Constructing Provenance-Aware Distributed Systems with Data Propagation

by

Ian Campbell Jacobi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2010

Certified by . . . .
Gerald Jay Sussman
Professor of Electrical Engineering
Thesis Supervisor

Accepted by . . . .
Professor Terry P. Orlando
Chair, Department Committee on Graduate Students

Constructing Provenance-Aware Distributed Systems with Data Propagation

by

Ian Campbell Jacobi

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Is it possible to construct a heterogeneous distributed computing architecture capable of solving interesting complex problems? Can we easily use this architecture to maintain a detailed history or provenance of the data processed by it? Most existing distributed architectures can perform only one operation at a time. While they are capable of tracing possession of data, these architectures do not always track the network of operations used to synthesize new data.

This thesis presents a distributed implementation of data propagation, a computational model that provides for concurrent processing that is not constrained to a single distributed operation. This system is capable of distributing computation across a heterogeneous network. It allows for the division of multiple simultaneous operations in a single distributed system. I also identify four constraints that may be placed on general-purpose data propagation to allow for deterministic computation in such a distributed propagation network.

This thesis also presents an application of distributed propagation by illustrating how a generic transformation may be applied to existing propagator networks to allow for the maintenance of data provenance. I show that the modular structure of data propagation permits the simple modification of a propagator network design to maintain the histories of data.

Thesis Supervisor: Gerald Jay Sussman
Title: Professor of Electrical Engineering

Acknowledgments

I would like to acknowledge and thank the following individuals for their assistance throughout the development of the work presented in this paper:

My thesis supervisor, Gerry Sussman, for having introduced me to Alexey Radul's work on propagator networks and advising me throughout the past two years as I have developed this work.

Alexey Radul for developing the concept of propagator networks and offering significant assistance and criticism of my proposed distributed propagator mechanism.

My advisor, Hal Abelson, for providing feedback during the early stages of this work and helping me to refine its goals.

Mike Speciner for helping to proofread the paper and finding several mathematical issues with my description of provenance.

Tim Berners-Lee for his criticisms of my use and implementation of REST within distributed propagation.

Joe Pato of HP Labs for contributing to my understanding of computational provenance and its use-cases.

Lalana Kagal for her continued assistance and support of my work, and helping to place it within the context of accountability and reasoning.

My lab-mates and other members of the Decentralized Information Group not named above for offering their assistance, criticisms, feedback, and proofreading skills as this project has slowly developed over the past two years.

Finally, I would like to acknowledge that funding for this work was provided by the National Science Foundation Cybertrust Grant award number 04281 and by IARPA Grant FA8750-07-2-0031.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

By harnessing the power of tens, hundreds, or even thousands of individual computers, it is possible to solve many problems that might not be tractable on a single system. Such distributed systems are a compelling application of large networks. These systems usually work on a heterogeneous network by design; large networks rarely consist of computers that all share the same configuration of hardware and software. To effectively scale a distributed system, it is usually necessary for the system to perform the same computation on many different host configurations. By partitioning data across the nodes in the network, it is possible to improve the performance of many algorithms. Some algorithms, such as those that feature a high level of interdependency within the data they process, are not amenable to such division of labor. For these algorithms, another approach to their distribution is required.

Distributed systems often maintain only a basic level of history for the data they process. This may include simply logging the particular node to which a piece of data was delegated for processing. Complex data histories may be constructed in distributed systems, but this often requires careful consideration of the role and need for this "data provenance" at design-time. It would be more efficient from a software design perspective if a system was able to handle such provenance without significant modification of existing code.

1.1 Heterogeneous Distributed Systems

As the archetypal large network, the Internet has enabled the construction of many distributed computing efforts. Architectures like SETI@home [3], the related BOINC platform [2], or even Google's internal usage of MapReduce [13] demonstrate the value that the Internet has added to distributed systems. By allowing the many machines connected to the Internet to work together, many otherwise intractable problems may be solved.

Whether on the Internet, as in the case of the former two systems, or in private data centers, as in Google's case, these distributed systems all operate on heterogeneous networks. There is typically little standardization of the hardware and software configurations of nodes on these networks. Even systems that are superficially similar may differ greatly in their actual specifications. The ability to adapt to these differences is a major requirement of a viable distributed architecture.

Distributed systems usually rely on partitioning input data so it may be processed identically on a number of nodes. This data is then distributed across the network for processing. Once this is complete, results are sent to a central server that merges them to produce a meaningful result. Usually, the same algorithm is applied on all nodes in the network. Due to the heterogeneous nature of distributed systems, this may result in compiling as many implementations of the same code as are necessary for it to run on most hardware and software in the network.

Partitioning data is not a practical solution for all problems we might wish to distribute. Highly interdependent data with a high degree of dimensionality is relatively difficult to partition, due to Bellman's classic Curse of Dimensionality [5, 31]. General-purpose reasoning is one such example of a problem that can suffer from the Curse due to sparsity of knowledge and potentially complex interactions of rules. Although small-scale reasoning has been distributed with some success [42], larger, more complex rule-sets may not be distributed so easily. In applying a particular rule, a general-purpose reasoner may depend on a number of logical statements to conclude a result. Any particular partitioning of input data must respect these interdependencies; partitioning data to account for this may be a costly operation. Furthermore, these interdependent facts must be able to fit into a single data partition.

One of the best-known algorithms for forward-chaining reasoning, the Rete algorithm [15, 19], suggests a possible solution to the data partitioning problem. In the Rete algorithm, each sub-pattern of a rule is matched against the knowledge base separate from every other sub-pattern. The results of a particular pattern-match are then merged with other results in a manner that unifies bound variables, and a rule may be applied.

In practice, the process of pattern-matching is treated as an entirely separate operation from that of merging the results of these pattern-matches. Unlike traditional distributed computing approaches, the Rete algorithm effectively divides the *algorithm* into smaller, more tractable parts, rather than the data. By applying this concept to distributed systems we may resolve the data partitioning problem by bypassing the problem altogether! Instead of partitioning data across a network of hosts running an identical algorithm, we might try to partition the algorithm itself across the nodes.

We must be careful not to blindly apply such an approach on an existing distributed architecture as this might prove to be impossible. Existing distributed systems have usually assumed a strong client-server relationship. Although they may have been built with the weak-connectivity of the Internet in mind, systems like BOINC and MapReduce still require an algorithmic client to communicate with a central server upon starting or completing a particular subtask. This is usually a desirable feature, as these systems were constructed with data partitioning in mind. By separating the data into tractable chunks, only the initial data and final results need to be communicated between a client and a server, making such a system quite feasible.

This client-server relationship is not so desirable when we try to partition an algorithm instead. Algorithm partitioning requires communication between nodes performing the different parts of a given algorithm. In order to implement a distributed system that distributes the algorithm rather than the data, we must be careful to use a decentralized model for intra-node communication rather than a centralized one. Different parts of the algorithm may need to receive or send results to each other. Relying on a central server to relay such communications may be infeasible for large networks.

1.2 Retrofitting Provenance

As data partitions are assigned to different nodes in a traditional distributed system, it is important to track how that data was generated and manipulated. Distributed systems must usually be constructed with redundancy in mind; nodes performing a distributed computation may perform an incorrect calculation or fail completely. Thus, most systems, including SETI@home [3], BOINC [2] and MapReduce [13], all maintain at least a minimal level of the history of how data was generated and manipulated, also known as provenance.

This minimal level of data provenance is generally collected to build redundancy and resilience into a distributed system. To defend against nodes that return results for data that they did not actually process, it is necessary to keep track of what data has been assigned to each node. Similarly, it is necessary to know when the data was assigned to a node, so that the server may be able to double check the results from a node or account for a node that may have failed to return its results.

Unfortunately, this provenance may be insufficient for many more complex tasks, as we may only be tracking which node was given what data. Although such basic provenance could be used to derive a more complete history of data in the system, the quality and detail may be unsuitable for more complex provenance use cases.

Complex, non-linear provenance has been integrated into workflow-management software for grid computing [36], but these systems have often been designed by including provenance handling as a major software design requirement. Handling of complex provenance is generally more difficult to add to systems after they have been initially designed. Integrating provenance into the system design may also lead to a less-flexible product, and extensive retrofitting may be needed to add support for data provenance to applications running on distributed systems that were not built to explicitly handle provenance.

A distributed system should ideally require minimal code redesign to retrofit provenance into any application that uses the system. Minimizing the cost of adding provenance would allow end-users and developers to more easily integrate provenance in their systems, especially if provenance handling may be deferred to the system itself. By transparently managing provenance on the programmer's behalf, we may be able to reduce the complexity of applications designed to use the distributed system as well as the number of software bugs encountered when using provenance.

1.3 Introducing Distributed Propagation

One programming model for concurrent systems, data propagation [33], offers a solution to both of these problems. Data propagation is a modular, implicitly concurrent computational architecture that connects state-bearing "cells" with monotonic computational "propagators" that do not discard accumulated state. By distributing the same computational propagator across the various nodes of a network, it is possible to perform traditional distributed computation, where the same operation is performed on all nodes. Unlike traditional distributed systems, propagator networks also enable more complex division of processing, as there are no restrictions on the structure of propagator networks.

Propagation does not depend on centralized coordination of computational resources; it is easy to decentralize a propagator network and perform computation without any further involvement of the host that started computation. Application of several constraints on propagator networks facilitate decentralized synchronization of data across a network. Although the initial presentation of propagator networks assumes a tightly-linked computational network [32, 33], the set of constraints I propose frees us from this assumption. Instead, we need only assume reliable broadcast and knowledge-convergence on a weakly-connected network. This has already been proven by Awerbuch [4] and Demers [14].

Propagators may also facilitate the addition of provenance into existing software designs. Propagator networks are easily extended and may be modified algorithmically due to their strong modularity. We need only add several cells and propagators to handle data provenance within an existing propagator network. There is no need to integrate any additional primitives within propagator networks to handle provenance, nor is it necessary to fundamentally redesign existing software to add provenance to a distributed propagator network. This results in obtaining provenance support "for free" with existing software codebases.

I have constructed an architecture that maintains and executes distributed propagator networks, named DProp.[1] Along with this architecture, I have also built an accompanying library, PyDProp, that may be used to easily add provenance to existing software making use of DProp. Together, DProp and PyDProp may be used to construct any number of interesting distributed propagator networks. I am able to demonstrate its viability not only for distributed processing, but also in adding provenance to existing distributed applications that use DProp.

My protocol for distributed propagation has been designed to build on the network architectural model known as Representational State Transfer, or REST. By basing the protocol design on REST, my protocol for distributed propagation achieves a flexibility that might not be possible in other implementations of such a protocol. Several constraints of a REST-based model not only map nicely onto the model of data propagation, but also lend themselves to the efficient implementation of distributed propagation that I describe.

1.4 Thesis Overview

In this thesis, I describe my proposed distributed propagation and provenance system as follows:

Chapter 2 introduces the concepts of data propagation and the Representational State Transfer architectural design.

[1]DProp is currently available in a Mercurial repository at http://dig.csail.mit.edu/hg/dprop/.

Chapter 3 details the design choices that influenced my implementation of distributed propagation.

Chapter 4 discusses how we may to extend existing propagator networks to handle provenance.

Chapter 5 offers a discussion of some potential issues with distributed propagation and provenance in the system I have described. I also discuss how I resolve these issues.

Chapter 6 relates this work to other work relating to data synchronization and distributed provenance.

Chapter 7 provides a summary of the contributions of this work, as well as potential future work that may be used to expand the distributed propagation system proposed here.

Finally, Appendix A offers a more detailed description of my implementation of distributed propagation and provenance, DProp.

Chapter 2

Background

2.1 Data Propagation

Data propagation is a concurrent programming paradigm proposed by Alexey Radul [32, 33]. Based on the principles of message-passing concurrency, data propagation relies on messages sent over networks of monotonic computational blocks, called "propagators", and state-bearing "cells". Propagator networks perform computation when an update of a cell's state triggers the execution of the propagators that are connected to it. These propagators may then update the state of other cells, causing further propagation.

Although the state of a cell may change, its value may only be assigned once. This may seem to be a contradiction at first glance, but the value assigned to a cell may be a partial value, rather than one that has been fully-specified. This allows for continued modification of its state. When we discuss "changing" a cell's state, or "updating" its contents, what we really mean is that the partial value of a cell has been updated with additional data.

Cells are initially created with an "empty" partial value. This special partial value represents a complete absence of knowledge of a cell's contents and provides no constraints on the value of the cell. A cell with an empty partial value closely resembles an unassigned variable, except that it can be easily identified as such during execution.

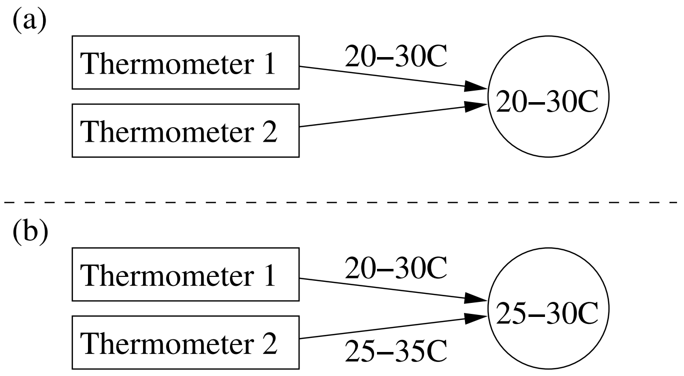

Figure 2-1: A cell applying a merge operation to merge two temperature readings.

Partial values stored in a cell need not be objects with partially defined fields, such as a `cons` pair in Lisp with a `cdr` that has been marked for delayed evaluation [20]. Partial values in a propagator network may, in fact, be any value, including sets and ranges that may be refined through appropriate mathematical operations.

The process of refining the partial value of a cell is dependent on a user-defined merge operation. This merge operation unifies an existing partial value with additional information provided in an "update" message sent to the cell. While the merge operation may be simple assignment when a cell is empty, merge operations are more interesting when they operate on cells that are not empty.

For example, a cell might contain a range of temperature values that represents the current air temperature recorded by a thermometer, including its measurement error. If a second thermometer is connected to the cell and sends its *own* measurement as an update, the merge operation of the cell may take the intersection of the two temperature ranges to obtain a more precise estimate of the current air temperature than either thermometer gives on its own. The cell may then take the intersection as its new partial value, as depicted in Figure 2-1.

A merge operation may be arbitrarily complex. If a cell is capable of performing logical deductions to construct its complete contents, we might be able to employ a merge operation that removes redundant facts for more efficient storage. For example, such a cell might contain two facts: "Socrates is a man" and "Socrates is mortal". If the cell were to receive an update stating that "All men are mortal", it could employ

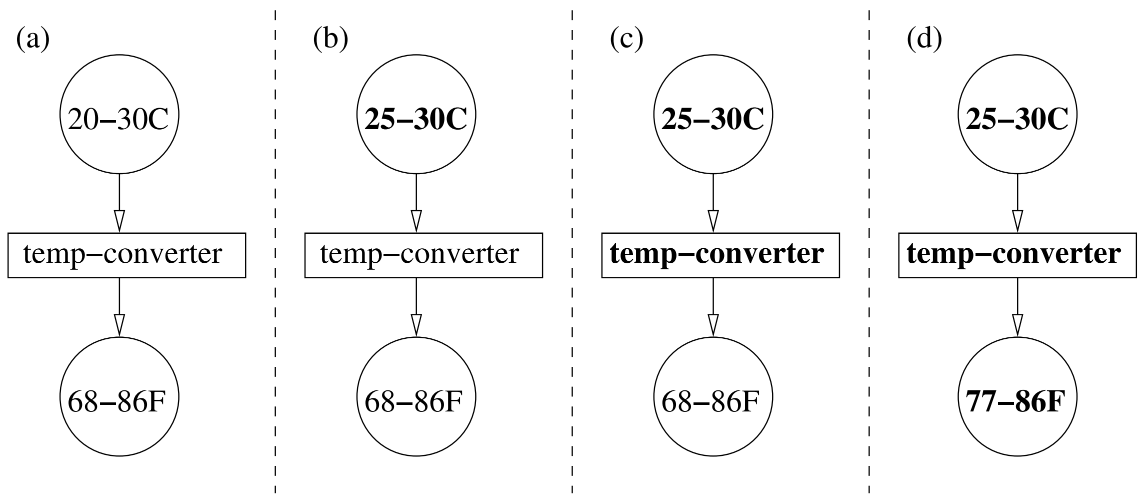

Figure 2-2: Data propagation at work: a temperature converter.

a logically-consistent merge operation that would not only add the fact "All men are mortal" to the cell, but also remove "Socrates is mortal", as it may be concluded from the other two facts. Despite the fact that the facts stored in the cell are not strictly increasing (i.e. the cell is not strictly monotonic), if the cell can still conclude "Socrates is mortal", the actual knowledge that the cell contains is still monotonically increasing.

