A RESTful Messaging System for Asynchronous Distributed Processing

Ian Jacobi and Alexey Radul
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{pipian,axch}@mit.edu

ABSTRACT

Traditionally, distributed computing problems have been solved by partitioning data into chunks able to be handled by commodity hardware. Such partitioning is not possible in cases where there are a high number of dependencies or high dimensionality, as in reasoning and expert systems. This renders such problems less tractable for distributed systems. By partitioning the algorithm, rather than the data, we can achieve a more general application of distributed computing.

Partitioning the algorithm in a reasonable manner may require tighter communication between members of the network, even though many networks can only be assumed to be weakly-connected. We believe that a decentralized implementation of propagator networks may resolve the problem. By placing several constraints on the merging of data in these distributed propagator networks, we can easily synchronize information and obtain eventual convergence without serializing operations within the network.

We present a RESTful messaging mechanism for distributing propagator networks, using mechanisms that result in eventual convergence of knowledge in a weakly-connected network. By enforcing RESTful design constraints on the messaging mechanism, we can reduce bandwidth usage and obtain greater scalability in heterogeneous networks.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications, Distributed databases; D.1.3 [**Concurrent Programming**]: Distributed programming; H.3.4 [**Systems and Software**]: Distributed systems

General Terms

Design, Reliability

Keywords

propagator networks, REST, distributed computing, consistency, synchronization

1. INTRODUCTION

As cheap commodity computers have become more powerful, there has been a corresponding explosion in the number of distributed computing efforts making use of large networks of commodity hardware to perform complex computation. Heteogeneous distributed systems such as SETI@home [?], the related BOINC platform [?], and Google's MapReduce implementation [?], have been used to solve many otherwise intractible problems in recent years.

Many of these efforts have relied on partitioning the domain of a problem into smaller, more tractable chunks. Individual hosts then may perform identical processing on each data partition to obtain partial results that are later merged to produce a final result. Unfortunately, data partitioning in this manner may not be feasible for all problems. Problems that feature a high degree of dimensionality and have a high number of data dependencies in the final solution are well known to be difficult to partition. [?, ?]

Expert systems and reasoning engines, for example, may depend on a sizeable number of logical statements to calculate a meaningful answer to a query. Although reasoning with just a few simple pattern-matching rules has been done successfully with distributed algorithms [?], more complex rule systems may have a larger number of dependencies in a rule which may not fit into a single data partition.

One solution to such problems in reasoning is splitting the algorithm into several "rounds" of reasoning over different rules, but this merely trades space for time. Another possible solution may not require such a trade off. In the Rete algorithm [?, ?], a well-known algorithm for forward-chaining reasoning, a network of nodes is used to separate pattern-matching filters from a merge function over the outputs of several filters and other merge functions. Rather than dividing the problem by partitioning data across multiple nodes, the Rete algorithm suggests that we divide the algorithm itself.

One issue arises when applying this solution more generally: traditional distributed computing efforts like MapReduce and BOINC assume a weakly-connected network of nodes that only check in with a central server when a given subtask is complete. Reducing communication in this way strongly divides the domain so that tasks that rely on continuously updating inputs, such as reasoning, are infeasible. A loosely-connected decentralized model could prove to be more flexible than centralized architectures, but would be dependent on more frequent communications. We believe that a distributed variant of the data propagation model of computation [?] may provide such an architecture.

Unfortunately, data propagation as generically described makes no constraints as to how strongly connected the network of propagators is. Many of the examples Radul presents assume the existence of a tightly-linked computational platform. This assumption may not hold in many distributed computing scenarios that rely on a weakly-connected network. By utilizing several key constraints on propagator networks, we can safely eliminate this assumption; instead, we need only assume reliable broadcast and convergence of knowledge in a weakly-connected network, which is known to be possible. [**?**, **?**]

Such a significant change to the implementation of distributed systems also requires careful reconsideration of the underlying technologies used by the systems. While traditional centralized models have tended to rely on simple client-server protocols, a decentralized, data-driven model like data propagation requires a different architecture. Given that propagator networks gradually refine data in distinct "cells", we believe that treating such cells as resources in a system using Representational State Transfer (REST) may give a combination of flexibility and simplicity of design that may not be possible with other implementations. Hence, we have constructed a "RESTful" implementation of a distributed propagation system, named DProp[1].

