the set of replicas implementing the neighboring servers of $server(r)$.

Although replicas are in most respects the ultimate hosts of servers and components, we define them as being "virtual" machines on physical *host machines* (hosts, for short), drawn from a domain $\mathcal{H}$ of named hosts. The $host$ relation of C is a finite relation $\mathcal{R} \mapsto \mathcal{H}$, mapping each replica r for which $reps$ is defined to the one host in $\mathcal{H}$ that *runs* r. A host may run multiple replicas for different servers, but we assume that replicas on the same machine are completely isolated from one another, e.g. through server virtualization [3]. Let $reps(h)$ be the set of replicas running on h, for any $h \in \mathcal{H}$. Let $hosts(v)$ denote the set of hosts that run some $r \in reps(v)$, for any $v \in V$.

5.2. Implementations

Let $\mathcal{I}_v$ be the set of *implementations* of an object or transaction automaton $v \in V$, and $\mathcal{I}$ the union of all such implementations. An implementation $m_v \in \mathcal{I}_v$ is some representation of v that can be stored and executed on physical hosts.

An object implementation m_o provides implementations of each of the operations of o. A transaction implemen-

tation m_t calls operations on the object implementations corresponding to objects in $R(t) \cup W(t)$. Each execution of a transaction implementation m_t corresponds to the execution of a fresh invocation automaton $t_j \in \mathcal{V}_I$. Implementation m_t implements the operation $create() : \{commit, abort\}$, which resets m_t to its beginning state, makes an arbitrary number of operation calls on object implementations and terminates, returning an indication of success or failure.

We define the *execution* of a transaction implementation m_t as the alternating sequence of operation calls it makes and the return values passed back to it. Similarly, an *execution* of an object implementation m_o is the alternating sequence of operation calls made on it and the return values it passes back.

For any implementation $m_v \in \mathcal{I}_v$ we let $execs(m_v)$ denote the set of all possible executions of m_v, that is: executions of m_v that could result given the right sequence of requests or return values, for transaction and object implementations, respectively. Implementation m_v *correctly implements* component v if for each execution of $o_1.f_1(\vec{p_1}), w_1, o_2.f_2(\vec{p_2}), w_2, \ldots, o_n.f_n(\vec{p_n}), w_n \in execs(m_v)$ there exists an execution in $execs(v)$ with trace $q_{o_1} f_{1i_1}(\vec{p_1})$, $r_{o_1} f_{1i_1}(\vec{p_1})$:$w_1$, $q_{o_2} f_{2i_2}(\vec{p_2})$, $r_{o_2} f_{2i_2}(\vec{p_2})$:$w_2$, $\ldots$, $q_{o_n} f_{ni_n}(\vec{p_n})$, $r_{o_n} f_{ni_n}(\vec{p_n})$:$w_n$, with each i_j some integer for $1 \leq j \leq n$. This strict correspondence is required since CLBFT depends on all implementations of a component behaving deterministically the same.

An object implementation m_o necessarily implements the pair of operations *setData* and *getData*, allowing object implementations to initialize their internal state in accordance with the passed-in user-view state (encoded in some generally agreed way [7] and to return their current user-view state, respectively.

The $impl$ relation of C is a finite relation $\mathcal{N} \times \mathcal{R} \mapsto \mathcal{I}$, mapping components and replicas to implementations.

5.3. System objects

The configuration of a SWFTI system is stored in data objects within the system itself. This simplifies implementation and affords configuration state the same Byzantine fault-tolerant protection as application state. It also serves as a mechanism for *reflection*, allowing applications to observe and modify system state. Such modifications, called system evolution steps (described in Section 4.6), occur through atomic operation calls on system objects, ensuring security and fault-tolerance for evolution steps.