Once a cell has merged an update and its partial value has changed, propagators connected to the cell wake up and perform computation. These propagators will use the newly updated partial value where they originally used the cell's previous contents. During computation, these propagators may generate updates for *additional* cells. For example, in Figure 2-2 above, an update to the top cell at time (b) causes the `temp-converter` propagator to fire at time (c). Upon receiving these updates, these cells will apply their own merge operations to update their partial data, starting another merge-and-compute cycle. This can be seen above when the propagator at time (c) causes an update to the bottom cell at time (d). These repeated cycles of merging and propagation of data drive computation within a propagator network.

No constraints are placed on the structure of propagator networks. For example, cycles may be used in propagator networks to create standard looping control-flow structures. By allowing a propagator to update a cell that indirectly contributed to its being fired, the propagator may ensure that it fires again to include additional information. Other propagators may prevent the forwarding of updates when the input cells meet some criteria. Such propagators would halt propagator loops, allowing for switch-like control-flow structures and finite loops.

We may also perform recursive computation with propagator networks. As propagators are opaque computational units, and networks of propagators also perform computation, we may define a class of propagators called “compound propagators”. These compound propagators may construct a propagator network when they are first triggered. This new propagator network may then act as a part of the original network, and may perform computation based on the original input cells of the compound propagator by mapping those input cells appropriately to the newly constructed propagator.

Constructing such networks on demand permits recursive computation; a compound propagator need only include itself as one of the propagators in its constructed sub-network to do so. Propagation of data may cause as many recursive expansions of the propagator as necessary simply by repeatedly waking each recursive copy of the compound propagator in turn.

Ordering of propagation events is undefined, outside of the implicit cause-effect chain caused by the propagate-and-merge cycle. Furthermore, propagator networks feature a strong modularity of computation that is enforced by strictly separating cell state from propagator computations. Together, these properties make data propagation a reasonable candidate for parallel processing and distributed computation.

2.2 Representational State Transfer

Representational State Transfer, or REST, is an architectural style for network-based software architectures based on the principles of hypertext and the HyperText Transfer Protocol (HTTP) and developed by Roy Fielding [18]. Architectures that make use of Representational State Transfer, also called “RESTful architectures”, are designed to store and maintain opaque data objects, called “resources”, that represent an abstract concept or another, more concrete, data object. Resources generally have four operations that may be performed on them: creation, retrieval, update, and deletion. Together, these four operations are known as CRUD operations.

Since an abstract concept associated with a resource may not be easily serialized, RESTful architectures are designed to be platform- and content-agnostic through the use of resource "representations". For example, a data set of hourly temperatures may have an image representation in the form of a graph in GIF format, as well as a tabular representation in plain-text. The four fundamental CRUD operations may be performed in a RESTful architecture by using any number of these resource representations to specify the relevant aspects of that resource.

More complex operations on resources, such as those often implemented as remote procedure calls, may be modelled in a RESTful architecture by applying the fundamental resource operations with appropriate representations. By restricting the number of fundamental operations and allowing the use of different resource representations, a RESTful architecture reduces the complexity of software implementations that use it. Software uses of a RESTful architecture need only be able to perform the basic operations with representations they are prepared to handle; there is no need to support all representations of a resource, only those relevant to the application.

In the doctoral thesis that first laid out the principles of REST, Fielding defines an architectural style as "a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to this style." In accordance with this principle, a proper RESTful architecture adheres to five constraints:

1. REST assumes a *client-server model* of communication. This separates the resources on the server from the software that modifies them. It also determines how data may be stored; only the server must find a suitable intermediate format for storage. Clients may use any representations they find appropriate.

2. RESTful designs are *stateless*. Information transmitted between the client and server of a RESTful architecture should be sufficient to perform the desired

CRUD Operation	HTTP Method	RESTful Meaning
Creation	PUT	Create a new resource based on the representation presented.
Retrieval	GET	Fetch the desired representation of an existing resource.
Update	POST	Modify an existing resource based on a given (partial) representation.
Deletion	DELETE	Delete an existing resource.

Table 2.1: The four CRUD operations and corresponding HTTP methods.

operation without maintaining any additional state (such as session variables) on the server or client.

3. Representations of resources transmitted as part of an operation should be *cache-able*. This constraint is particularly useful in terms of the retrieval operation. By caching information that has been previously retrieved, future requests may be optimized by referring to a nearby cache, rather than waiting on the central server. It also reduces the burden on the server, as many requests may be serviced by a cache instead.

4. RESTful designs should employ a *uniform interface*. By standardizing the interface by which resource operations are performed, many different client and server implementations may be deployed in an inter-operable way. A uniform interface also reduces client and server complexity.

5. Finally, RESTful architectures are *layered architectures*. This allows for the separation of network administration from the application itself, providing increased scalability. Caches, back-ends, and proxies may be implemented on the network layer without restricting the functionality of the application itself.

Most REST-based applications use the Hypertext Transfer Protocol (HTTP) as the underlying network protocol of their RESTful architecture. HTTP handles data transfers as a series of requests for data and responses to those requests. Each request may be made for a resource identified by its Uniform Resource Locator (URL). These requests are also associated with the HTTP method corresponding to the operation desired. Four HTTP methods are usually used to implement REST design: PUT, GET, POST, and DELETE. These methods are aligned with the CRUD operations of creation, retrieval, updating, and deletion, as outlined in Table 2.1. The PATCH method [16] may also be used in place of the POST method to perform updates.

I have chosen to implement a propagator network by employing RESTful techniques for several reasons:

1. Propagator networks need minimal modification to adhere to the five constraints of REST. The stateless constraint in REST is stricter than the computational monotonicity of propagators in propagator networks. This means that it is possible to model RESTful communication as a propagator in a propagator network. Furthermore, the implicit atomicity of update messages sent to cells makes these messages well-suited to being representations of resources in our implementation.

2. If we assume that update messages are the representations being transferred, it seems possible to consider cells to be the resources within the design. Just as cells are the only state-bearing objects within propagator networks, resources are the only state-bearing objects in a RESTful architecture.

3. Enforcing uniformity of interface in an implementation of a distributed propagator network creates an extensible system. My implementation of distributed propagation, DProp, is built using Python and has been tested for use on GNU/Linux systems. By requiring that DProp use standard HTTP for cell synchronization, it is possible to construct a heterogeneous propagator network that has nodes that run various different operating systems and propagator implementations. This makes DProp a suitable platform for distributed systems, which already assume a heterogeneous network.

Together, these points make Representational State Transfer a desirable architectural design on which to build our system for distributed propagation.

Chapter 3

A Design for Distributed Propagation

There are several different ways we may construct a distributed propagator network, based on where one draws the boundary between each node on the physical network. For example, we may draw boundaries between cells and the propagators that depend on them, which would imply that cells would notify propagators on other hosts. We could also draw boundaries between propagators and the cells they update, which would imply that propagators would send update messages to other hosts.

Regardless of how the propagator network is divided, propagators and cells *must* reside on the nodes of the network, and not in the network itself. The physical infrastructure of a network is incapable of performing computation or storing content; it may only serve as a web of links across which data may be transferred. As a result, only the edges between propagators and cells may be placed on the physical network.

Although we cannot place propagators in the network, we may still construct propagators that contain a network link within them. This network "bridging" may be derived from our understanding of compound propagators. Just as a single compound propagator "contains" a new network of propagators and cells, we could easily think of the inverse, in which a collection of propagators and cells is turned into a compound propagator. Since the edges that connect propagators and cells may be placed across a physical network, we may construct a compound propagator that includes such an edge, effectively "bridging" the network. As a result of this, we may also assume that *any* propagator may cross the network, as compound-propagators are normally opaque.

My approach to constructing distributed propagation framework builds on this basic concept of propagators that bridge the network. I treat HTTP communication as part of a network-spanning propagator designed to maintain identical copies of a cell on multiple hosts. I do this by intercepting update messages sent to a local cell as part of its merge operation and broadcasting the intercepted messages to all other hosts with a copy of that cell. I then may ensure convergence of this mechanism by occasionally attempting to re-synchronize all remote copies of a cell to ensure that no updates were lost on the network.

3.1 Constraining the Merge Operation

Before we may consider our specific implementation of this approach to distributed propagation, we must first determine exactly what costs are incurred by simply copying the update messages and sending them across a network. Implementing propagation over a network introduces uncertainty in the order of updates and lacks guaranteed message reception. Unfortunately, propagator networks may have issues with messages that are in the wrong order, and the behavior of these networks assumes that messaging is reliable. As a result, propagation-based computation may not be deterministic over a computer network without ensuring that certain properties are present in the data propagation model. In order to remedy these issues and ensure deterministic computation, we must enforce four properties on the merge operations used by cells that receive messages from other hosts:

1. **Idempotence:** If we already know a fact and are told it again, we do not know anything new. Thus, a cell's contents should appear the same no matter how many times a particular update is received, provided that it has been received at least once.

2. **Monotonicity:** A cell never forgets. This means that a cell may never "un-merge" an update. A cell may be able to mark the results of an update as "contradictory" or "disproven", as both changes would refine the content of the cell, but it should not be able to undo the original update.

3. **Commutativity:** A cell's contents should be independent of the order of the updates that were merged to create them. If we are told "it's a warm day" and "it's a sunny day", we will know both of these facts regardless of the order in which we are told.

4. **Associativity:** A cell that merges two messages should have the same contents as a cell that merges a single message that is the "sensible merge" of the two messages. For example, a cell that is told "$T > 40°$F" and "$T > 50°$F" should appear the same as if it had only been told the latter, as the potential values of T implied by $T > 50°$F are a subset of the values of T implied by $T > 40°$.

These constraints do not impose any restriction on the number of *effective* merge operations that may be implemented; merge operations that do not adhere to one of these constraints may be modified so that they do so. For example, a non-idempotent operation may be "retrofitted" to become idempotent by adding a unique identifier as part of each update. If this unique identifier is stored with the data following the merge, a propagator reading the cell may modify its "perception" of a cell's contents based on the number of distinct identifiers of the otherwise duplicate updates.

Operations lacking the other constraints may be modified similarly. Extensive research in the fields of computer networking and distributed databases has attempted to meet the need for serialization of communications (implying non-commutativity and, indirectly, non-monotonicity) despite the lack of guaranteed message reception offered by Internet protocols.[10, 7, 39, 37] Packets on the Internet may be received out of order or even not at all. Any application desiring serialization of messages, such as distributed databases, must be able to deal with the commutativity of messages on the Internet. We may apply the results of this research in a more general fashion to enforce commutativity constraints on non-commutative functions.

3.2 Choosing a Distribution Application Model

Like many other network applications, distributed propagators could be implemented using one of two primary distributed application models:

1. Cells could be managed using a *client-server model*. In such a model, a particular cell would be maintained by a single host, the server. All other hosts interested in the cell would be clients of the server.

   As the maintainer of the cell, the server would be the only host to decide whether or not an update is accepted. It would also be responsible for merging all data into the canonical version of the cell that it stores and would update all interested clients when the contents of the cell changed. As clients would not have local copies of the cell, any client that needed to read the cell's contents would need to request the server's canonical version. Although clients are able to cache the contents of the cell, these caches may only contain static content; they may not be modified, as only the server controls what is merged into the cell.

   Implementing a client-server model is easy when compared to the complexity of the peer-to-peer model described below, but it is highly reliant on a single server to perform merges and to service other requests. Failure of the server could potentially cause a cascading halt to all computation in the propagator network, as propagators would no longer be able to modify the contents of the cell, and, without a cache, no propagator would be able to read the contents of the cell either.

2. Cells could alternately be managed using a *peer-to-peer model*. Every cell would be duplicated across all hosts interested in the cell, called the peers of the cell. Unlike the client-server model, no single host would have special status. Rather than allowing one server to handle updates and merge the cell, each peer does so for its *own* copy of the cell. As long as all peers implement the same merge operation on the cell and may reliably forward any update messages generated

by local propagators to remote peers, an update should eventually reach all hosts and the cell's contents should converge across the network.

A peer-to-peer model requires a more complex implementation than the client-server model; there are no explicit constraints on the network topology of a peer-to-peer system. In contrast, a client-server model enforces a star-like topology on the network. In exchange for this additional complexity, peer-to-peer systems have a robustness that is not present in a client-server model. In a peer-to-peer model, a node failure need not halt all computation. As each peer is in charge of maintaining its own copy of the cell, the peer may continue updating and referring to that copy even if that peer is completely isolated from the remainder of the network by a node or network failure.

Choosing a peer-to-peer architecture does not mean that we must abandon a RESTful design for our application. Although REST assumes a client-server model, a peer-to-peer distributed propagator network may be viewed as a virtual network that operates *above* traditional RESTful communications. Rather than abandoning the client-server restriction of REST, each peer effectively runs separate client and server applications that happen to share the same stored resource, but do not interact at all in any other respect. This is not all that different from how a representation of a resource in existing RESTful systems may be generated in part by fetching an external resource.

It is important to note that neither model requires timely communication in a distributed propagator implementation. The commutativity constraint on the cell merge operation, as described in Section 3.1, ensures that the contents of the cell will be the same, regardless of the order in which messages are received.

This lack of a timeliness constraint may seem strange because it allows global inconsistencies between the cells being synchronized. Propagator networks admit such inconsistencies by design. Progress can still be made on a partitioned problem with only local consistency, and each individual global inconsistency will eventually be resolved. Ensuring ultimate convergence is up to the programmer.

Rather than ensuring timeliness, we need only guarantee that any update will *eventually* arrive at every other cell in the network. That said, some timeliness guarantees may be appropriate in practice to ensure that results are generated in a reasonable time-frame.

The choice between a client-server model and a peer-to-peer model is ultimately a tradeoff between system complexity and fault tolerance. As we seek to build an architecture that is comparable to existing distributed systems, we must consider that, as the number of nodes in a network increases, so do the chances that at least one node will fail. This means that fault tolerance is likely to be of greater benefit than simplicity of the system's design. As a result, I have chosen to implement distributed propagation using a peer-to-peer model, rather than a client-server model.

3.3 Resource Representations

Now that I have established the basic distributed application model to be used for distributed propagation, we must now examine propagation with respect to the architectural model of REST. As resources and their representations are the central focus of a RESTful design, it is important to identify the types of resources and representations used in the system. As mentioned previously, there are two primary components of the propagator paradigm that may be candidate resources in a RESTful implementation: cells and propagators. One additional construct of propagation, that of "update messages", may be a third possible resource type. Finally, as a peer-to-peer system must be able to keep track of the neighboring peers of a node, we may also consider this collection of peers to be a fourth potential resource type. I examine each of these potential classes of resources in turn.

Propagators are the class of object least suited to being turned into a resource. Although we may be able to represent a propagator using a basic description of a propagator, this seems to be of limited use within distributed propagation. Outside of enabling introspection of the propagator network, there is not much need to actually manipulate propagators themselves. As propagators are opaque computational blocks

Relative URL	Description
`/Cells`	The collection of all distributed cells on the server.
`/Cells/{UUID}`	The distributed cell identified by *UUID*.
`/Cells/{UUID}/Peers`	The collection of peers of the cell identified by *UUID*.

Table 3.1: The URL structure used by DProp.

rather than entities that may be directly manipulated, it seems more appropriate to not include them as resources within the system.