In this paper, we first introduce the concept of propagator networks in Section **??**. Section **??** gives a brief overview of the REST architecture, while Section **??** outlines the design choices we made to construct our RESTful implementation of propagator networks. We then discuss mechanisms that reduce the incidence of race conditions (Section **??**) as well as possible security mechanisms within our system (Section **??**). Finally, we review related work in this field.

2. DATA PROPAGATION

The data propagation model of computation [**?**, **?**] is a concurrent message-passing programming paradigm. Data propagation operates over a network of stateless computational "propagators" connected by a number of stateful "cells" that may be used for both input and output.

Although superficially similar to traditional memory, cells normally store a single partial value that may be refined, but not overwritten. The initial state of a cell is an "empty" partial value, in which no assumptions may be made as to the cell's contents. Propagators connected to the cell may forward an update to it to add to the partial value stored within.

Upon receiving an update to its contents from a neighboring propagator, a cell will apply an appropriate merge operation to unify the existing partial value with the new information in the update. These merge operations must not only be designed with consideration of the data-types being merged, but also the intended contents of the cell, so that a set intersection operation is not applied where a set union is needed.

For example, if we assume we are constructing a reasoner using propagator networks, a cell might wish to contain a set of facts about animals. Upon receiving a new set of facts about dogs via an update, the cell may merge it into its own set by taking the union of the new dog facts with those already known and set that as the new contents of the cell.

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

While a naïve implementation might merge facts using set operations, we may use any arbitrarily interesting function for the merge operation. If a cell properly supported reasoning about its contents, we could eliminate previously known facts if they are subsumed by a new one. For example, explicit statements about two dogs, Rex and Fido, both having four legs might be able to be eliminated if a new statement stating that "all dogs have four legs" is learned. By allowing arbitrary merge operations, we can obtain greater power and expressivity in the propagator paradigm.

In order to ensure predictable computation in propagator networks, all merge operations must have four key properties:

1. **Idempotence:** The number of times a particular update is merged into a cell has no impact on the results of the merge. Knowing a fact and being told that fact again in an update doesn't mean we know anything new.

2. **Monotonicity:** Once a particular update is merged into a cell, no update should be able to directly unmerge that update. While the contents of a previous update may be marked as contradictory or disproven, it should not be possible to simply undo the update. In short, things once learned by a cell are not forgotten.

3. **Commutativity:** The order of merge operations in time is irrelevant. If we are told that "the cat is brown" and "the dog is black", we eventually know both facts regardless of order.

4. **Associativity:** Updating a cell with two separate update messages should have the same result as updating the same cell with a message that is the sensible "merge" result of the two. For example, if we are told that "Fido is at least 3 years old" and that "Fido is at least 5 years old", our knowledge should be the same as if we had just been told "Fido is at least 5 years old."

Each of these properties has influenced the structure of our system, and we will refer to them throughout the paper, summarizing their effects at the end of the paper.

These constraints are not as restricting as they may seem. Many functions that do not adhere to these constraints may be retrofitted to work in a propagator system. Much work in the design of distributed databases and internet protocols has attempted to satisfy demands for serializability of operations with the unreliability of the Internet, which may result in data being received out of order or not at all. By doing so, non-communtative operations may be performed despite the implicit commutativity of of packet ordering in the Internet [**?**, **?**, **?**, **?**].