The relations of a system Y are partitioned and mapped onto system objects, in a way that reflects the distribution of SWFTI system implementations. We define the *projection* of system Y onto a server s in Y as a SWFTI system Y_s, whose relations map subsets of the domains of each relation in Y to the same values as the synonymous relations in Y. Specifically, relations $comp_s$, $constr_s$, $\prec_s$, $server_s$, $reps_s$, $host_s$ and $impl_s$ of Y_s are defined respectively for the domains $\{x \mid x \in comps(s)\}$, $\{x \mid x \in comps(s) \cap constr\}$, $\{app(s)\}$, $\{x \mid x \in comps(s)\}$, $\{s\}$, $\{r \mid r \in reps(s)\}$ and $\{(x \in comps(s), r \in reps(s)\}$. For each server s in Y there is a *server object* named x_s, that stores in its state the $comp_s$, $constr_s$, $\prec_s$, $reps_s$, $impl_s$ and $host_s$ relations of Y_s. Server object x_s is hosted on server s, that is $server(x_s) = s$. The $server$ relation of Y corresponds to component data in server objects, that is, $server(x) = s$ iff $(x, s) \in comp_s$.

The implementation of server objects represents the SWFTI replica system software that runs in (a virtualization compartment on) the host. The exception is the implementation of the server objects for host servers, which represent the host's hypervisor software. Since system software is included in configurations, it may be modified using normal system evolution mechanisms.

5.4. Distributed scheduling

In a distributed system each server makes it scheduling decisions autonomously. We fix for each logical server $s \in \mathcal{S}$ a *distributed server scheduler* automaton $sched_s$, that schedules transactions in $comps(s)$ (if any) for execution. Each scheduler shares actions with the request handlers of all objects in V_O, as serial schedulers do. We define the *distributed system automaton* AD of a system Y as the composition of all the automata in $\mathcal{V}_O \cup \mathcal{V}_I$, the request handlers of the objects in $\mathcal{V}_O$ and the distributed schedulers for each server in Y.

An execution $\alpha \in execs(AD)$ is *correct* if there exists a linearizable execution α' of AE such that $trace(\alpha) = trace(\alpha')$ and for each state $e \in \alpha$ there exists a valid distributed SWFTI system Y such that the projection of Y on each server $s \in \mathcal{S}$ corresponds exactly to the relations stored in the state of server object x_s in state e. The latter condition states that the system state encoded in server objects always corresponds to a well-formed system.

Claim 2: If the object request handlers in the system use a standard concurrency control (locking) algorithm CC_Y then each execution of AD corresponds to an execution of a corresponding evolvable system AE. The proof is based on standard concurrency control theory [4] as in [19] then executions of AD will be linearizable.

5.5. Distributed, replicated system

A replicated SWFTI employs multiple replicas of each server and a Byzantine fault-tolerant agreement protocol that synchronizes the state of the replicas so that they effectively simulate a single, centralized server.

5.5.1 Extended CLBFT algorithm

To accommodate the replication of active clients, we extended the CLBFT protocol (Section 2.1) roughly as follows [13]. Let f_c and f_s be the number of faulty client and server replicas tolerated, respectively. The client replicas run CLBFT to agree on the operation to perform as well as a *designated sender* replica, that will send results back to client replicas, obviating the need for all server replicas to reply to all client replicas. The client replicas send their request to the primary server replica, which, if it receives at least $f_c + 1$ matching requests, proceeds to run the operation using CLBFT. The server replicas send their (signed) results to the designated sender, which, if it receives at least $f_s + 1$ matching results, relays them to all client replicas.

If client replicas time out waiting for a reply, they send their request directly to all server replicas, circumventing a potentially faulty designated sender. The clients then agree on a new designated sender during the subsequent operation.

5.5.2 Replica group correctness

We define the correct executions of a replicated system.

We define for each replica r two *replica automata*: a *server replica automaton* As_r for the objects of $server(r)$ and a *client replica automaton* Ac_r for the transactions of $server(r)$. A replica automaton defines the logic for a Byzantine fault-tolerant replication algorithm and controls the input actions of its components. Each replica automaton has independent instances of its components, so we introduce the set $\mathcal{V}^{\mathcal{R}}$ of replica-subscripted components, that has for each component $v \in \mathcal{V}$ and each replica $r \in \mathcal{R}$ an automaton v_r that is identical to v except each of its actions has been subscripted with r. Let $\mathcal{V}_I^{\mathcal{R}}$ be the set $\mathcal{V}_I$ extended in the same way. Note that replica automata are generic, each server and client replica automaton is identical to respectively every other server or client replica automaton, except its actions are subscripted with an identifier $r \in \mathcal{R}$.