Cells seem to be a far more appropriate class of resources in a RESTful distributed propagator design. Not only do they bear state, as resources do, but they are the *primary* source of state in a propagator network. Cells are also more closely linked with communication of data than propagators are, as they are refined by way of updates that are sent to them. Together, these facts suggest that we should represent cells as resources. Cell updates, on the other hand, are tied to a specific cell and have no modifiable state. Although we could model updates as a resource associated with the cell itself, it seems more appropriate to consider them to be a "partial" representation of a cell, due to their use in actually modifying the cell's state.

Peers also appear to be an appropriate class of resources in our system. In order to maintain a peer-to-peer network, it is necessary to have as much redundancy of communication as possible to route around temporary network failures. This means there must be a mechanism for *node discovery*. I do not address the issue of discovery of the initial node and treat it as out of the scope of this thesis. Instead, when I refer to node discovery, I mean the discovery of additional nodes already present on the peer-to-peer network. To enable such discovery mechanisms, we must have a way of representing the peers in a network, and we thus consider peers to be another set of resources in our system.

3.4 Naming Cells

The issue of cell naming in our peer-to-peer RESTful model is not a simple one. A RESTful architecture identifies resources using some globally-unique identifier. In

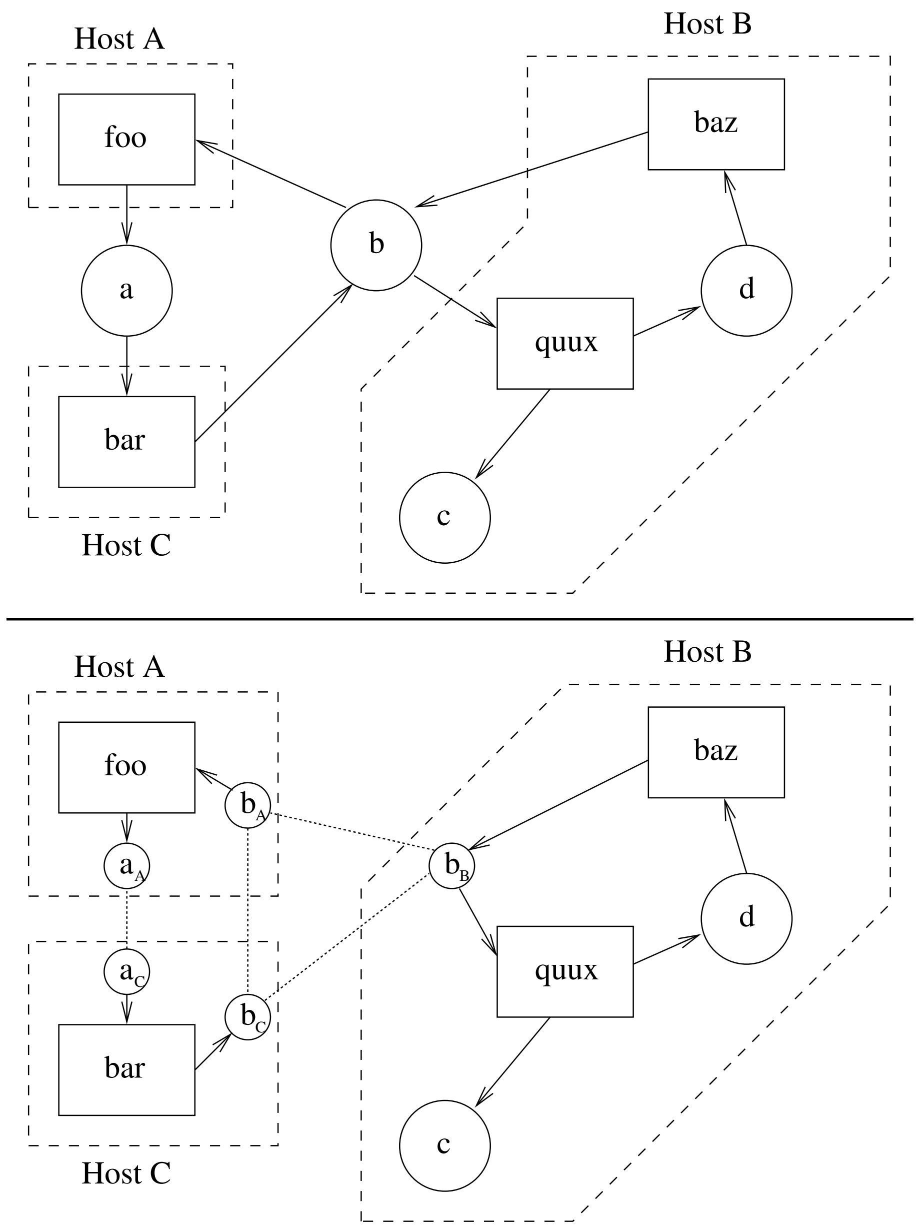

Figure 3-1: Distributing propagator networks by dividing shared cells.

implementations that use HTTP, this identifier comes in the form of a URL, or Uniform Resource Locator.[6] URLs provide for such identifiers by concatenating a globally unique server name with a unique name of the resource local to the server itself. URLs allow us to construct a hierarchical ordering of resources, as they are built on the concept of "paths" on a server. We use such a hierarchical structure to associate the collection of a cell's peers with that cell, as described in Table 3.1.

A complication arises when we need a way to identify multiple copies of the same resource, such as when we wish to identify all copies of a cell that has been copied to multiple peers. When I discuss distributing a propagator design, like that depicted at the top of Figure 3-1, we end up creating copies of cells for each host, and synchronizing them, as at the bottom.[1] In practice, the URLs for each copy of a cell identify the *specific* copy of a cell (e.g. a_A or b_C); they do not identify which original cell (e.g. a or b) a cell copy is associated with.

A naïve solution that resolves this problem relies on having, *a priori*, a globally unique identifier for the collection of cells as a whole. Although we could use a URL for this identifier, we may not have a server name prior to constructing the cell; we may be creating the cell on a host that is temporarily disconnected from the network. Instead of using URLs then, I choose to use universally unique identifiers, or UUIDs [23], to identify each collection of cells. These UUIDs are meant to be globally unique, like URLs, but they do not require a namespace to be associated with them.

We integrate these UUIDs into the URLs we construct to identify each copy of a cell. Although this is not strictly necessary, as we could construct a mapping from URLs to UUIDs, this eases the task of uniquely identifying cells within the server namespace. It also allows for the easy derivation of the UUID of a cell from the URL of a copy of a cell. We use this approach when initializing a cell.

[1]In Figure 3-1, synchronized cell copies are depicted by clusters of cells connected by dotted lines.

3.5 Maintaining the Network

As mentioned in Section 3.3, we consider peers to be resources in a RESTful architecture, so that we may maintain the peer-to-peer network using HTTP. By tracking the peers that a particular cell has been copied to, a particular peer will know which other peers hold copies of a cell that it must forward updates to. Making the collection of known copies of a cell available via HTTP also allows new peers to easily connect to an existing network by simply adding their copy to the set of known cell copies on an existing peer with a simple POST or PUT request. They may also bootstrap their knowledge of copies of the cell by requesting the set of copies from the existing peer.

The set of peers that hold copies of a cell may be independent of those holding any other cell in the propagator network it belongs to. This means that the set of peers is only dependent on the cell itself. We may model the collection of peers as a single resource named "Peers" associated with each cell. In fact, as long as we do not need to modify each peer itself, and only need to know their URLs, we may model the collection of cell copies as a single resource, merely updating the list of cell copies that make up that resource as needed.

Each peer is responsible for maintaining its own list of other copies of a cell; it may choose to keep track of any number of copies of the cell. Thus, the topology of the peer-to-peer network of cell copies may take any shape. Some topologies may be more preferable than others. Topologies that are eventually connected are required to ensure the convergence of cell contents and that all propagators will fire; cell contents may not globally converge if update messages cannot propagate to all peers sharing a cell. Increasing the number of peers visible from a single node is also preferable: network diameter will decrease, and update messages will be more timely. It will also increase fault tolerance, as there is added redundancy of the update messages sent out. For this reason, I choose to maintain a clique in my model of distributed propagation.

By assuming that all peers are connected in a clique, we gain several benefits. As we may assume that every node is a distance of 1 from every other node, we may

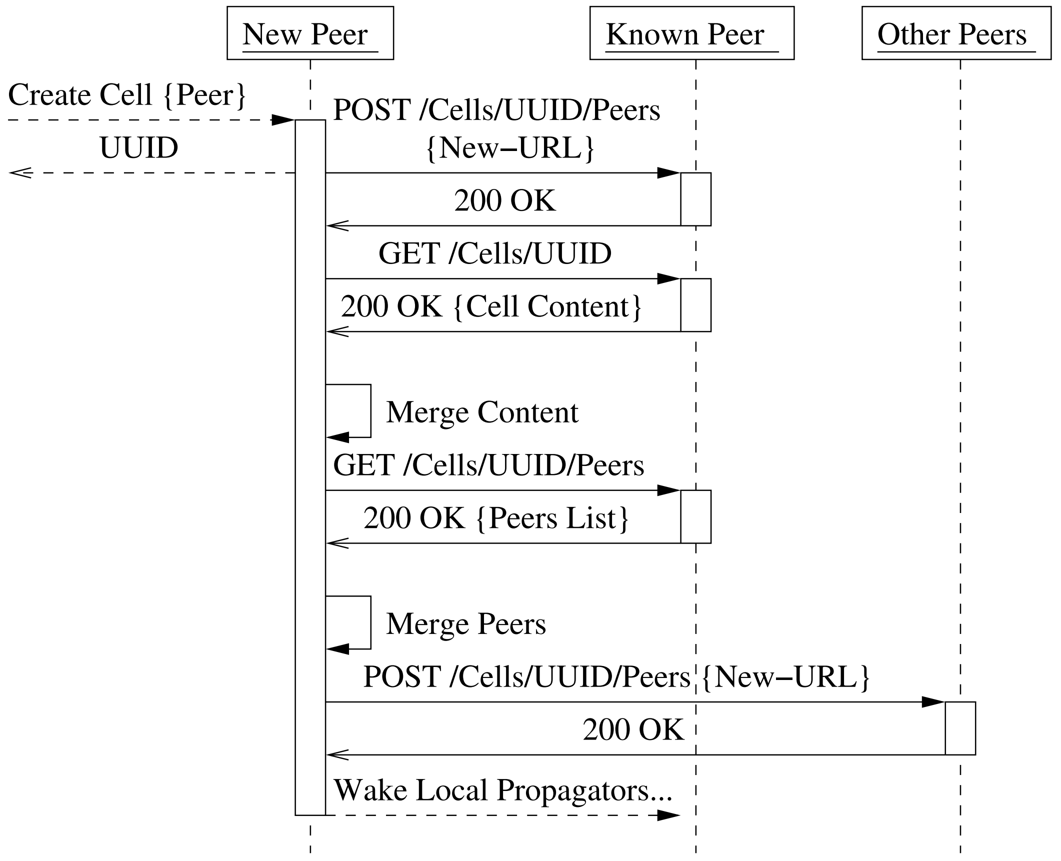

Figure 3-2: Creating a distributed propagator cell.

avoid implementing message routing mechanisms in our network. In a clique, sending a message to all neighboring nodes means that we have forwarded an update message to *all* nodes. However, even if temporary network fluctuations result in a network topology that is not a clique, I also implement a synchronization mechanism that will still ensure global convergence. If a clique is not used with the algorithms I use for distributed propagation, it is not possible to guarantee timeliness of message arrival for nodes that are farther than distance 1 from the original node.

3.6 Cell Initialization

Now that I have established the basic architectural guidelines for distributed propagation, how can we actually construct a cell and initialize it? Initialization of the cell is actually quite simple, and is outlined in Figure 3-2.

Cell creation begins much as it would in a local propagator network; storage is allocated for the cell and the cell is associated with the appropriate merge operation. Once this is complete, we must associate the cell with its appropriate distributed network, creating it if necessary.

If the cell is a completely new cell for which no network exists, we must generate a new UUID for the particular collection of cells, if we are not given one. We then associate this UUID with the cell. Finally, we make the cell available as an HTTP resource by assigning it an appropriate URL, derived from the cell's server-based namespace, appending its UUID as in Table 3.1. Local updates may be merged as soon as the cell is initialized, before its UUID is assigned. Remote connections are only possible once the URL and the UUID have been associated with the cell.

If we are instead joining an existing collection of cells, we must provide the URL of at least one of the existing cell copies in that collection. Creation of a cell from a remote copy resembles the process of creating a local cell, except that the UUID is specified as part of the URL provided as an argument. Once the cell has been made available to remote hosts, we then submit an HTTP POST request to the "Peers" collection of the cell initially passed to the constructor. This POST request contains the newly minted URL of the new peer. It effectively adds the peer to the network so that it may receive updates from the remote cell copy.

We then submit an HTTP GET request for the cell resource itself. In requesting the cell, we desire a representation of the cell's contents as if it was an update message. This allows us to merge the content of the remote cell copy with any content present in the newly created cell. Once this merge is complete, local propagators may fire and begin computation, as with any other merge operation.

We finish the initialization process by initializing the list of cell copies. We submit an HTTP GET request for the "Peers" collection of the initial remote cell and then set the contents of the local "Peers" collection to the contents of the remote cell. This makes the local cell aware of all other cells in the network (assuming that the network is currently a clique). Finally, to ensure that the network is fully repaired to a clique again, the cell submits HTTP POST requests to each peer in the newly initialized

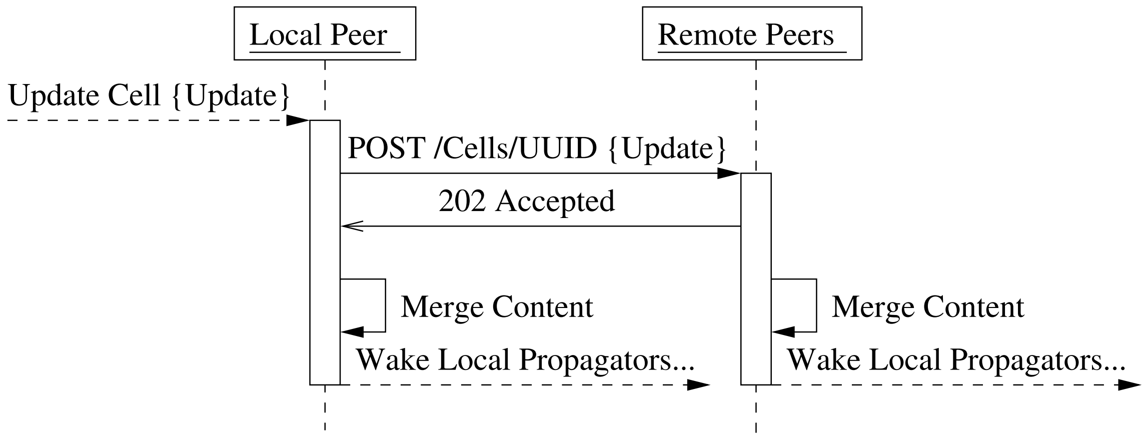

Figure 3-3: Updating a distributed propagator cell.

set of "Peers", adding its own URL to the collection of peers of that copy of the cell. At the end of initialization, the new peer is aware of all peers in the network, and all peers in the network are now aware of the new peer.

Due to the associativity and idempotency constraints that we have placed on the merge operation of the cell, we need not "lock" the cell before we have merged the contents of the remote cell or learned its set of peers. We may begin processing without any data whatsoever and update the cell before synchronization with the initial remote cell has taken place, as the contents of the cell are guaranteed to be monotonically increasing as well! As the merge from the "initialization" will still cause local propagators to fire, we need not preclude computation before the cell becomes non-empty. There is *one* benefit to allowing the cell to synchronize as soon as possible, because any updates merged before the cell has fully connected to all peers will not propagate over the network in a timely fashion.

3.7 Cell Updates

The process of updating a distributed cell is superficially similar to the process of updating a cell on a non-distributed propagator network. The cell has a merge operation that takes the particular update and determines the new contents of the cell. In a distributed cell, an unmodified merge operation that would have been applied to a cell on a non-distributed propagator network is used to complete a generic stub method which will serve as the actual merge operation of the cell. This generic stub copies the received update and forwards it to all other copies of the cell by sending a POST request to all other copies present in the local "Peers" collection. In effect, the merge operation intercepts the update before performing the local merge. By forwarding the update to all other peers, we may delegate the merge to each peer, rather than assuming that the *local* copy of the cell controls which updates are *globally* merged. This may be seen in Figure 3-3.