Once a cell's content has changed, any propagators that directly depend on the cell wake up and perform further processing, as in Figure **??**. These propagators may send updates to additional cells, prompting yet more processing. For example, suppose we have a pattern-matching propagator attached to a cell containing facts about animals. This pattern-matcher will look at facts in that cell and then copy those of the form "x is brown" to another cell containing knowledge about brown animals. When some facts about dogs are merged into the first cell, the pattern-matcher will fire and any new knowledge about dogs that are brown will

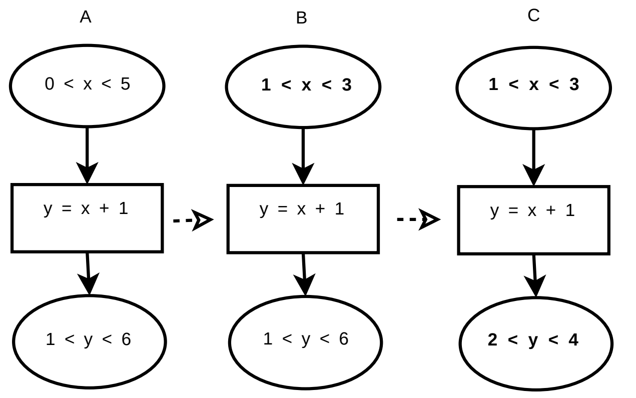

Figure 1: A propagator system at work with time increasing from A to C. Note that the updated constraints on x in the top cell in B cause the $y = x + 1$ propagator to wake up and update the constraints in the bottom cell in C.

be sent as an update to the "brown animals" cell. It will then merge that knowledge with the knowledge it already holds, may cause other propagators to fire, and so on.

Propagator networks may be cyclic, leading to complex loops and tail-recursive computation. As both single propagators and groups of propagators are separated by cells on their boundaries, we may also be able to treat a group of propagators as a single abstract propagator in its own right. In this way, we can actually instantiate them as needed, allowing for recursive processing.

The separation of computation from data in propagator networks explicitly modularizes computation. No time ordering is explicitly defined over the execution of propagators other than implicit dependencies caused by the order of cell changes. This makes propagator networks inherently concurrent and suggests a suitability to distributed systems.

3. THE REST ARCHITECTURE

Representational State Transfer, or REST, is an architectural style rooted in the concept of hypertext and the HTTP protocol.[**?**] The REST architecture aims to model resources as entities strictly capable of being created, read, updated, and deleted, also known as the CRUD operations. These operations are to be carried out through the transfer of *representations* of a resource. More complex operations may be modelled in a RESTful system in terms of these more fundamental resource operations. By restricting the number of operations that may be performed within the system, REST ultimately reduces API complexity.

Five primary constraints serve as the basis of a RESTful architecture, including:

1. a *client-server model* that separates data storage on a server from local display or handling of content

2. a *stateless design* such that any communication contains all information necessary for processing

3. *cacheability* of content to reduce network usage

4. *uniformity of interface* which reduces complexity of client implementations

5. and a *layered architecture* that helps to modularize network structure and treat it independently from the application structure.

Applications with a RESTful design are commonly implemented using HTTP, and ideally align the CRUD operations of creation, retrieval, updating, and deletion with the HTTP methods of PUT, GET, POST, and DELETE respectively.[2] For example, the content of a RESTful resource delivered with HTTP may be updated simply by submitting a POST request containing the information needed to update the resource.

By relying on the common HTTP protocol, RESTful applications are not only able to rely on existing software libraries, but also make themselves useful within the context of the World Wide Web. This allows applications to refer to external resources through the use of the HTTP URIs that identify them.

We choose to implement propagators using a RESTful architecture for a number of reasons:

1. Propagation maps nicely to a RESTful model. The stateless constraint on RESTful applications mirrors the inherent statelessness of the propagators themselves and the atomicity and associativity of cell updates make them particularly well-suited for implementation in a stateless architecture.

2. As cells are the only objects that contain state within propagator networks, it seems appropriate to model them as a class of resources in a RESTful architecture. This reduces the complexity of the implementation. This is only made more appropriate by the fact that only two primary operations, reading and updating, happen to cells.

3. The uniformity-of-interface constraint ensures that distributing propagators with RESTful techniques will be an easily expandable system. While our implementation, DProp, relies on Python, one may envision a a propagator network consisting of some nodes running Python on Linux, others using JavaScript in a web browser, and yet others using the .NET framework in Windows. By adhering to a RESTful architectural style, we ensure that such heterogenous systems are more easily constructed.

Together, these facts make REST a suitable architectural model for distributed propagation.