A replica automaton A communicates with another replica automaton A' through a pair of *channel automata* $c_{AA'}$ and $c_{A'A}$, for messages from A to A' and from A' to A, respectively. A channel automaton represents an asynchronous communication link for replica protocol messages, that may reorder or duplicate messages but always delivers any message deposited into it to the recipient in a finite amount of time.

We define the *replicated system automaton* AR of a system Y as the composition of the component replica automata for each $r \in \mathcal{R}$, the components in $\mathcal{V}_O^{\mathcal{R}} \cup \mathcal{V}_I^{\mathcal{R}}$ and a pair of channel automata for each pair of distinct replica automata. We say that a protocol message between a client and server replica is *sent* or *received*, respectively, when it is added to or removed from a channel automaton, respectively.

Claim 3: Suppose the server replica automata of AR use the same concurrency control algorithm CC_Y used to ensure linearizability of executions of AD. Then any execution of AR corresponds to some linearizable execution of AD.

We sketch the proof, which is by construction of an execution trace $\beta' \in traces(AD)$ for any execution trace $\alpha \in execs(AR)$. Let $\mathcal{QR}$ be the set of all request and response actions of replica-subscripted components, that is the set $\bigcup_{v \in \mathcal{V}^{\mathcal{R}}} requests(v) responses(v)$. Let $\beta = trace(\alpha)|\mathcal{QR}$. We build β' from an initially empty β' by scanning β, message by message from the beginning, counting protocol messages sent and received for request and responses and appending events to β'. Let $f(v)$ be the maximum number of faulty replicas tolerated by replica group $reps(v)$, for any $v \in V$. Let $|q|_n$ and $|r|_n$ denote the number of messages containing request or response q or r, respectively, in the prefix of β of length n. If, as a result of scanning the n-th event in β, we have $|q_o^t f_{ij}(\vec{p})|_n = f(t) + 1$, we make that the serialization point for the sending of the request, by appending $q_o^t f_{ij}(\vec{p})$ to β'. Similarly, if $|r_o^t f_{ij}(\vec{p}) : w|_n$ or $|r_o f_i(\vec{p}) : w|_n$ reaches $f(t) + 1$ or $f(o) + 1$, respectively, we append $r_o^t f_{ij}(\vec{p}) : w$ or $r_o f_i(\vec{p}) : w$ to β', respectively. Finally, if $|q_o^e f_i(\vec{p})|_n$ or $|r_o^e f_i(\vec{p}) : w|_n$ reaches 1 or $f(o)+1$, respectively, we append $q_o^e f_i(\vec{p})$ or $r_o^e f_i(\vec{p}) : w$ to β', respectively. Note that only a single external request is needed, as external clients are not replicated, by definition.

We cannot, however, use a $q_o f_i(\vec{p})$ message in AR as the serialization point for the reception and execution of the request, since in the replicated system it signifies agreement on the order of the request with respect to other requests, not its execution. We therefore expand $\mathcal{QR}$ to include messages of the form $\langle prepare(q) \rangle_r$, which are sent by a server replica r after it has received $2f$ prepare messages matching its own message for q. This is when the sequence number for the request is irrevokably decided and the request is *prepared*. Though operations are executed strictly in order, they may become prepared out of order. Therefore, suppose that as a result of scanning a message with request $q = q_o f_i(\vec{p})$ for an object $o \in comps(s)$, request q is prepared with sequence number m. Let l be the lowest request sequence number for s for which we have not yet added a request to β'. If now for each $l \leq k \leq m$ request q_k for s with sequence number k is prepared, then we append each q_k to β', in increasing order.