A remote peer that receives an update in a POST request also performs a merge, but uses a different stub method. Instead of forwarding the update to all other cells, this stub method prevents rogue updates by confirming that the sender is actually a peer before merging the update. Aside from this action, the stub method behaves much as the non-distributed merge would: it wakes up local propagators to perform computation, just as if the update had been received from another local propagator.

Some of the previously mentioned constraints on the cell merge operation prove useful here. If we ensure that the basic cell merge operation is the same for all copies of a cell, we need only forward the update, rather than the new contents of the cell. This may reduce the amount of bandwidth used by a cell update operation. Furthermore, the commutativity constraint relaxes the need for timely updates or a strict global ordering of operations. Instead, we may receive two remote updates in any order and not be concerned that we will lose synchronization with any other peer.

3.8 Failure Recovery Through Re-synchronization

Although the proposed system appears to be relatively robust, we must define one more operation to ensure that the architecture is truly fault tolerant. As described thus far, distributed propagation may suffer from the loss of a link between two copies of a cell as this may lead to a loss of synchronization when messages sent by one copy are not received by the other. If the system uses HTTP over TCP as its underlying communication protocol, we may gain the benefits of the fault tolerance mechanisms of TCP and reduce the chances of such de-synchronization. Unfortunately, simply introducing TCP is not enough to ensure that network failures do not impact the integrity of the content of a cell, and it may not guarantee global consistency.

Instead, we achieve these goals by implementing a recurring operation to re-synchronize the contents of the local cell with remote copies of that cell. This operation sends a GET request to each remote cell, requesting a representation of the cell as if it was an update. It then applies the response to the local cell as if it was an update from the remote cell. Because of the idempotency and associativity constraints on the merge operation, we do not need to concern ourselves with the fact that the representation of the cell as an update may include the effects of updates that have already been merged into the local copy of the cell. Similarly, we need not worry that the update representation might not explicitly include updates that had no substantive effect on the cell's contents.

We may also re-synchronize the list of peers in the network in the same way; this ensures the maintenance of the clique. As the list of peers is not a cell, care must be taken to correctly merge the list of peers. We merge the contents of the remote "Peers" collection into the local collection by taking the union of the two collections. This ensures that the list of peers is monotonically increasing. Unfortunately, this means that this system is unable to *remove* peers from the collection. This is not currently a significant issue; propagator networks do not currently permit the destruction of existing parts of the propagator network.

Performing these occasional re-synchronization operations may consume more bandwidth than is necessary, especially if there are no global inconsistencies. As we transfer the entire contents of the cell as an update, if a cell is rather large, we may transfer a significant amount of data, even if the update results in no change to the local copy of the cell. We may resolve this problem by making use of the *cacheability* constraint of RESTful architectures. HTTP offers several methods to control client-side caching; these methods may be controlled through manipulation of the headers of HTTP messages, collections of meta-data that are sent with each HTTP request and response.

Each HTTP response to a GET request may include an appropriate entity-tag, or ETag. These ETags uniquely identify a particular state of a resource, much as a hash of its contents does. This ETag may then be supplied back to the server in subsequent GET requests by using the If-None-Match header. If the ETag of the current state of the resource on the server matches the one provided in the If-None-Match header, the server need only respond with a simple “304 Not Modified” response that has no other content. This implies that any cached copy of the resource bearing the ETag provided in the If-None-Match header is still valid. If the ETag does not match, or the If-None-Match header is not included, then the server will return the full content of the resource and a “200 OK” response.

We use the ETag and If-None-Match headers to reduce bandwidth usage during re-synchronization. Each peer maintains a hash of the current cell state, being the entity tag for that cell. When a cell performs re-synchronization, it assumes that both copies are already equal to each other and provides its local ETag in the If-None-Match header. The receiving cell may then return a 304 response if the ETag matches, or the full contents of the cell if it does not.

This re-synchronization mechanism allows for processing to continue when cells have become temporarily disconnected. If a link to a particular peer fails and no updates are received by a cell, both the remainder of the network and the disconnected peer may continue to compute, as each cell is managed and updated locally. Once the cell is able to reconnect to the network, the re-synchronization mechanism will eventually reconcile any global inconsistencies that arise due to the failure of a link. If a node fails outright, we may simply treat this as a special case of link failure, in which the node performs no further computation or modification of its local cell state, and may choose to simply reinitialize with the network.

Chapter 4

Provenance-Aware Propagation

Distributed propagation allows the distribution of a number of algorithms that would not otherwise be possible within a traditional distributed system. However, in distributing computation, we lose the ability to determine how a particular result is derived. As distributed computing is inherently concurrent, there is no explicit sequence of operations that we may follow backwards in an attempt to determine the cause of an error. Furthermore, there is no way to easily identify any host that may be returning incorrect results. Distributed systems usually implement some sort of rudimentary mechanism to track the history of data for this reason.

These data histories, or provenance, provide an interesting use case for distributed propagator networks; due to the modularity of propagator networks, support for maintaining provenance may be done independently of how the networks themselves are implemented. The method of provenance handling I describe in this chapter may be applied not only to the distributed propagator networks described in Chapter 3, but, with appropriate modifications, non-distributed propagator networks as well.

We may construct these "provenance-aware propagator networks", capable of constructing and maintaining the provenance of data within them, simply by transforming the structure of the original propagator network. We need only add cells and propagators that are designed to explicitly handle the provenance of the data in the network. However, before I outline this transformation, we first must understand what is meant by the term provenance.

4.1 Provenance

For the purposes of constructing a "provenance-aware propagator network", we define provenance to be a collection of records that details the history and derivation of all data that contributes to our understanding of a specific resource. We distinguish this concept of provenance from what we call a *justification*, which is the minimal subset of this provenance from which we may derive *the current state* of a resource.

An example illustrates this distinction: Suppose we have a fact that states that the temperature, T, is greater than 40°F. We may then combine this fact with another which states that T is greater than 50°F. The combination of these two bounds now means we have a new bound on T, as it must be greater than 50°F, rather than 40°F. When we combine these two bounds, our *new* knowledge of the bounds only takes into account the second fact. We need not know the first fact ($T > 40°\text{F}$) at all in order to account for our knowledge of T. We thus say that the *justification* of the value of T is the statement "$T > 50°\text{F}$". The first fact, $T > 40°\text{F}$, which corroborates our knowledge, is a part of the *provenance* of T, along with $T > 50°\text{F}$, but it is not part of the *justification*.

This dichotomy between the *provenance* and *justification* of a resource is very similar to the distinction between "why- and where-provenance" outlined by Buneman, et al.[9] However, Buneman distinguishes between why- and where-provenance within the context of *structured data*. Why-provenance corresponds to the provenance of the data structure itself, while where-provenance corresponds to the data in the structure. In contrast, provenance and justification must be able to detail the history of conceptual resources rather than actual concrete data structures. Provenance concerns itself with the history of the resource as a whole, effectively mirroring Buneman's why-provenance; justification provides the support for the fields of the *current* state of the resource, mirroring Buneman's where-provenance.

We treat the provenance of a fact to be a collection of "records" having a partial order that either indicate where and how a piece of data was stored ("possession records") or where and how a piece of data was modified or derived ("derivation

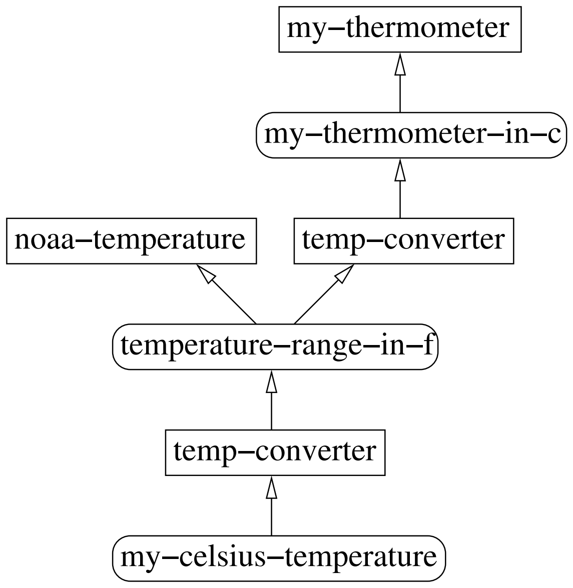

Figure 4-1: A possible provenance graph for a temperature range in Celsius.

records"). These record types are depicted in Figure 4-1 by rounded and unrounded rectangles, respectively. Each record includes an identifier of the record, as well as any additional meta-data, such as authenticity tokens, the algorithm or application used, or the current date and time, that might prove useful when auditing the provenance.

Knowledge that we hold may depend on several pieces of data from which it is derived, as in Figure 4-1. This collection of provenance may be considered a directed acyclic graph, where exactly one node, the seed, has an in-degree of 0. I refer to this collection of records as a provenance graph. The seed of the graph is the most recent provenance record; the parents of each node (the nodes to which each node points) are records of the data from which the data corresponding to that node was derived.

We may define an "additive-union" operation over two provenance graphs so that we may join them in a meaningful manner. This additive-union operation serves as the basis for our cell merge operation for provenance-aware cells. Throughout this thesis, when we refer to taking the union of, or adding two provenance graphs, we refer to this operation.

The additive-union operation may be defined mathematically. We define a provenance graph P to be the tuple $(N, A(n), s)$, where N is the set of all nodes in the graph, the relation $A(n)$ gives the set of parents of a given node n, and s is the seed of N. We may then define a new graph, P' to be the additive-union of two other graphs, $P_1 \uplus P_2$, which has the same seed, s_1, as P_1. This may be defined as follows:

$$P_1 \uplus P_2 = (N_1 \cup N_2, A'(n), s_1) \tag{4.1}$$

Here, we define $A'(n)$ to be:

$$A'(n) = \begin{cases} A_1(n) & \text{if } n \in N_1 - N_2 - \{s_1\} \\ A_2(n) & \text{if } n \in N_2 - N_1 \\ A_1(n) \cup A_2(n) & \text{if } n \in N_1 \cap N_2 \\ A_1(r_1) \cup \{s_2\} & \text{if } (n = s_1) \wedge s_1 \neq s_2 \\ A_1(r_1) & \text{if } (n = s_1) \wedge s_1 = s_2 \\ \emptyset & \text{if } n \notin N_1 \cup N_2 \end{cases} \tag{4.2}$$

Note that, in order to maintain the acyclic nature of the provenance graph, the ancestors of a node in the graph P_1, $A_1^+(n)$, must not contain any of the descendents of that node in P_2. That is, the following must hold:

$$\forall n \in (N_1 \cap N_2)\, \{n' | n \in A_1^+(n')\} \cap A_2^+(n) = \emptyset$$
$$\forall n \in (N_1 \cap N_2)\, \{n' | n \in A_2^+(n')\} \cap A_1^+(n) = \emptyset$$
$$s_1 \notin N_2 \vee s_1 = s_2$$

4.2 Data Structures on a Distributed Propagator Network

How can we actually store provenance in a propagator network? Although some naïve methods may seem obvious, they may not actually be the best choice overall. In order to better understand this, we must first examine how we may represent data structures in a distributed propagator network, as the association of data with provenance relies on having a data structure that relates the two.

Although distributed propagator networks are well-suited to handle synchronizing small amounts of data, larger amounts of structured data are more difficult to process in a distributed propagator network. Memory pointers, a traditional tool used to construct large and complex data structures, are not feasible in a distributed system. Pointers usually have meaning only when they are expressed in relation to a specific memory address space. A pointer within one program rarely has any meaning in another program, as virtual memory usually ensures that the address space of each program is independent of all others. Similarly, a pointer on one machine has no meaning on another machine, as they usually do not share any memory, and may not even share the same computer architecture.

Even though we may not simply share a memory pointer with another computer and expect the other computer to do computation on that data, the design of distributed propagator networks offers a naïve solution: as there may exist copies of the same cell that are synchronized between two hosts in a distributed propagator network, it is possible to use memory pointers that are relative to the address space of the *cell itself*, as this address space may be synchronized.

Unfortunately, this approach to sharing memory pointers is unfeasible, as all data would need to be stored in the same cell for the pointers to have any context; the address space that would be used for pointers is unique to that cell alone. If any global inconsistency arose between copies of the cell, it would be necessary to transfer the entire contents of the cell as the contents of an update during a re-synchronization. This may result in excessive bandwidth consumption if the amount of data stored in a cell is large.

The use of UUIDs to identify cells in a distributed propagator network suggests an alternative approach that may serve us better. Rather than relying on memory pointers relative to a single cell, we might be able to use UUIDs to point to other cells containing additional data. We would only need to replace traditional pointers with URL pointers to construct useful data structures in a distributed propagator network.

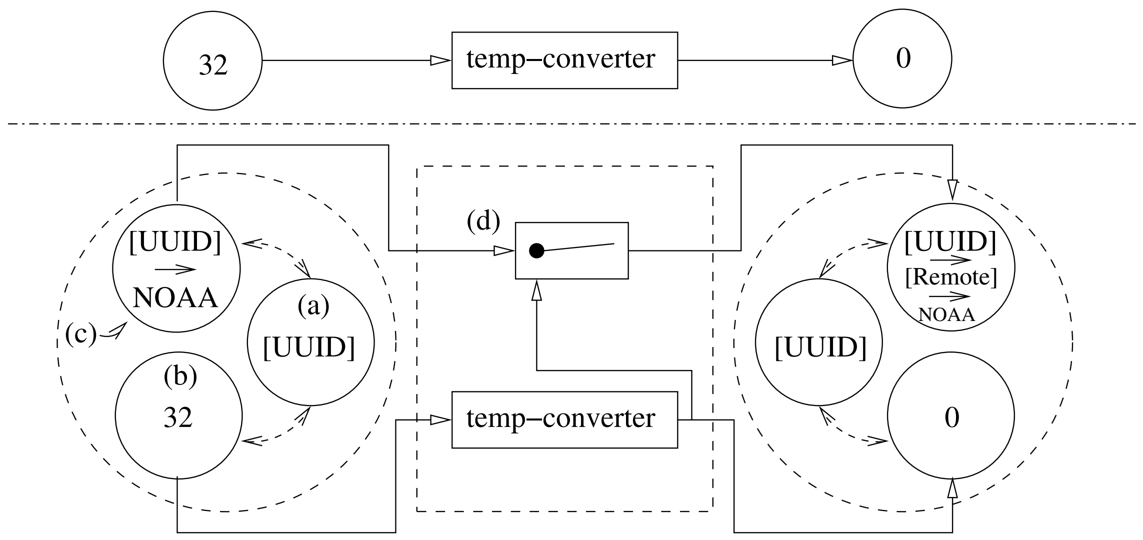

Figure 4-2: Making a simple propagator network provenance-aware.

Allocated memory normally identified by distinct pointers would be replaced with distinct cells identified by appropriate UUIDs.

We gain a number of benefits by adopting this approach to pointers in distributed data structures. As mentioned above, this approach facilitates the synchronization of large data structures by dividing them into smaller, more tractable chunks. We also may reduce the number of propagators that must be notified when a data structure changes; we only need to attach propagators to the parts of the data structure they may be interested in. For example, if a propagator is only interested in the first element of a linked list, it need not be notified to perform computation when any of the other elements of the linked list change. By separating the linked list into separate cells, the propagator will only need to be a neighbor of the cell containing the first element of that list.

4.3 Modifying Existing Propagator Networks for Provenance

As mentioned previously, the process of adding support for provenance in propagator networks, making them "provenance-aware", requires a simple transformation of existing propagator networks. In practice, we add provenance by simply introducing "another level of indirection". We combine basic propagators and cells to construct more complex compound-propagators and cell-clusters that will be our new "provenance-aware" propagators and cells. Figure 4-2 illustrates this transformation from a "simple" propagator network at top to the corresponding provenance-aware version at the bottom.