4. A RESTFUL PROPAGATION MODEL

As we intend to distribute the propagator model using RESTful techniques, we must first decide what the nodes in the physical network represent. We choose to treat hosts on the physical network as containers of both propagators *and* the cells that these propagators use to compute. In order to link hosts and properly distribute computation, we may simply use a simple network communication algorithm as a propagator designed to synchronize the contents of cells on

[2]Many variations exist, including those that account for web-browsers that may not support the PUT and DELETE methods, and several caused by confusion about the precise meaning of the PUT and POST methods.

multiple hosts. This may be done by simply forwarding any new updates observed by one cell to all other cells.

Two primary mechanisms for achieving such synchronization of cells seem to be likely candidate architectures for any such system:

1. We may choose to use a *client-server architecture.* In such a model, one host acts as the canonical server of a cell, in charge of performing merges and maintaining the canonical representation of the cell. All other hosts wishing to use the data stored in the cell act as clients, sending update messages to the server for its consideration. These hosts must remotely fetch data from the canonical representation following a successful merge.

   While this model simplifies the cost of maintaining the network with a simple star topology, use of a single canonical server means that this model is prone to failure of a single node (the server) causing a halt to all computation.

2. We may choose to use a *peer-to-peer architecture.* In this model, every host interested in a cell acts as a server for its own copy of that cell, and a client to other copies. Provided that there is a reliable method of both registering interest in a cell and forwarding updates to all hosts, any update to a cell should eventually be synchronized across all hosts.

   This model is more difficult to maintain, as it may have more complex network topologies, with a fully-connected network being the most efficient, but is not prone to complete failure caused by the failure of any one node.

   Choosing a peer-to-peer architecture *need not* imply that we must abandon the client-server constraint specified by REST. Instead, we may consider the peer-to-peer model to be an overlay model for how updates are propagated to remote cells. Similar to how web applications may invoke server-side methods that act as clients to other servers, we may employ a server and client acting in concert to manage a local copy of a cell to achieve the desired peer-to-peer propagation of updates.

While both models have their strengths and weaknesses, we believe that the gains to be found through decentralization, such as an increased tolerance of arbitrary network topologies and increased redundancy of connections, outweigh the costs of ensuring that synchronization will eventually occur in all hosts in a timely manner. We thus chose to implement DProp as a RESTful peer-to-peer-like system for cell synchronization.

4.1 Resource Representations

In constructing a RESTful system, we should note the kinds of objects modeled in our system as resources. Though propagators are generally stateless and need not be represented as resources, we do need to model the cells and the peers that have an interest in them.

The peer-to-peer model of synchronization introduces several complications to what would otherwise be an intuitive modelling of cells in a RESTful architecture. As mentioned previously, RESTful architectures assume the constraint of a client-server model. As a result, a resource must be identified uniquely only within the context of the server, rather than globally. HTTP somewhat alleviates this problem by providing a URI based not only on the local identifier but also the identifier of the server. Taken together, these two components may identify a resource globally.

This is ideal for our purposes, as we treat cells as being contained on individual hosts, rather than spanning them. This allows us to more easily identify distinct cells. A complication arises when we wish to associate the cell with a specific “synchronization propagator,” which *does* span hosts.

If we were to simply implement this naming scheme, we would lack a way to easily identify which particular group of cells a cell is synchronizing with, short of listing those cells in the group. This means we would lack a method to describe relationships between different groups of cells and may unnecessarily complicate the process of manually analyzing a distributed propagator network.

We resolve this by giving each synchronization propagator a universally unique identifier (UUID)[**?**] so that distinct cells connected by such a propagator may be referred to with a single identifier. We include the UUID in the URIs of the copies of the cell, so we may identify the synchronization group that each representation belongs to.

In order to more easily add peers to a network, we must also represent each peer of a cell as a resource, or part of a resource. These resources must be able to be created or modified by new peers. Thus, each cell copy has a collection of “Peers” containing the URIs of other peers of the cell. This collection may then be modified as needed.