We note that the concurrency control algorithm CC_Y used for AD can also be used for AR [14]. We claim that for any execution $\alpha \in fairexecs(AR)$, our construction yields an execution $\alpha' \in execs(As)$ such that (i) α' is linearizeable; (ii) α' is a correct execution of AD and (iii) $\alpha' \in fairexecs(AD)$. Claim i) holds, since our

construction goes from a linearizeable execution of AR and chooses serialization points such that the operation responses received by each transaction in α' are the same as those received by the transaction replicas in α. To show ii) and iii) we appeal to two properties of E-CLBFT, that are proven in a separate paper [13], namely (a) The composition $\prod_{r \in reps(s)} As_r \times \mathcal{V}_O^{\mathcal{R}}$ of the server replica automata for a server $s \in \mathcal{S}$ simulates the composition $\prod_{o \in comps(s) \cap \mathcal{V}_O} A(o) \times \prod_{o \in comps(s) \cap \mathcal{V}_O} rh_o \times sched_s$ of the corresponding objects and request handlers and distributed scheduler for server s in AD and (b) The composition $\prod_{r \in reps(s)} Ac_r \times \mathcal{V}_I^{\mathcal{R}}$ of the transaction replica automata for a server $s \in \mathcal{S}$ simulates the composition $\prod_{t \in comps(s) \cap \mathcal{V}_t} A(t) \times sched_s$ of the corresponding transaction invocations and distributed scheduler for server s in AD. This enables us to argue that for any interaction between the replicas for a pair of servers $s_1, s_2 \in \mathcal{S}$ our construction will yield an execution in AD where the same interactions take place.

6. Conclusions and Future Work

We have described the architecture of SWFTI, an infrastructure for survivable applications, that uses replicated physical hosts to tolerate a bounded number of Byzantine faulty hosts.

We described its transactional programming abstraction, based on active transaction components with transient state performing atomic operations on passive object components that persistently retain their state. The infrastructure allows replication of both data objects and the active transaction components acting on them, so that fault-tolerance is extended to the whole of an application, not just its state. This is particularly valuable for long-running application that must maintain progress in spite of failures and attacks. System evolution is explicitly captured in the execution model and configuration state is survivably maintained in data objects within the system itself, with evolution steps occurring as atomic object operations.

We defined formally the executions of a serial and static SWFTI system, including fairness and a novel way to specify transitive, inter-application invariant constraints. We defined the correctness of an evolvable, replicated system in terms of tree increasingly abstract models. The replicated system simulates the distributed system, which simulates the evolvable serial system, which simulates a sequence of static serial systems.

We are currently working on a reference implementation of our architecture, to validate its feasibility and performance. We are also working on algorithms for non-interfering concurrency control and for secure, atomic and scalable upgrades of SWFTI applications and system software, as well as the fully decentralized security subsystem underpinning the security of evolution and application processing in general.

References

[1] Globus Alliance Open Grid Services Architecture. `http://www.globus.org/ogsa/`.

[2] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, February 1987.

[5] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS.

[7] M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3):236–269, 2003.

[8] D. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[9] P. Dasgupta, J. R. J. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds Distributed Operating System. *Computer*, 24(11):34–44, 1991.

[10] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. In *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 59–70, New York, NY, USA, 1985. ACM Press.

[11] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. volume 2735, pages 118–128, Jan 2003.

[12] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[13] K. Goldman, S. Pallemulle, and I. Wehrman. Forthcoming. Technical report, 2005.

[14] K. J. Goldman and N. Lynch. Quorum consensus in nested-transaction systems. *ACM Trans. Database Syst.*, 19(4):537–585, 1994.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[17] J. Ledlie, J. Shneidman, M. Seltzer, and J. Huth. Scooped, Again. volume 2735, pages 129–138, Jan 2003.

[18] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, Sep 1989.

[19] N. A. Lynch, M. Merrit, W. E. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, August 1993.

[20] A. R. P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Nov 2001.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[22] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA, 1992.

[23] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[24] A. Tanenbaum, M. Kaashoek, R. Renesse, and H. Bal. The Amoeba Distributed Operating System, 1990.

[25] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan 2004.