A naïve approach often used to associate provenance with data is to simply wrap the data together with the provenance in a single object. The wrapper object may cryptographically seal the data to the provenance so that the provenance may not be removed. It may also be unwrapped as needed to allow access to the underlying data so that it may be handled by operations that are unable to handle provenance. Although we could simply store provenance-wrapped data in our cells instead of the data itself, this is inefficient. Data and provenance may need to be separated to allow for independent modification of the data and provenance. Such independent modification is particularly needed within propagator networks, where additional sources of evidence may corroborate data (adding to the provenance) without actually changing the data itself.

As an example, suppose we constructed a cell that contained the maximum and minimum temperatures ever observed in Boston. It is not very likely that an update containing today's temperatures will actually change the contents of the cell, as these temperatures are unlikely to be that far from the mean. Even so, it is necessary to include the provenance of today's temperatures in the provenance of the maximum and minimum temperatures, so that an audit would be able to confirm that the maximum and minimum temperatures were selected from a dataset that included today's temperatures.[1]

The separation of provenance and data also allows us to treat their security separately as well. In some cases, such as when dealing with classified data, it may be necessary to restrict access to the data itself, while permitting auditor access to

[1]Note that we again differentiate our *provenance* of these temperatures from the *justification* of these temperatures, which would presumably come from the actual temperature readings that provided the maximum and minimum.

its provenance. Other use cases may require the opposite behavior, such as when an organization wishes to protect a whistle-blower's identity. [8] If we were to treat provenance and data as distinct entities, it is possible to set up separate access control mechanisms for the data and the provenance.

4.3.1 Provenance-Aware Cells

To maintain this separation of data and provenance, I introduce the concept of "provenance-aware cells", which are a transformation of the cells in the traditional propagator network. Instead of using a single cell to represent data, a provenance-aware cell is actually a group of three cells, the *master*, *data*, and *provenance* "sub-cells." These three sub-cells work together to associate data and provenance by maintaining UUID pointers between the sub-cells. However, separating these cells in this manner still allows them to change independently from one another. In practice, we treat a provenance-aware cell as a distributed data structure, as described in Section 4.2.

The UUID of the *master sub-cell* ((a) in Figure 4-2) provides a general identifier for the collection of the three sub-cells as a whole. This cell contains pointers to the data and provenance sub-cells. It may also contain additional meta-data that might apply to the entire provenance-aware cell. Although a complete discussion of how such meta-data would be used is outside the scope of this thesis, such meta-data might assist with use cases that require authentication of the provenance by providing a pointer to the public key of the cell's owner.

Depending on the meta-data stored, the master sub-cell usually will not need a proper merge operation, as its contents normally will not change. Since we are likely to know the meta-data of the provenance-aware cell when it is being created, we may use simple assignment as a merge operation when the contents of the cell are empty. Otherwise the merge operation may be a simple null operation.

The *data sub-cell* ((b) in Figure 4-2) contains the data of the provenance-aware cell in a simple wrapper that adds a pointer back to the master sub-cell. The merge operation on this sub-cell would be constructed from the original merge operation of the non-provenance-aware cell and a stub operation that would add the wrapper containing the pointer to the master sub-cell. This would ensure that the actual data of the sub-cell was the only data in the sub-cell that changed.

The *provenance sub-cell* ((c) in Figure 4-2) is similar to the data sub-cell: its data would consist of the provenance of the provenance-aware cell wrapped together with a pointer to the master sub-cell. The merge operation of the provenance sub-cell would be similar to that of the data sub-cell. It combines the basic merge operation used for the format of provenance that is stored in the cell with a stub that would add the pointer to the master sub-cell.

In practice, I implement a basic provenance merge operation that constructs the model of provenance described in Section 4.1 by taking the union or addition of the provenance as needed, depending on the form of the update. For example, if the cell was modified by a provenance-aware propagator, the associated update to the provenance sub-cell will contain the entire provenance of the source cell, which will be *added* to the local provenance. During re-synchronization, however, the update will contain a *copy* of the local provenance. In this case, we would take the *union* of the two provenance structures.

Unlike normal cells in a propagator network, the three sub-cells of a provenance-aware cell are not initially in a true empty state. Instead, they are initialized such that the contents of the provenance-aware cell as a whole may be considered to be empty. By this, I mean that the data and provenance cells are effectively empty. Although the pointer back to the master cell is specified, the actual data wrapped with them would be treated as an empty cell by the merge operation.

Separating provenance and data in this way gains us an additional benefit in a propagator network: we may construct and maintain provenance only at those points in the propagator network where we desire it. We would effectively treat the propagator network between such cells as a single compound-propagator which has then been augmented to be provenance-aware.

For example, a number of basic arithmetic propagators may be used to convert a temperature between the Fahrenheit and Celsius temperature scales. Although we could track the provenance of the values through each arithmetic operation, there is little knowledge gained from doing so. Useful knowledge is only obtained following complete conversion between the two scales. Instead of adding a provenance-aware propagator and cell for each operation, we need only use a simple propagator network between the two data cells and forward the provenance directly to the provenance-aware cell at the other end of the propagator network. This effectively "bypasses" the computational operations where we do not need to maintain provenance.

4.3.2 Provenance-Aware Propagators

Modification of a propagator to become provenance-aware requires the addition of a simple propagator that will read the contents of a input's provenance sub-cell and then forward its contents as an update to the output's provenance sub-cell for merging ((d) in Figure 4-2). This propagator would also modify the input provenance to add a record for the particular propagator (and possibly one for the output cell as well, if its merge operation does not add one already).

This propagator is only triggered when an update has been sent to the output's data sub-cell by the primary propagator. The added propagator effectively becomes a switch, allowing provenance updates to be forwarded only after the primary propagator has sent an update. As propagators may have multiple inputs and outputs, there must be one provenance switch-propagator for each pair of input and output cells in the original propagator. This ensures that the provenance of all input cells will only be forwarded to output cells that have actually received an update.

Chapter 5

Security and Stability

The distribution of propagation and provenance across a network introduces several issues that must be addressed to ensure that any such system is useful in constructing large systems. Some of these issues, such as garbage collecting unused copies of cells, are not immediately harmful if left unresolved; I consider such issues to be future work in this area. Other issues, such as data security and loss of stability due to race conditions, are more directly harmful to distributed propagation. If left unresolved, these issues offer significant hurdles to the consideration of data propagation as a viable computational platform. I address these issues in more detail in this chapter.

5.1 Distributing Propagators

5.1.1 Merging and Race Conditions

Although the periodic re-synchronization operation described in Section 3.8 helps to alleviate many causes of de-synchronization in propagator networks, we must also consider potential race conditions in a distributed propagator network. Race conditions naturally arise within distributed and concurrent systems as a form of non-deterministic behavior that may be caused by a reordering concurrent events. [30, 29] Such race conditions may lead to de-synchronization of copies of a cell in which data is lost in a distributed propagator system.

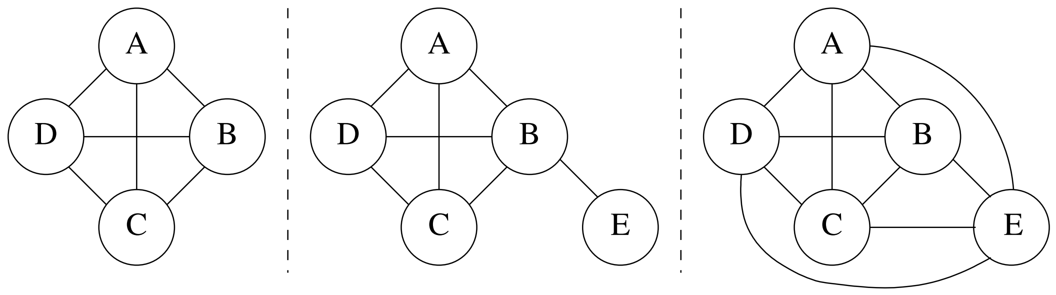

Figure 5-1: A clique of cells is temporarily broken when a new cell joins.

It is usually possible to generate a race condition in any concurrent, general-purpose computing system. With proper adherence to the protocol, however, distributed propagation reduces the chance of accidentally inducing a race condition. The idempotency and commutativity constraints placed on the cell merge operation, described in Section 3.1, limit the chances of developing many race conditions. Merges must behave identically regardless of the order or number of updates received. This implies that the contents of the cell will not usually depend on the timeliness or ordering of update messages. While it is still possible to construct applications with race conditions (e.g. a non-commutative operation retrofitted to be commutative may still result in race conditions), these problems are clearly the responsibility of the application layer, rather than caused by the propagation mechanism itself.

Other race conditions in distributed propagators are induced by the implicit concurrency of cell initialization. It is possible to construct a scenario in which update messages from an initializing cell copy (E in Figure 5-1) are dropped at existing peers between the time that the initializing cell copy learns the set of all peers and the time that the new cell has notified these peers of its existence. Similarly, the remote cell copy (B) may receive an update *after* the initializing cell copy has read the existing peer, but *before* it has learned all other remote peers in the network. In this case, the update is dropped at the new cell.

Both of these race conditions are byproducts of the fact that the network is not a complete clique during the initialization process. Although the network may begin as a clique (left side of Figure 5-1), the clique is broken when a new cell adds itself to the remote peer's "Peers" collection; this adds a single edge between that cell and the cell from which it is initializing (center of Figure 5-1). The network only returns to being a clique when the new cell has added itself to every known peer; this adds the final edge that restores the clique (right side of Figure 5-1). During the time that the network topology is not a clique, timeliness guarantees about updates do not hold.

The re-synchronization operation resolves this. Once the new cell is being initialized, the entire network may no longer be a clique, but it remains *eventually connected*. Updates made by the initializing cell will always be propagated to the remote cell, and from there, the re-synchronization mechanism will suffice to eventually diffuse those updates to the remainder of the network. Similarly, any updates that are not directly received by the initializing cell *will* be received by the remote cell, which may then propagate the message to the initializing cell through its re-synchronization operation.

5.1.2 On Connection Failure

Similar issues arise due to the need to connect to a pre-existing distributed cell, although these are not strictly race conditions. How may we handle temporary unavailability of a remote cell without blocking local computation? If the remote cell isn't available at initialization time, we still permit local computation; we may merge local data into the cell any time after it has been created, even if it hasn't been initialized from a remote cell. That said, the cell must be able to reconnect to the network when possible in order to ensure that its updates are propagated to the currently unavailable remote cell. We must be able to guarantee *eventual connectedness* of the network, or the contents of the cell will not converge.

There are several ways to approach this issue, including the naïve approach of simply retrying the connection. We may also allow for several backup URLs to be provided at initialization time. If the primary URL is unavailable, the cell would then be able to fall back on an alternate URL for another copy of the same cell. Regardless of the approach taken, recovering from an initialization failure caused by a lack of network connectivity requires the cell to continue trying to initialize. A failure to initialize may not compromise the ability of the distributed propagator system to perform local computation, but it may prevent local computation from obtaining useful results.

5.2 Securing Distributed Cells

Applications that use distributed propagation may require securing the data transferred over the distributed propagator network. Although a cell that contains facts about meteorological temperatures may seem to be a trifle, unworthy of any security whatsoever, there are no constraints on the data a cell contains; cells may contain anything from benign personal data to highly classified government information. Thus, security of data within a propagator network may be necessary to ensure wide acceptance of distributed propagation.

There are several methods that may be used to secure propagation. For example, existing methods used to secure otherwise unencrypted channels may be useful on a distributed propagator network without any security. Unfortunately, this is not necessarily a safe practice; even unsecured distributed propagation will reveal the hosts who have copies of a specific cell, identified by its UUID. It will also reveal information about the frequency and source of updates and re-synchronization attempts. By mapping this network structure to the structure of networks for which the data is *known*, it may be possible to derive knowledge about the otherwise secured network. This general approach to deriving network structure has already been used to deanonymize social networks [28], and it would be reasonable to assume that this class of attacks could be used to deduce some useful knowledge of a propagator network as well.

A more secure approach to securing propagation would be to secure the underlying protocol itself. Although the flows of data would still be traceable, meta-data about the requests, such as the identity of the cell, would be more difficult to determine without a detailed traffic analysis, if possible at all. As my proposal for distributed propagation uses HTTP as a substrate for distributed propagation, we may utilize

SSL to secure the HTTP protocol so that communications are completely opaque to eavesdroppers.

By using SSL to encrypt the underlying HTTP communications, we are not only able to defend communications against man-in-the-middle attacks, but we also provide a method of non-interactive authentication by using certificates to identify client and server. Unfortunately, naïve use of SSL is insufficient for encrypting propagation; SSL assumes that connections need only be secured between two physical hosts, rather than between two users on those hosts. This may be unacceptable on an implementation that hosts cells for a number of different individuals. These individuals may desire their own point-to-point encryption that is not possible with SSL; SSL allows a server to offer only a single server-wide certificate, rather than allowing the certificate to vary according to the URL requested.[1] The following scenario demonstrates this issue more clearly:

Assume that Alice wishes to connect to an intermittently available cell operated by Bob. Bob's cell is hosted on a DProp peer that he shares with Eve. If Eve was aware of the times when Bob's cell was unavailable, she could create a cell with the same URL as Bob's and wait for updates from Alice. Alice would be unable to distinguish whether the cell had been created by Bob or by Eve, as the only mechanism for authenticating the DProp peer is a server-wide certificate. This means that the same certificate is provided to Alice to identify the cell's creator, regardless of who actually created it.

Several naïve solutions to this issue exist. Permanent reservation of cell URLs is one such solution, as it would effectively prevent an eavesdropper from registering a cell that was previously used by another person. We could also use a secondary identifier or key that would directly identify the owner of the cell.

A better mechanism would be to encrypt the contents of the cells themselves. If Bob shared knowledge about the cell's encryption with Alice, Alice would be able to

[1] SSL's most recent incarnation, TLS, allows for certificates to be presented based on the *hostname* rather than the server's IP address. This allows the server some flexibility in selecting its certificate to present to the client, but even TLS is required to offer the server's certificate before the full HTTP URL being requested is made available to the server.

decrypt the cell and its corresponding updates. She would be safe in the knowledge that the cell had not been modified, as *only* Alice and Bob had the information necessary to encrypt the cell in the first place. This approach is problematic, however, as it requires double-encryption of the contents. This adds complexity to the code and effectively double the computational power needed to propagate data, as every new propagation would require encryption *twice*.

One other possible solution may resolve this issue; instead of double-encryption or abandoning SSL, we may be able to use the SSL protocol's existing renegotiation mechanism. SSL renegotiation allows either the server or client to force the other party to re-authenticate in the middle of SSL communications. This also allows the client or server to change its set of encryption and decryption keys. Thus, we may use this approach in distributed propagation to allow the server to change its keys to match those used to identify the requested cell while keeping all parts of the HTTP communications secure. This requires the client to know the set of server-wide keys that will be presented initially, and assumes that the server would renegotiate the set of keys upon receiving the request for the cell. If this was not done, the client could simply opt to not send any updates or requests to a cell which did not renegotiate to the expected keys.

5.3 Auditing Provenance Traces

When constructing a provenance structure like those described in Section 4.1, it may be necessary to audit the structure to ensure that data and provenance were not tampered with. Traditional methods of provenance verification may prove to be a useful basis to implementing provenance auditing mechanisms within our system.

Much work in securing provenance against tampering has assumed that provenance takes the form of a *linear chain* of modification or possession records, rather than the directed acyclic graph we assume in this thesis. [35] Linear provenance may be sufficient for some use cases; provenance-aware file systems, for example, define provenance through the operations performed on a single file, so there is no need to trace multiple inputs and outputs. [34] Propagators do not simply have singular inputs and outputs; this is why we assume that our provenance has a graph-like structure. This also means that we must be able to secure this graph-like structure against modification.

We may be able to use a method of signing chains of provenance recently proposed by Hasan, et al [21] as the basis for securing provenance graphs. Hasan proposes that each record in a provenance chain should include a cryptographic signature of that record. This signature would consist of a hash of a set of meta-data in the record combined with the signature of the previous record in the provenance chain. This hash would then be signed with the private key of the individual responsible for that particular record of the provenance. This would assign provable authorship of the record to a specific individual.