Each peer would only keep track of the peers it is interested in, so any of a number of arbitrary network topologies of the peers is possible. Although the propagator network needs only to be eventually connected, certain topologies are more preferrable to others: an increase in the number of links in the network will increase timeliness of update messages due to a reduced network diameter; it will similarly increase failure tolerance due to replication of updates sent on those links.

Our implementation is designed to maintain a clique. By assuming existence of a clique, we may eliminate the additional complexity of routing update messages and still ensure that the number of messages needed to propagate an update across the network is $O(n)$, the number of nodes. Even if a clique is not maintained, our protocol will still ensure convergence, although timeliness is not guaranteed for nodes farther than distance 1 from the updated node.

4.2 Initializing a Cell

Creating a brand new cell and making it globally available is rather simple. After creating the local storage for the cell and associating the corresponding merge operation with it, the host need only mint an UUID for the new synchronization propagator for the cell and assign it a URI. Local updates may happen immediately following creation of the local cell, while the URI of the cell as a resource must be made available before others may synchronize.

Joining an existing network of cells is less trivial. First, one must locate a copy of the remote cell to initialize from. We will assume that this has already been done and a URI has already been obtained. We make this assumption as we believe the process of peer discovery to be independent of the problems of initialization and synchronization.

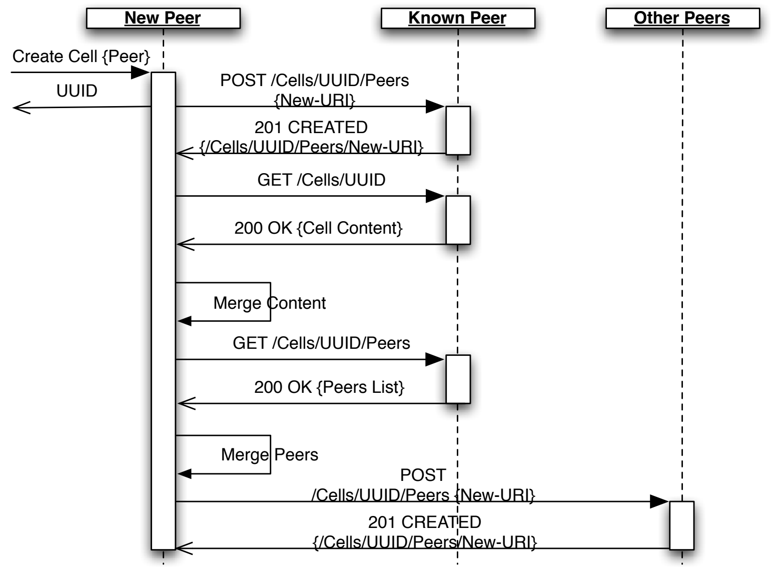

Figure 2: Connecting a new cell to an existing group of synchronized cells. Note that all actions are driven by the connecting peer.

Basic initialization follows a simple algorithm, depicted in Figure **??**. A propagator wishing to connect to a remote cell will first create the cell as above, but assign it the UUID of the remote cell. Once the URI of the cell has been created, the cell submits a POST request to the collection of "Peers" of the cell held by the remote host. This POST request will contain the URI of the new peer's copy of the cell, and serves to add the peer to the network of peers "listening" to the cell. The remote peer will now be able to forward any updates to the new peer.

The copy of the cell is then synchronized with the remote peer by performing a GET request to the URI of the remote host's cell. The response will contain the contents of the cell on the remote host, and may be merged into the local copy of the cell. This may cause propagators local to the new peer to fire. Further GET requests are made to initialize the contents of the "Peers" collection from that known by the remote host. This completes the copy of the cell.

After the local peer copies the collection of peers, it submits a POST request to each peer in the newly updated "Peers" collection. This way, the new peer becomes visible to all other peers in the network, much as it did to the initial remote peer. These POST requests will result in the creation of a new, larger clique of synchronizing cells.

Note that we do not require any locking mechanism during the initialization of the cell. The associativity and idempotency constraints on the merge operation ensure that the merge of data from the remote peer will only increase the amount of knowledge known by the cell. The firing of local propagators following the merge also ensures that any change of the cell caused by the update will still generate meaningful computation once something is known by the cell.

4.3 Updating a Cell

As a propagator processes, it may send an update to another cell. It will do so by sending an update on the local host, which will then merge the new fact before causing any connected propagators to fire. In a distributed propagator network, the custom merge operation is actually used to complete a generic stub which additionally sends that fact as a POST request to all cell copies known in the local peer's "Peers" collection, as in Figure **??**. In this way, the local peer that made the update also informs other peers about the new fact it has learned and lets them handle merging it.

After receiving a POST request representing the update, a cell will first confirm that the POST request came from a known peer in its "Peers" collection, so that facts cannot be tampered with by someone who is not trusted. If the update does come from a known peer, it will merge the newly received update message using the merge operation and, unlike local merging, will not do any further notification of nodes on the network. Finally, it will wake up any local propagators to perform any computation, just as if the update had been generated locally.

We assume that the merge algorithms on each peer are identical. This allows us to only transmit the updates across the network, rather than the fully merged data. Furthermore, the commutativity constraint on the merge operation implies that the order in which messages are received is irrelevant to the contents of the cell. This eliminates the need for synchronized timestamps across the entire network. Finally, the clique embodied by the interested peers ensures that the number of update messages sent is $O(n)$, the number of peers in the clique. This reduces communication within the system as well as the diameter of the network for iterative algorithms.

5. FAILURE RECOVERY

Although this basic system is functional at first glance, there are a number of potential points of failure in the system. The most obvious points of failure are links in the physical network. While use of HTTP over TCP makes the

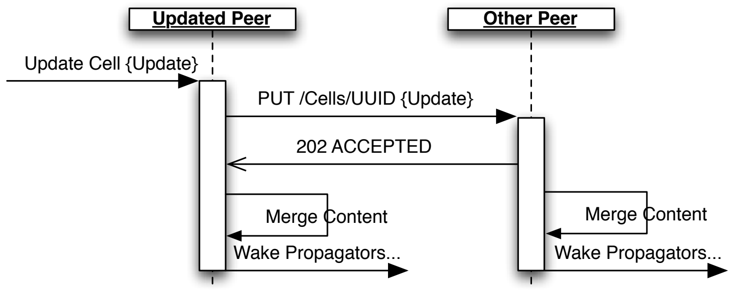

Figure 3: Process of propagating an update to a cell. Note that all actions are driven by the updating peer.

system somewhat tolerant of communication errors, routing irregularities and inconsistent uptime of connections must still be accounted for to ensure eventual convergence of the cell contents.

Recovery from such errors relies on the associativity and idempotency constraints on the merge operation. These allow for contents of cells to be merged without having seen the same update messages. Each peer interested in a cell will occasionally perform a full synchronization of its contents with all peers it is aware of. This synchronization consists of a number of GET methods applied to the remote peers' cells and "Peers" collections. The results of these GET requests may then be merged into the existing cell.

Synchronization in this way allows the local peer to determine connectivity to its set of peers. It also allows it to properly correct any possible loss of synchronization that may have been caused by messages that did not manage to reach the peer, without needing to resort to caching a history of all update messages on each node.

This operation is likely to be quite wasteful of bandwidth, as an identical resource representation may be obtained from each peer. The RESTful constraint of cacheability proves useful in reducing such waste. HTTP provides several headers that assist with client-side caching. Each HTTP resource may be served with an ETag, which uniquely identifies a particular state of a resource. When a resource changes, its ETag should change as well to uniquely identify the new state.

Requests made to an HTTP server may provide the If-None-Match header with a previously cached ETag value of the cached resource. If the ETag of the resource matches the If-None-Match header, the server may respond with a simple "304 Not Modified" message and no content. This indicates that the content in the client's cache remains the most recent version of the resource. We may thus use the ETag and If-None-Match header in our GET requests to ensure that data is only transferred when there is an inconsistency, rather than on every GET.

Should a link fail and a peer of the network lose connectivity, the loss of update packets between the peer and other members of the clique does not mean that processing is forced to halt. Assuming connectivity is achieved again, the above synchronization operation will suffice to converge the system. Furthermore, failure of a peer may simply be treated as if all links to the node have failed and no further processing is done. Reconnecting to the network will resolve any global inconsistencies that arise.