This provenance may then be verified by processing each record of the chain in order. The verification algorithm calculates the expected value of the hash from the signature of the previous record and the current meta-data and validates the calculated hash against the signature of the current record using the record owner's public key. If the signature cannot be validated, then the provenance or data has been tampered with. The provenance algorithm then processes the next record in the chain.

Although Hasan's proposal assumes that provenance is a linear chain, the algorithm may be easily modified to verify a graph-like provenance structure. Rather than signing a hash that uses the single most recent record checksum, we may sign a hash that uses *all* of the parents of a provenance record. The verification algorithm would be modified to include the signatures of all parent records when constructing the hash. Instead of traversing a linear structure, the verification would move from the ancestors of the graph towards the seed using a depth-first traversal of the graph from the seed. In this sense, Hasan's original proposal is simply a degenerate case in which the number of parents of a record is restricted to one.

To ensure integrity of provenance within distributed propagator networks, we would include this variant of Hasan's signature as part of each record stored in the provenance sub-cell. In the stub method forming the basis of the merge operation of the provenance sub-cell, we would also add a subroutine to automatically calculate the new signature for the root provenance record given the signatures of the most immediate parents of that record and include this signature as part of the record structure itself.

Chapter 6

Related Work

6.1 Distributed Cells and Replicated Databases

The work on distributed propagation described in Chapter 3 builds upon a significant amount of work in the field of replicated databases. Although cells are not as complex as databases, the replication and propagation of updates are analogous to the replication and propagation of queries in a synchronized, replicated database. Replicated database systems adhere to a different set of constraints than distributed propagator networks; these may include requiring the serialization of operations and support for non-monotonic operations, such as row deletion. Nevertheless, database update propagation mechanisms provide a useful baseline against which this system may be compared.

Grapevine [7] offers a mechanism for database replication that relies on message forwarding, much like distributed propagator networks. Although this system is similar to that of distributed propagation, Grapevine treats messaging and databases as two distinct subsystems of its architecture. Both messaging and distributed databases depend on each other to carry out their duties. Unlike distributed propagation, Grapevine assumes that the network on which it is operating is strongly-connected and that all nodes are always reachable. Distributed propagation's re-synchronization operation ensures that cells will globally converge even if a peer is temporarily disconnected from the network, which Grapevine is unable to do.

Just as distributed propagation forwards updates to copies of a cell, Grapevine propagates database changes to clones of that database. This propagation mechanism does not provide any guarantees about timely convergence. Although my mechanism for distributed propagation also provides no timeliness guarantees, it explicitly maintains a clique to attempt to provide timeliness; Grapevine makes no attempt to provide anything beyond eventual convergence. Grapevine also does not specify a resynchronization process following a node failure or failed message reception. Although the authors describe a "nightly" cloning operation designed to resolve inconsistencies, the operation is not clearly defined and does not explicitly make use of caching or other structures.

The Bayou system [39] stands in contrast to Grapevine, and adheres to a constraint of weak-connectivity in the network, much as distributed propagation does. Unlike distributed propagation, Bayou constrains itself to the use of a strict client-server model when controlling information; only one host controls a canonical version of the database. This design choice is enforced by Bayou so as to guarantee eventual serializability of updates made to a database.

These "canonical" representations of a database are not compatible with the decentralization of distributed propagator networks. In contrast to the decentralized, peer-to-peer model used for distributed propagation, Bayou implements a centralized system that makes reorderable "soft" writes to a database before "locking" them into a canonical order. In addition, Bayou, like Grapevine, offers a method to delete information from the database, and thus attempts to satisfy a constraint of non-monotonicity. Distributed propagation instead uses monotonicity as a constraint and does not need to implement this feature of Bayou or Grapevine.

The update propagation algorithm proposed by Singhal [37] also resembles the method of distributed data propagation described in Section 3.8. Singhal does not assume the existence of a weakly-connected network, however. This renders Singhal's approach unsuitable for the weakly-connected distributed computing applications we desire to create using distributed propagation. Singhal's approach also resembles Bayou in that it attempts to enforce a serialization constraint on the operations performed, which is not necessary for propagation due to propagation's explicit associativity and commutativity constraints.

6.2 Decentralized REST

As mentioned in Section 3.2, DProp utilizes REST in a decentralized manner. Rather than strictly adhering to a client-server architecture, DProp attempts to apply a peer-to-peer network overlay on top of traditional client-server communications. The work of Khare and Taylor in developing their ARRESTED architecture provides a useful comparison to our approach of constructing peer-to-peer communications with RESTful constraints. [22]

Although many of the principles of ARRESTED are similar to the principles behind my proposal for distributed propagation, ARRESTED also suggests several features that have been designed to adhere to constraints that distributed propagation does not have. ARRESTED explicitly includes estimation, locking, and routing in their decentralized REST architecture. These operations are unnecessary for distributed propagation due to its monotonicity constraint.

Furthermore, there is no need to ensure reception of a message in a distributed propagator network, as the cell re-synchronization mechanism, combined with the existing constraints of associativity and commutativity ensure that a repeated attempt to merge an update will not change the final contents of the cell. Similarly, propagator networks do not have a particular need to estimate the current contents of the cell based on previous updates. Although such a mechanism could be constructed with propagators, there is no implicit need to estimate, as timeliness is not a virtue in distributed propagator networks. We need only guarantee eventuality.

6.3 Distributed Provenance

There are a number of different approaches to the integration of provenance in a system. Many of these approaches are domain-specific, such as for file systems [34] or web services [38]. Our approach is a more holistic one, demonstrating the viability of provenance collection and maintenance in a general-purpose computing architecture and the easy modification of existing software to add support for provenance. There are many similarities between my proposed system and more general approaches to the inclusion of provenance in service-oriented architectures, such as those proposed by Moreau, et al. [26]

In Moreau, et al's work, an architecture for provenance handling is proposed with the aim of achieving many of the same goals of provenance-aware propagators. This includes the automatic documentation of processes with what Moreau calls *p-assertions*, which are functionally similar to the possession and modification records we use in our provenance model in Chapter 4. Moreau addresses the problem of querying constructed provenance, instead of focusing on the construction and maintenance of provenance, with which we are concerned. Furthermore, Moreau assumes that general-purpose provenance, such as the *purpose* of an action, may be collected within their system. While this is certainly possible within provenance-aware propagator networks, we make no explicit claims about how such potentially subjective provenance traces are handled.

The Matrioshka system proposed by da Cruz, et al. [12] also attempts to provide provenance in distributed workflows much as provenance-aware propagation does. Unlike the provenance system described here, Matrioshka relies on a centralized provenance store rather than distributing the provenance with the data to which it applies. The system also assumes that logging functionality already exists in a distributed application. This makes Matrioshka more brittle than provenance-aware propagation; provenance-aware propagation does not require any intermediate steps to generate logs or store provenance, as provenance is generated and stored along with the data it pertains to. Provenance-aware propagation also allows for existing applications to be retrofitted as provenance-aware applications, which da Cruz does not demonstrate with Matrioshka.

The work of Altintas, et al [1] in extending the Kepler Scientific Workflow System is perhaps the most similar effort to integrate provenance in existing distributed systems. Altintas makes use of the underlying flexibility of Kepler to add a provenance framework, much as I do with distributed propagation. This framework simply acts as part of the Kepler system, constructing and tracing the provenance of events that operate within it, and it may be easily added to existing programs that use Kepler.

This system does not port itself well to the propagation paradigm, due to propagation's inherent decentralization. Kepler appears to rely instead on a client-server model of computation. This allows the server to additionally track provenance. Although we could create a provenance host that would trace the activity of all cells in a manner similar to that of Kepler, this is not scalable. A large enough propagator network would effectively drown any host listening to all updates while trying to construct the provenance of data within the network. This defeats the purpose of adopting distributed computation.

In many respects, provenance-aware propagator networks complement Munroe, et al's PrIMe system [27]. Rather than providing an explicit mechanism for the inclusion of provenance in existing systems, PrIMe is a *software design methodology* that may be used to construct provenance-aware applications by taking provenance into account during the design process. As a software design methodology, PrIMe differs greatly from my proposal, being a more concrete implementation of a provenance-aware processing system. PrIMe does not specify a particular *implementation* of provenance, but merely helps developers consider implementing provenance in systems that require it. Provenance-aware propagation may offer an opportunity to facilitate design processes that use PrIMe by reducing the work necessary to retrofit existing systems with provenance capabilities.

As mentioned in Section 5.3, it is possible to extend provenance in a provenance-aware propagator network to include provenance integrity mechanisms by building on the approach set forth by Hasan, et al. [21] Hasan applies a provenance integrity mechanism to construct a provenance-aware file system. In contrast, provenance-aware propagation is a more general mechanism for provenance construction that can be used to construct distributed file systems, as well as many other applications. I thus expand on Hasan's approach to general purpose computation.

Chapter 7

Summary

7.1 Contributions

I have discussed and implemented a distributed system based on the computational paradigm of data propagation in an attempt to provide distributed solutions to problems with high dimensionality. This architecture, which I call "distributed propagation", achieves useful computation within a weakly-connected network, making it comparable to existing distributed computing approaches that also assume such. I designed this system with the principles of Representational State Transfer, or REST, in mind, resulting in a system that is both robust to failure and simple enough to be easily implemented on multiple platforms.

The development of distributed propagation has also led me to outline and implement a provenance mechanism for distributed propagator systems. This mechanism takes advantage of the explicit modularity of propagator networks and provides for the simple modification of existing networks to add support for provenance. I have implemented a library to support the creation of distributed propagator networks which demonstrates this by implementing fundamentally identical APIs to construct both traditional and provenance-aware distributed propagator networks.

I have also outlined several constraints on the merge operations used in data propagation; these constraints reduce the complexity of distributed propagator networks and allow for meaningful distributed computation. The four constraints and their

Constraint	Benefit
Idempotence	(with associativity) no locking during cell initialization
	no filtering for duplicate updates during re-synchronization
Associativity	(with idempotence) no locking during cell initialization
	don't need to save all updates for re-synchronization
Commutativity	removes need for global synchronization (timestamps)
	removes timeliness constraint on communications
Monotonicity	removes complexity caused by deletion operations
	allows computation of results with partial knowledge

Table 7.1: The four constraints on the merge operation and their benefits.

corresponding benefits are outlined in Table 7.1. Finally, I have also developed a loose guideline for how data structures may be represented in propagator networks, either by including data structures in the cells themselves, or by treating cell URLs as de-facto memory pointers.

7.2 Challenges and Future Work

7.2.1 Propagator Networks & Garbage Collection

In this thesis, I have assumed that the collection of all cells in a propagator network is monotonically increasing. There is no explicit "deletion" or "unregistration" mechanism for a distributed cell. Once a cell has been created, it may not be deleted, nor may a host disclaim any continuing interest in it. There is no mechanism for garbage collection in a distributed propagator network; this results in an intractably large propagator network as time progresses.

In practical systems that use distributed propagator networks, it may be necessary to support non-monotonic actions, such as the deletion or unregistration of a cell from a given network. In doing so, we may allow for more efficient allocation of the resources on a host. Rather than spending effort to re-synchronize or maintain a copy of a cell that will never be re-used, a host may allocate resources to more useful computation.

There is a similar need for a mechanism to allow propagator networks to eventually prune unreachable peers. This would allow networks to eventually stop making efforts to synchronize a node that no longer exists. A implementation of garbage collection within propagator networks in general will be needed before distributed garbage collection may be considered.

7.2.2 Data Structures and Propagators

It is not entirely clear that the URL-based distributed data structure design that I propose in Section 4.2 is useful for more general computation in propagator networks. It may be difficult to merge two data structures for which only pointers are known. If two pointers are assigned at the same time to different (shared) cells, there is no way to determine how to merge these otherwise opaque pointers. A sensible merge algorithm may need to look at the contents of each pointer to unify cells that may have been assigned different UUIDs.

The provenance-aware cells proposed in Section 4.3.1 are not affected by the issue raised above. When a provenance-aware cell is created, all pointers are allocated to appropriate sub-cells automatically. This means that any merge operation on the sub-cells of a provenance-aware cell should never have to merge two distinct pointer values that are assigned to the same field; the merge operation may safely ignore the contents of the pointer field other than ensuring that it remains assigned.

7.2.3 Redundancy of Distributed Operations

One advantage of distributing propagation is that we may *duplicate* a host as many times as we like and have all duplicates perform the same computation in parallel. This would allow for error-correction and fault-tolerance, as a failure to produce correct results would be ameliorated by the ability of other hosts to generate the correct values.

At this point, all such duplication would have to be designed into the application. This may be problematic, as the number of duplicate clients may not be known at run time. Resolving this issue and allowing dynamic allocation of redundant computation will require the resolution of the compound propagator issue described below.

7.2.4 Scalability and Practicality Issues

Although I have tested distributed propagation successfully across a network, these tests have been limited in scope to only two or three peers. I have not yet tested larger distributed propagator networks, and it is unknown whether propagator networks will scale practically; the amount of re-synchronization communications is likely to result in flooding the network as the propagator network grows large. As each cell re-synchronizes with every other cell that it knows about, and my distributed propagation model assumes the existence of a clique, the number of re-synchronization messages per unit time will grow according to the square of the number of peers.

Cell Discovery

Although this thesis has considered discovery of an initial cell copy to be orthogonal to the problem of distributing propagator networks, a scalable distributed propagator network will need to be able to determine what cells exist and where copies are located. A number of mechanisms may be helpful in moving beyond the assumption of a priori knowledge passed to the API, which I have assumed in this thesis. Other mechanisms may include name servers like DNS [24, 25] or service-discovery protocols such as DNS-SD [11] or SSDP (part of the universal plug-and-play architecture (UPnP) proposed by the UPnP Forum [40, 41]).

Compound Propagators in Distributed Propagator Networks

As distributed cells must be assigned a UUID identity *before* they may be connected to propagators or remote cell copies, there is no mechanism to create a cell on a remote host and spawn a given propagator attached to it. This makes it difficult to dynamically negotiate or delegate tasks in a distributed application. This also makes it impossible to create distributed compound propagators, one of the more compelling elements of propagator networks. Although we may instantiate *local* propagators when a cell receives data, we may not dynamically assign tasks to remote hosts by providing a propagator and cell.

It may be possible to create such dynamic propagator networks by treating propagators as first-class objects that may be stored in a cell. We might then be able to distribute propagator and cell descriptions to remote nodes so that they may be instantiated for further processing. Allowing arbitrary code execution does raise security concerns, so any mechanism for compound propagation within distributed propagator networks will need to be carefully crafted to eliminate issues that arise due to allowing the dynamic construction of distributed propagator networks.

7.2.5 Security

I have outlined several approaches to security in Sections 5.2 and 5.3. However, my implementation, DProp, and its accompanying library, PyDProp, currently do not implement them. As such, the security implementations proposed in this paper should be studied to ensure that data remains secure and that effective access control is possible in this system. Improved encryption may be had by implementing propagation with a new protocol that allows for cell-based security mechanisms.

SSL may also suffer scalability issues. As my proposed SSL implementation of security depends on knowledge of certificates, the number of certificates that must be "known" by a user of distributed propagation may be high in a large distributed system. Unified authentication systems such as Kerberos may be more appropriate within larger networks than SSL, allowing greater scalability and larger propagator networks.

7.2.6 Representational State Refinement

Strictly speaking, the combination of propagation and REST I have set out in Chapter 3 does not strictly adhere to several of the underlying principles of REST. Although it nominally adheres to the five primary constraints of REST, distributed propagation does not follow some of the underlying tenets of hypertextuality and resource management that a true RESTful architecture provides. In particular, distributed propagation does not generally concern itself with the creation or deletion of resources; most, if not all, actions on cells are done solely through the GET and POST operations.

Nevertheless, it seems inappropriate to simply abandon a RESTful model for distributed computation. Distributed propagation does make effective use of resource retrieval and update, and actually places a more meaningful constraint on the meaning of "update" implied by the POST operation. Updates within a distributed propagator network may only be used to "refine" a resource, rather than allowing any arbitrary modification.

As completely disposing of REST would be inappropriate, I propose that the work presented here is not so much that of "Representational State *Transfer*" but of "Representational State *Refinement*", or RESTAR. This architectural style essentially synthesizes the principles of propagation and resource management into a single model. Rather than simply providing for the underlying CRUD operations of creation, reading, updating, and deletion, RESTAR supposes the replacement of the nondescript update operation with a "refinement" operation which selectively updates the resource by refining partial knowledge of the resource.

7.2.7 Which HTTP Method? POST vs. PUT vs. PATCH

One critical aspect of constraining ourselves to a new REST-based model is the correct choice of HTTP method for conveying updates to remote cell copies. Although I have modeled propagation of updates through the use of the POST method, we must be careful that this is the correct HTTP method to use. The effective semantics of a POST are vague; POST is stated to be a non-idempotent method that is to be used to effectively create a new "subordinate" of the resource to which the POST request is sent. [17] However, this is commonly interpreted in practice to allow for the modification of the resource itself, and this is how I use it in this paper.

Unlike POST, PUT is specified to be idempotent. Although this might seem to be preferable, PUT is much more strongly associated with the semantics of "creating" or "replacing" an object at the URL provided in the request. As distributed propagation never actually creates cells, and updates cannot simply "replace" the old content of the cell without abandoning the monotonicity constraint on the merge operation, PUT may not be used for distributed propagation.

The PATCH method of HTTP [16] carries more appropriate semantics, in that it allows a set of changes to be applied to the resource to which a PATCH request is sent. However, PATCH is still specified to be non-idempotent. Distributed propagation explicitly requires idempotency, so this would require an implicit additional restriction on the PATCH operation.

Unfortunately, while the PATCH method would thus be the most appropriate method to implement distributed propagation, some HTTP implementations may not be able to support this relatively new method. The Twisted library and Python's `httplib` module, which I have used to construct my implementation, DProp (See Appendix A), do allow the PATCH method, and I plan to revise DProp to support the PATCH method in a future release.

7.3 Conclusion

Distributed propagation provides a useful alternate mechanism for distributed computation that does not rely on simply distributing the data across the system for identical processing. This will provide for a larger number of computations that may be distributed, and perhaps allow for a greater number of applications for distributed computing. As the power of individual computers begins to reach a plateau without making use of concurrent processing, it will become ever more important to construct systems to make use of a larger number of hosts and processors and to move away from sequential processing.

By providing a mechanism for the distributed *refinement* of resources, distributed propagation and related applications may provide a means to develop more push-based, data-driven algorithms over a network and on the world wide web. As interest in other push-driven web application technologies like Comet and Web Sockets grows, distributed propagation may provide a meaningful development model for those applications that do not wish to abandon the resource-based architecture style of REST.

Appendix A

DProp: Distributed Propagation in Practice

In developing the system for distributed propagation described in Chapter 3, I have constructed a working implementation of the protocol, called DProp. DProp makes use of Twisted, a Python-based, event-driven networking framework to provide the HTTP server functionality needed for distributed propagation. The Twisted library may be used to easily construct an HTTP-based server application with optional SSL encryption. Together with Python's built-in httplib module, used to implement HTTP requests, I am able to construct a useful mechanism for distributed propagation

I have also constructed a library to accompany DProp, named PyDProp. This library simplifies the construction of cells shared via DProp by offering a simple API for cell creation. This library may also be used to construct provenance-aware cells, which share the same basic API used for non-provenance-aware propagation. By sharing the API for creating both provenance-aware and non-provenance-aware cells, I am able to demonstrate the ease with which provenance may be added to existing applications that use distributed propagation.

A.1 The Architecture of DProp

Although I have outlined a basic protocol for distributed propagation in Chapter 3, that outline simply lays out the architectural requirements of the protocol itself; there are no provisions made for the architectural design of applications that implement the protocol correctly. As long as a program adheres to the protocol, its implementation may be integrated into the application, or made to be reused by many programs. DProp has a more flexible architecture that allows it to be reused by many existing programs.

As described above, DProp uses the Twisted networking stack to implement the actual HTTP communications. Although it would be possible to construct a separate propagation stack for each program that utilized propagation, I have instead chosen to implement DProp using a client-server architecture. Local client applications may create cells on the DProp server that may be made available to the network at large. This design choice separates propagation functionality from the applications that use it; they do not need to actually implement distributed propagation and may rely on the DProp daemon to handle propagation for them. Client applications would then only be contacted for cell-specific operations, such as cell merges and updates.

A client application need only be able to communicate with the DProp daemon through the DBus messaging library, an open-source inter-process messaging library. Programs may use a common API enforced by DBus to create cells. By relying on a single daemon for all propagation, we reduce the number of distinct propagator implementations, and reduce the number of bugs in propagator networks that may be caused by an incorrect implementation of the propagation protocol.

More importantly, this client-server architecture *also* allows propagators on the same host to run as separate processes, provided that they can both communicate with the same DBus messaging bus. The DProp daemon can manage the sharing of a single cell across multiple processes simply by using the same APIs in different processes. By using these APIs to merge and update the same cell, two processes may effectively share a cell as they might share memory.

`escapePath(path)`	Escape part of a path to a cell for use with DBus.
`useSSL()`	Returns True if the daemon is hosting using SSL, False otherwise.
`port()`	Returns the port number the DProp daemon is hosting on.
`registerCell(uuid)`	Creates a new cell identified by `uuid()` if it doesn't exist and returns the UUID.
`registerRemoteCell(url)`	Creates a new uninitialized local copy of the cell at `url` if it doesn't exist and returns its UUID.
`cellExists(uuid)`	Returns True if a cell with the given UUID already exists, False otherwise.

Table A.1: DBus remote methods available on a DPropMan object.

`escapePath(path)`	Escape part of a path to a cell for use with DBus.
`changeCell(data)`	Change the contents of the cell to `data` (which should be encoded as a JSON string) and notify local propagators.
`updateCell(data)`	Send a JSON-encoded update message, `data`, to the merge operation associated with the cell and forward the message to all remote peers.
`data()`	Return the data of the cell encoded as a JSON string. data() effectively returns the argument of the previous call to changeCell(data).
`url()`	Returns the URL of the cell.
`connectToRemote(url)`	Actually connect to the cell at the remote peer, specified by `url`, and set it to synchronize with the cell.

Table A.2: DBus remote methods available on a Cell object.

`UpdateSignal(message, peer)`	Sent by the cell when a (local or remote) update has been received by the cell from `peer`.
`NotifyPropagatorsSignal()`	Sent by the cell to wake up any connected propagators.

Table A.3: DBus signals sent by a Cell object.

DBus is an *object-oriented* messaging system. This means that messages in DBus are associated with particular data objects. These data objects are usually created by one application, but may be sent messages from many other client applications. Shared data objects in DBus reveal remote methods defined specifically for the type of object being shared.

The applications that have created data objects on DBus may also communicate back to client applications through a number of DBus "signals". Signals are special methods associated with a data object that result in firing call-back methods that may have been registered to receive the signal in various client applications.

DProp shares two types of objects on DBus: a single `DPropMan` object, and a collection of `Cell` objects. The `DPropMan` object represents the state of the DProp daemon itself. It has methods that test whether a cell exists, as well as some that create new `Cell` objects. These methods are outlined in Table A.1.

A `Cell` object represents a single cell copy in a distributed propagator network, shared with the world at large. Many of the methods made available on a `Cell` object relate to updating and merging the contents of the cell. These methods are listed in Table A.2. Cells also expose several DBus signals to client applications. The signals associated with each `Cell` are used to notify client applications of updates received by the cell and completed merges. A complete list of signals is given in Table A.3.

A simple example of cell creation and updating using DProp is depicted in Figure A-1. When an application first requests that a cell be made, the server will permanently instantiate a cell with the provided UUID if one does not already exist. This cell may then be used as the recipient of further remote method calls over DBus. Once the cell has been created, a client may register appropriate local functions to handle the signals generated by a particular cell. These include `UpdateSignal`s generated whenever a local or remote update message is received.

Locally generated updates may be sent to the cell by using the `updateCell` method. When an update is sent locally, the server will distribute the update to remote cells, as outlined in Section 3.7, and then signal any locally registered merge operations by generating an `UpdateSignal`.

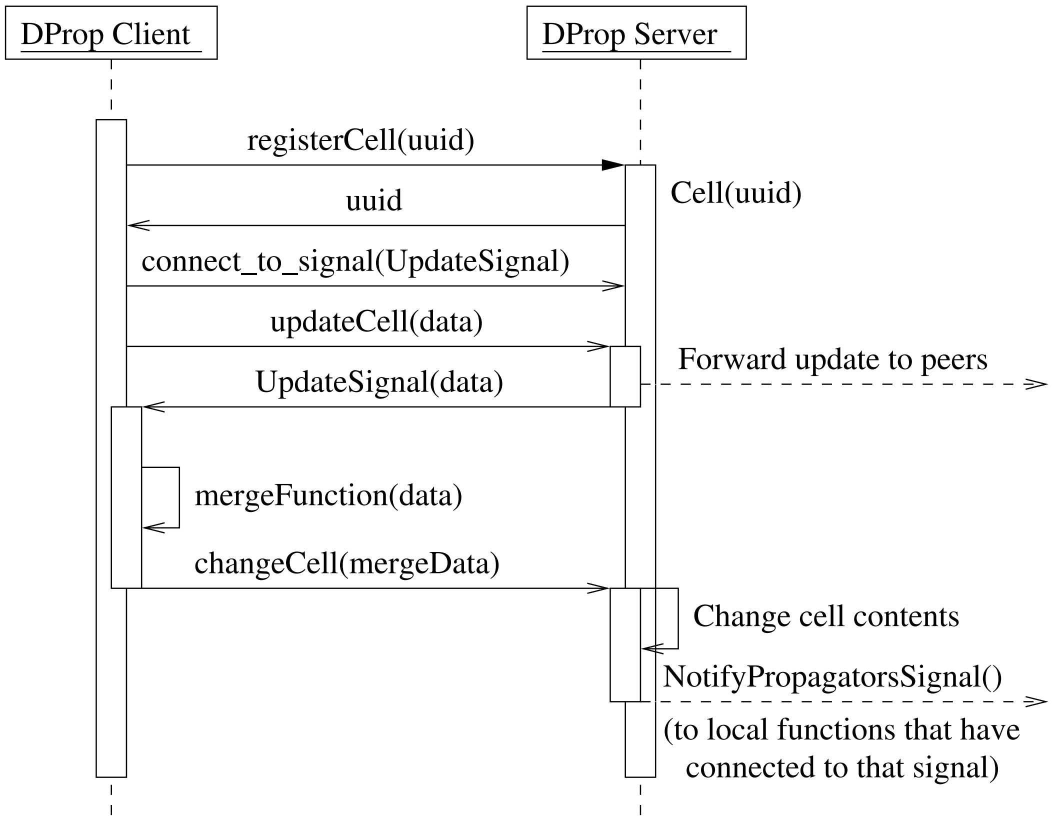

Figure A-1: Example communications between a DProp client and server via DBus.

When an `UpdateSignal` has been generated by the DProp daemon, all local merge functions that have been registered to the `Cell`'s `UpdateSignal` will be invoked. These merge functions may then perform computation based on the current cell state, which may be obtained from the cell using the `data` method on the `Cell` object. They may then properly set the contents of the cell to the newly merged state by using the `changeCell` method.

This second method actually changes the contents of the cell that are returned by the `data` method via DBus or a remote GET request over HTTP. The `changeCell` method also wakes any local propagators associated with the cell by generating the DBus signal `NotifyPropagatorsSignal`. Local propagators interested in the cell should be registered to receive this signal and perform any computation; this may include further calls to `updateCell` to send updates to additional cells.

A.2 The DProp Daemon

The DProp daemon handles the synchronization of local cell copies, as well as handling requests from other copies. It implements both the HTTP server and client mechanisms necessary to perform the distributed propagation protocol outlined in Chapter 3.

When the DProp daemon starts, it will instantiate an HTTP server on port 37767 and connect to DBus. The DProp daemon creates a `DPropMan` object at runtime and associates it with the DBus path `/DPropMan`. The remote methods of this object may then be called by any client; clients need only reference the name of the bus that the object is hosted on, 'edu.mit.csail.dig.DPropMan', and the path of the object, `/DPropMan`. Any of the methods described in Table A.1 may be called on the DPropMan object.

When a cell copy is created using the `registerCell` or `registerRemoteCell` method, the cell object gets created at the path `/Cells/{UUID}`, where it may be called upon to update or register signals. This cell is also made available on the HTTP server by assigning it a URL that is named with reference to the host's name:

‘http[s]://{host}:{port}/Cells/{UUID}’. The collection of the peers of the cell is also created below this URL.

Regardless whether a cell is created with `registerCell` or `registerRemoteCell`, the cell that is created is initially empty. To finish the initial synchronization process with a cell created by `registerRemoteCell`, the `connectToRemote` method must be called on the cell. We separate cell creation from initialization of the cell as no merge function is registered with the cell immediately following the `registerRemoteCell` call. A client must register a call-back function with the `UpdateSignal` signal of the cell before we may actually perform any merges or synchronizations; if this is not done, the contents of the remote cell will not be properly merged when they are fetched. Likewise, call-backs to `NotifyPropagatorsSignal` must be added to ensure that propagators fire.

This separation of cell creation and initialization also helps users initialize against a remote cell without blocking execution; if this was not done, a thread might hang while it waits to connect to a non-existent remote cell. Finally, it allows the DProp client application to finely control connection to a remote cell. This allows a client application to provide fall-back mechanisms should a cell fail to connect. For example, it may be able to try alternate remote URLs or initialize the contents of the cell based on some local saved state.

A.2.1 Serving Cells

The DProp daemon uses the Twisted framework to serve HTTP resources. The built-in `twisted.web` module allows for the creation of resources simply by sub-classing the `twisted.web.Resource` class. We then provide a top-level resource when the HTTP server is instantiated. We use a `DPropManTopLevel` object to represent the top-level resource containing the `/Cells` collection. Similarly, a `DPropManCells` object represents the collection of cells, a `DPropManCell` object represents a single cell, and a `DPropManCellPeers` object represents the collection of a cell’s peers.

Each HTTP resource may respond to any number of HTTP methods simply by defining instance functions on their Python objects with names having the form `render_METHOD(self, request)`. For example, an HTTP GET method on a cell resource will result in calling the `render_GET(self, request)` method on the corresponding `DPropManCell` object. The `render_GET` method has been implemented to check whether an `If-None-Match` HTTP header has been provided. If it has, and the value of the header matches the `ETag` of the cell's contents, the resource will return an empty response along with the "304 Not Modified" response code to imply that any caches with that ETag are still valid. Otherwise, the cell will return the current contents of the cell along with a "200 OK" response code and a new `ETag`.

When a cell receives a POST request and the `render_POST` method is called, the Python object will confirm that the HTTP `Referer` header field has been specified and that the content of this field is the URL of a cell that is already present in the cell's peers collection. If the `Referer` is unrecognized, the cell will return a "403 Forbidden" response code to signify that the POST was not allowed. Otherwise, the cell will forward the update to any interested cell merge operations by calling the `UpdateSignal` associated with the cell. The cell returns a "203 Accepted" response to denote that the submitted POST was accepted for merging into the cell, even though it may not actually change its contents.

We may similarly accept a POST on the Peers collection. The `render_POST` method on the corresponding `DPropManCellPeers` object does not check the `Referer` however. It simply adds the URL of the peer provided in the contents of the POST to the peers collection without doing any validation or verification of the new peer named in the `Referer`.

A.2.2 Fetching Cells

The DProp daemon is also in charge of handling the re-synchronization protocol described in Section 3.8. This is performed automatically by running a method to perform this re-synchronization, `startSyncThunk`, repeatedly on a timer. When this method fires, the daemon will attempt to simultaneously re-synchronize the local cell with each cell copy in the Peers collection. This is done in a concurrent, multi-threaded manner.

This multi-threaded behavior is necessary to prevent deadlocks that might be caused when two peers attempt to re-synchronize with each other at the same time. If the HTTP server handles requests on the same thread as the HTTP client makes them, responses to any requests made to the server will not be generated while DProp waits on a response from the remote server.