6. REDUCING RACE CONDITIONS

While race conditions are a natural byproduct of a concurrent system, our implementation of distributed propagation has several features that help to reduce the number of race conditions possible. The idempotency and commutativity constraints eliminate the harm from double updates and out-of-order updates, while several other race conditions are ameliorated by other means.

Most remaining race conditions are resolved through the synchronization mechanism. For example, suppose that a remote peer has not yet merged a new peer into its "Peers" collection. Should it receive a POST message from the new peer, it will drop the update as being from an unrecognized host. However, if the new peer has been registered with at least one other peer, the synchronization mechanism will ensure that the remote peer that dropped the update will not only eventually be made aware of the new peer, but also of the update that was dropped.

Despite the power of synchronization, there still exist several instances where inconsistencies may arise. For example, a remote peer may be down when the local peer attempts to connect, or the desired cell may not exist on that remote peer. In this case, the local peer is never able to properly join the network in the first place, and may be inconsistent with the network from the start. The addition of a mechanism that allows connecting to a cell through alternate peers, or simply retrying such connections may allow the local peer to connect to the cell anyway, after which the synchronization mechanism will resolve any inconsistencies between the disconnected cell and the other cells.

7. SECURITY

A practical distributed propagator system must be able to ensure the security of data within it. While a cell containing animal facts may unworthy of security, a similar cell could be used to share classified information which a government has a vested interest in keeping secure.

As our implementation uses HTTP as the substrate for operation, there are several possibilities for securing data. We may choose to secure the protocol through the use of SSL or TLS; we may also choose to secure the data by encrypting the contents of the cells themselves. We believe that a combination of the two approaches is necessary to achieve sufficient security.

Encryption of the protocol with SSL or TLS will defend communications against man-in-the-middle attacks and provide a mechanism for non-interactive authentication. How-

Constraint	Benefit
Idempotence	(with associativity) removes need for locking on cell initialization
	(with associativity) permits synchronization procedure to simply exchange knowns
Associativity	(with idempotence) removes need for locking on cell initialization
	(with idempotence) permits synchronization procedure to simply exchange knowns
Commutativity	removes need for global timestamps
	removes timeliness constraint on communications
Monotonicity	removes need to account for deletion of information
	allows for computation of results using partial knowledge

Table 1: Benefits of the four constraints on the merge operation within this synchronization system

ever, SSL/TLS itself is insufficient as multiple distinct propagators may use the same port on the server to host their cells. As SSL/TLS certificates are presented before the cell itself is requested, it is difficult to confirm that a particular cell is actually "maintained" by any particular user on the shared instance, such as in the following scenario:

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 URI 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 identifying certificate is provided to Alice upon connecting to the DProp peer, even if Bob and Eve were to have separate client certificates for when they sent updates to Alice.

There are a number of ways to resolve this problem. We may enforce a permanent reservation system for cell identifiers, or instead have a secondary identifier used to contain information about the cell's owner. The most foolproof method would be to encrypt the updates themselves. By encrypting them, Alice and Bob could ensure that only the propagator with the correct key would be able to decrypt updates, despite the limited level of security granularity offered by SSL/TLS.

While this double encryption requires additional overhead, we cannot simply do away with SSL/TLS either. Without SSL/TLS, the identity of the cells contacted could not be encrypted without abandoning HTTP. By adding SSL/TLS as an additional security layer below HTTP, we can ensure that this meta-data is not subject to man-in-the-middle attacks. As a result, we must perform double encryption if we are to maintain compatibility with the HTTP standard.

8. RELATED WORK

The work presented in this paper bears strong similarities to work done in the field of database replication. Basic database replication approaches would be sufficient for a cell synchronization system if we assumed that cells are nothing more than simple databases with data rows that correspond to the updates received by the cell. However, such techniques tend to adhere to constraints that are unnecessary for propagator networks, such as serializability, used in weighted voting [?], and the non-monotonicity of row deletions.