Once an HTTP request has been made for a cell, the response may be treated according to the rules outlined in Section 3.8. The cell will fire a local update if there was a loss of synchronization or do nothing if no re-synchronization is needed.

Updates to a `Cell`, regardless of whether they were received from a remote cell copy or a local call to `updateCell`, ultimately call the `Cell`'s `doUpdate` method. This method, not publicly visible via DBus, is called by both the `render_POST` method of the HTTP server and `updateCell` method of the DBus `Cell` to send an `UpdateSignal` to any local merge functions that have been registered to receive it. It also calls the `updatePeers` method if the update was sent locally. This method is used to forward the update to remote peers.

As mentioned previously, the `connectToRemote` method may be called to properly connect the cell to a network of synchronized cells. Like `startSyncThunk`, this method spawns a thread to initialize the cell from the remote copy. It will fetch the contents of the remote cell with an HTTP GET request and call `doUpdate` to pass the contents of that cell to the local update function. The method then proceeds to merge the peers known to the remote cell into the local Peers collection and informs these peers of the existence of the new cell copy.

A.3 PyDProp

The DProp daemon is accompanied by a separate Python library, called PyDProp, which is designed to interface with the daemon through DBus. This Python module may be used with any Python code to create and synchronize remote cells without writing any low-level DBus code. PyDProp also includes classes that implement the provenance-aware cell model described in Chapter 4.

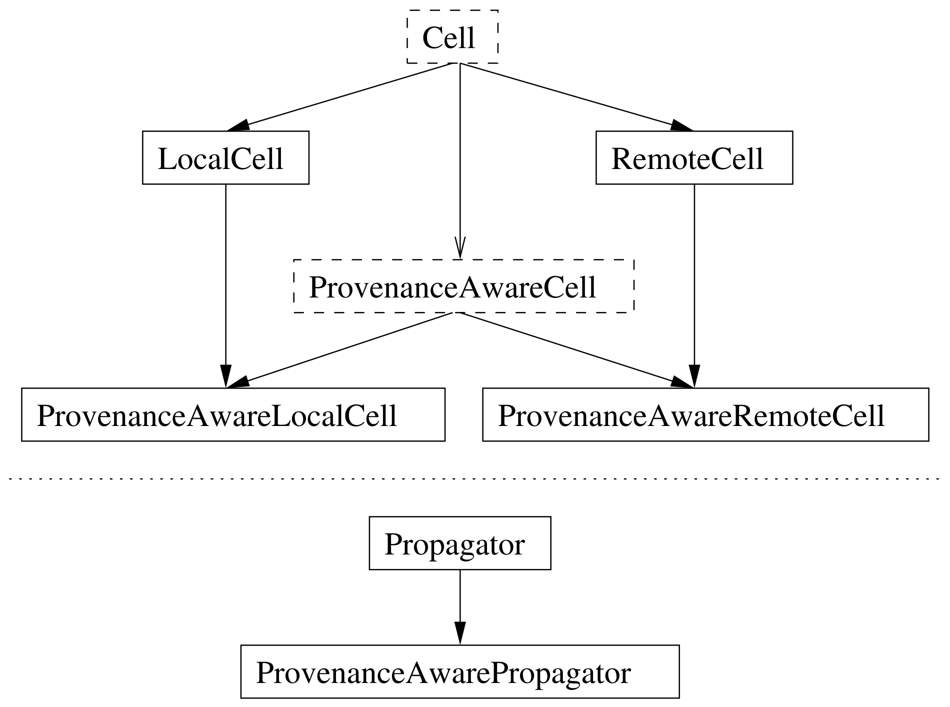

Figure A-2: Class hierarchy of PyDProp.

`addNeighbor(propagator)`	Registers the `function` method of the provided propagator object with the `NotifyPropagatorsSignal` signal of the cell so that it is called when the cell updates.
`update(data)`	Updates the cell with the JSON-encoded update provided in `data`. Wraps the `updateCell` method.
`set(data)`	Sets the cell to the JSON-encoded update provided in `data`. Wraps the `changeCell` method.
`data()`	Returns the JSON-encoded data stored in the cell.
`url()`	Returns the URL of the cell.

Table A.4: Methods available on a Cell object in PyDProp.

PyDProp provides four concrete classes, including `LocalCell`, `RemoteCell`, and the provenance-aware versions of these classes, `ProvenanceAwareLocalCell`, and `ProvenanceAwareRemoteCell`. These four classes are sub-classed from one of two abstract classes `Cell` and `ProvenanceAwareCell`, as illustrated by the class-hierarchy in Figure A-2. The `LocalCell` classes are used to represent a locally-created cell which will not be initialized from a remote cell copy. The `RemoteCell` classes, in contrast, may be initialized from a copy of a cell at another URL. All of these classes inherit methods from the abstract `Cell` class and share the five basic methods of that class, listed in Table A.4. These five methods may be used to perform basic cell-related tasks, such as retrieving the cell's contents and updating a cell, without needing to directly invoke methods exposed through DBus.

A.3.1 The LocalCell and RemoteCell Classes

The `LocalCell` and `RemoteCell` classes are subclasses of the `Cell` class that differ only in how they are initialized. Both are initialized with two arguments. The first argument is the UUID, if the cell is a `LocalCell`, or the URL, if a `RemoteCell`. The second argument of both constructors is a merge function that takes three arguments: the Cell object itself, the update data encoded in JSON, and the peer that sent the update.

The `LocalCell` object simply registers the cell with the DProp daemon using the `registerCell` method of the `DPropMan` object on DBus. It then registers the provided merge function with the `UpdateSignal` of the `Cell` created by `registerCell`. Instead of calling `registerCell`, `RemoteCell` uses the `registerRemoteCell` method. It also calls the `connectToRemote` method after the merge function has been registered with the `UpdateSignal`.

A.3.2 Provenance-Aware Cells

`ProvenanceAwareLocalCell` and `ProvenanceAwareRemoteCell`, the other two subclasses of `Cell`, are based on the abstract `ProvenanceAwareCell`. Like `LocalCell`

`mainCellMerge`	Performs the default merge (replacement) operation for the main cell.
`provCellMerge`	A wrapper for the provenance merge function given as the first argument `provMergeFunction` and performs the merge on the provenance cell.
`dataCellMerge`	Like the `provCellMerge` function, this function wraps the first argument `mergeFunction` and handles merging of data.
`defaultProvCellMerge`	The default provenance merge function.
`updateProvenance`	Like `update` on the original Cell class, but specifically updates the provenance cell.
`setProvenance`	Like `set` on the original Cell class, but specifically sets the provenance cell.
`provenanceData`	Like `data` on the original Cell class, but specifically fetches the data of the provenance cell.

Table A.5: Methods available on a ProvenanceAwareCell object in PyDProp.

and `RemoteCell`, these provenance-aware subclasses differ only in the arguments they take, although both now take an optional provenance merge function as their last argument. Furthermore, `ProvenanceAwareLocalCell` now requires three UUIDs, one for each sub-cell, rather than one UUID for the entire cell.

The parent class of these classes, `ProvenanceAwareCell`, adds several additional functions to the basic functions provided by the `Cell` class. These additional functions are listed in Table A.5. Several wrapper methods and default merge functions are provided to help with merging the individual sub-cells of provenance-aware cells. `ProvenanceAwareCell` also overrides some of the methods of the `Cell` type to account for the differences between the structure of a basic `Cell` and a `ProvenanceAwareCell`. Despite these differences, sub-classing `Cell` still allows us to easily reuse existing code written to use non-provenance-aware cells, as the basic five functions of `Cell` remain functionally the same.

A.3.3 Propagators

Propagators in PyDProp are first-level objects. These objects are all members of a class that wraps propagator functions, named `Propagator`. The constructor of a

`Propagator` object takes three arguments: an identifier for the propagator, a list of neighboring cells that will wake the propagator when their content changes, and the propagator function itself. The constructor automatically registers the propagator function with the `NotifyPropagatorsSignal` of each neighboring cell so that the propagator will wake up when one of the neighboring cells changes. The `Propagator` object also wraps the propagator function through its instance method `function()`.

Provenance-aware propagators are constructed similarly to non-provenance aware propagators. Rather than using the `Propagator` class, provenance-aware propagators use the class `ProvenanceAwarePropagator`, which takes an additional argument following the list of neighbors. This argument contains the list of output cells of the propagator. This ensures that the provenance sub-cells of each provenance-aware cell that has had its *data sub-cell* updated will be updated as well. It also takes an optional provenance-propagation function to allow for custom structures of provenance rather than the one described in Section 4.1. Otherwise the propagator will provide one of its own to create the provenance structure described in that section.

A.4 DProp and JSON

As may be noted in Tables A.2 and A.4, the arguments to the methods of a `Cell` object in both DBus or PyDProp are objects encoded as strings using JSON, a data interchange format based on JavaScript syntax. This choice is primarily driven by the static typing of functions exposed through DBus. As it would be impossible to store an arbitrary data type in a `Cell` and retrieve it using DBus without knowing which of the static types is stored in the cell, I chose to use a neutral data interchange format, JSON, to represent the data stored in a cell.

To handle conversion between arbitrary data and JSON, it is necessary to map arbitrary objects to a JavaScript "object" bearing key-value pairs that map a string to a string, number, array, or other object. Encoding and decoding the data of a cell in JSON may be handled by existing JSON code libraries. However, several special objects used in distributed propagation and provenance require special handling. This

Type	Description
`dpropMainCell`	Data stored in the main sub-cell of a provenance-aware cell, including pointers to the provenance and data sub-cells.
`dpropDataCell`	Data stored in the data sub-cell of a provenance-aware cell, including a pointer to the main sub-cell.
`dpropProvCell`	Data stored in the provenance sub-cell of a provenance-aware cell, including a pointer to the main sub-cell.
`dpropProvenance`	Provenance data pushed by a provenance-aware propagator.

Table A.6: Types of JSON objects used in PyDProp's provenance-aware cells

is why I use a special DProp-specific Python JSON library, DPropJSON, for both the DProp daemon and the PyDProp library.

DPropJSON operates like the `json` module provided by Python 2.6, which exposes two methods, `dumps` and `loads`, to encode and decode JSON respectively. DPropJSON depends on either the internal `json` module, if present, or the python-json library (also known as json-py) for versions of Python earlier than 2.6.

DPropJSON supports two additional data types in JSON. These data types, `URI` and `Nothing` are encoded as specifically formatted objects in JSON output. The `URI` type may be identified by the special key-value pair `__uri__=True`. It also has a key-value pair which specifies the value of the `URI`, `uri=`*`value`*. This type may be used to represent URLs.

The `Nothing` type, on the other hand, is simply identified by the key-value pair `__nothing__=True`. The `Nothing` type represents the empty value of a `Cell` before it has merged any updates; all `Cell` objects are initially set to contain a `Nothing` object.

Provenance-aware cells require several additional "special" types of objects. Although DPropJSON does not handle these objects, the provenance-aware PyDProp classes automatically work with these objects in the default merge functions for provenance sub-cells. Four additional object types are required for provenance-aware cells, which are specified in Table A.6. Each of these object types may be identified by a JSON object that has a key-value pair `type=`*`typename`*.

The `dpropMainCell` type has two additional key-value pairs, named `dataCell` and `provCell`. These two key-value pairs take values that correspond to the UUIDs of the data and provenance sub-cells of the provenance-aware cell respectively.

The types `dpropProvCell` and `dpropDataCell` also have two additional key-value pairs, `mainCell` and `data`. `mainCell` is a UUID pointer to the main sub-cell of the provenance-aware cell. `data`, in contrast, is assigned the actual wrapped value of the cell.

One additional class, `dpropProvenance`, is used to transmit provenance through provenance-aware propagators and differentiate a `dpropProvCell` from the provenance stored in the cell. This class has only one key-value pair, `data`. This key corresponds to the provenance data being forwarded to the destination provenance sub-cell.

Bibliography

[1] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In *Provenance and Annotation of Data: International Provenance and Annotation Workshop, IPAW 2006*, volume 4145/2006 of *Lecture Notes in Computer Science*, pages 118–132, Chicago, IL, USA, May 2006. Springer.

[2] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[3] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[4] Baruch Awerbuch and Shimon Even. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 278–281. ACM, 1984.

[5] Richard E. Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton University Press, Princeton, NJ, USA, 1961.

[6] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738: Uniform Resource Locators (URL), December 1994.

[7] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[8] Uri Braun and Avi Shinnar. A security model for provenance. Technical Report TR-04-06, Computer Science Group, Harvard University, 2006.

[9] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330, London, UK, 2001. Springer.

[10] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, May 1974.

[11] Stuart Cheshire and Marc Krochmal. DNS-based service discovery, March 2010.

[12] Sérgio Manuel Serra da Cruz, Patrícia M. Barros, Paulo M. Bisch, Maria Luiza Machado Campos, and Marta Mattoso. Provenance services for distributed workflows. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 526–533. IEEE Computer Society, 2008.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*. USENIX Association, 2004.

[14] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12. ACM, 1987.

[15] Robert B. Doorenbos. *Production Matching for Large Learning Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, January 1995.

[16] L. Dusseault. RFC 5789: Patch Method for HTTP, March 2010.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, June 1999.

[18] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, Irvine, CA, USA, 2000.

[19] Charles Lanny Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, February 1979.

[20] Daniel P. Friedman and David S. Wise. Cons should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284, Edinburgh, UK, 1976. Edinburgh University Press.

[21] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake Picasso: Preventing history forgery with secure provenance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 2009)*. USENIX Association, 2009.

[22] Rohit Khare and Richard N. Taylor. Extending the representational state transfer (REST) architectural style for distributed systems. In *Proceedings of the 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004.

[23] P. Leach, M. Mealling, and R. Salz. RFC 4122: A Universally Unique IDentifier (UUID) URN namespace, July 2005.

[24] P. Mockapetris. RFC 1034: Domain names - concepts and facilities, November 1987.

[25] P. Mockapetris. RFC 1035: Domain names - implementation and specification, November 1987.

[26] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga. The provenance of electronic data. *Communications of the ACM*, 51(4):52–58, March 2008.

[27] Steve Munroe, Simon Miles, Luc Moreau, and Javier Vásquez-Salceda. PrIMe: a software engineering methodology for developing provenance-aware applications. In *SEM '06: Proceedings of the 6th International Workshop on Software Engineering and Middleware*, pages 39–46, Portland, OR, USA, September 2006. ACM.

[28] Arvind Narayanan and Vitaly Shmatikov. De-anonymizing social networks. In *30th IEEE Symposium on Security and Privacy*, pages 173–187, Oakland, CA, USA, 2009. IEEE Computer Society.

[29] Robert H. B. Netzer, Timothy W. Brennan, and Suresh K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 31–40, Philadelphia, PA, USA, 1996. ACM.

[30] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[31] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: A review. *ACM SIGKDD Explorations Newsletter*, 6(1):90–105, 2004.

[32] Alexey Radul. *Propagation Networks: A Flexible and Expressive Substrate for Computation*. PhD thesis, Massachusetts Institute of Technology, 2009.

[33] Alexey Radul and Gerald Jay Sussman. The art of the propagator. Technical Report MIT-CSAIL-TR-2009-002, MIT Computer Science and Artificial Intelligence Laboratory, January 2009.

[34] Can Sar and Pei Cao. Lineage file system. Department of Computer Science, Stanford University. http://crypto.stanford.edu/˜cao/lineage.html, 2005.

[35] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, 1999.

[36] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31–36, September 2005.

[37] Mukesh Singhal. Update transport: A new technique for update synchronization in replicated database systems. *IEEE Transactions on Software Engineering*, 16(12):1325–1336, December 1990.

[38] Martin Szomszor and Luc Moreau. Recording and reasoning over data provenance in web and grid services. In *On the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888/2003 of *Lecture Notes in Computer Science*, pages 603–620. Springer, 2003.

[39] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–182, Copper Mountain, CO, USA, 1995. ACM.

[40] UPnP Forum. UPnP device architecture 1.0, April 2008.

[41] UPnP Forum. UPnP device architecture 1.1, October 2008.

[42] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable distributed reasoning using MapReduce. In *Proceedings of the ISWC '09*, volume 5823 of *Lecture Notes in Computer Science*. Springer, 2009.