Update propagation mechanisms for database replication, such as those developed for the Grapevine system [?] are similar to the mechanisms described here. Like the synchronization mechanism above, Grapevine seeks eventual convergence of knowledge in the network, rather than guaranteeing immediate convergence. Unlike Grapevine, however, we integrate the database more tightly with the messaging system on a single host, rather than implicitly allowing for their separation. Furthermore, our system has an explicit mechanism for resolving inconsistencies as part of the synchronization protocol.

The Bayou system [?] is in some ways more similar to our system than Grapevine. Unlike Grapevine, it guarantees convergence on a weakly-connected network. Nevertheless, its guarantee of eventual serializability of updates leads to the construction of a centralized system of "soft writes" which our system neither requires nor implements. Furthermore, Bayou again concerns itself with the problem of non-monotonicity caused by deletions which is not required for cell synchronization.

Singhal [?] provides an algorithm that is similar to the synchronization protocol described here, allowing for the synchronization of replicated databases through update distribution. However, like in Bayou, Singhal also adheres to a serializability constraint that is unneeded for propagators. Furthermore, Singhal does not assume a weakly-connected network as DProp does.

Decentralization of RESTful practices has been proposed in Khare and Taylor's ARRESTED architecture [?]. While its principles provide an important basis for our implementation, the ARRESTED architecture ultimately implements several features that are unnecessary for a propagator network, such as estimation, locking, and routing. As propagator networks are constrained to be monotonic, there is no need to guarantee receipt of an update or to estimate a cell's content.

9. CONTRIBUTIONS & FUTURE WORK

We have implemented and demonstrated a RESTful data propagation system that permits useful distributed computation with a weakly-connected network. In doing so, we avoid constructing a centralized model that is subject to point failures. It is our belief that a system such as this will prove to be a viable computational platform that provides a greater flexibility than that offered by traditional client-server architectures for distributed computing like that of BOINC or other similar grid architectures.

We have also identified four constraints of the merge operation used in propagator networks, idempotence, monotonicity, commutativity, and associativity. These constraints give us a number of advantages, described in Table **??**, that allow us to simplify our system and provide greater redundancy and flexibility than existing approaches to distributed databases.

Although DProp has been tested in small propagator networks to demonstrate the feasibility of RESTful distributed propagation, we have not yet tested it in larger networks. We also have not yet fully implemented the security protocol described in this paper and tested it within a working propagator network. In order to test the system with a large-scale application, we currently intend to implement an application that manages information sharing with provenance using the DProp framework.

We believe a more complete analysis of the proposed security mechanisms is warranted following implementation of the security component in DProp, as we have only performed a cursory inspection of possible security concerns with the proposed system at this time. This analysis may require migration away from a strict HTTP implementation to avoid the double encryption issue mentioned above, as well as any other issues that arise with the use of SSL for this protocol. We are also unclear on the overhead that an HTTP client-server model incurs for this implementation, although initial tests of DProp have not exhibited significant issues with overhead when propagating small updates.

10. ACKNOWLEDGEMENTS

We would like to thank Lalana Kagal, Gerry Sussman, Hal Abelson and other members of the Decentralized Information Group at MIT for their advice and criticism throughout the entire process of refining the ideas discussed here and putting them down on paper. We would also like to acknowledge that this work was supported in part by the National Science Foundation under NSF Cybertrust award number CNS-0831442 and IARPA award number FA8750-07-2-0031.

11. REFERENCES

[1] D. 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.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[3] B. Awerbuch and S. 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.

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

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

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

[7] J. Dean and S. 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.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. 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.

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

[10] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] C. L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, February 1979.

[12] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–162. ACM, 1979.

[13] R. Khare and R. 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.

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

[15] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: A review. *ACM SIGKDD Explorations Newsletter*, 6(1):90–105, 2004.

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

[17] A. Radul and G. J. Sussman. The art of the propagator. Technical Report MIT-CSAIL-TR-2009-002, MIT Computer Science and Artificial Intelligence Laboratory, January 2009.

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

[19] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. 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.

[20] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using MapReduce. In *Proceedings of the ISWC '09*, volume 5823 of *LNCS*. Springer, 2009.