Reconciling while Tolerating Disagreement in Collaborative Data Sharing[*]

Nicholas E. Taylor Zachary G. Ives
Computer and Information Science Department
University of Pennsylvania
{netaylor,zives}@cis.upenn.edu

ABSTRACT

In many data sharing settings, such as within the biological and biomedical communities, global data consistency is not always attainable: different sites' data may be dirty, uncertain, or even controversial. Collaborators are willing to *share* their data, and in many cases they also want to selectively *import* data from others — but must occasionally *diverge* when they disagree about uncertain or controversial facts or values. For this reason, traditional data sharing and data integration approaches are not applicable, since they require a globally *consistent* data instance. Additionally, many of these approaches do not allow participants to make updates; if they do, *concurrency control* algorithms or *inconsistency repair techniques* must be used to ensure a consistent view of the data for all users.

In this paper, we develop and present a fully decentralized model of *collaborative data sharing*, in which participants publish their data on an ad hoc basis and simultaneously *reconcile* updates with those published by others. Individual updates are associated with provenance information, and each participant accepts only updates with a sufficient authority ranking, meaning that each participant may have a different (though conceptually overlapping) data instance. We define a consistency semantics for database instances under this model of disagreement, present algorithms that perform reconciliation for distributed clusters of participants, and demonstrate their ability to handle typical update and conflict loads in settings involving the sharing of curated data.

1. INTRODUCTION

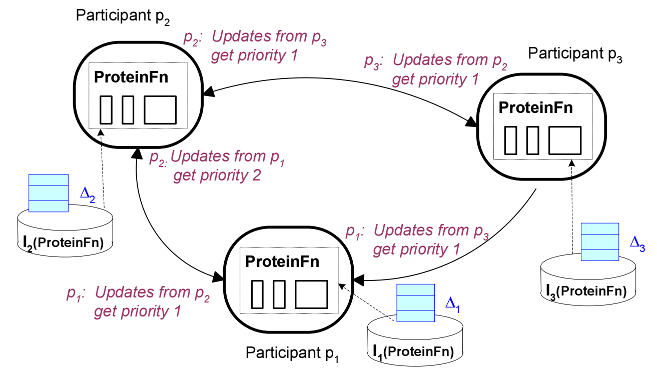
When multiple autonomous, collaborating parties agree to share data, they often encounter situations where that data is mutually *inconsistent*: each party has a data instance that is internally consistent, but the different parties may each have "facts" that conflict with the others. Such inconsistency arises from many reasons: sites may have different levels of data freshness, some sites' data may be dirty, or, frequently, the sites may have different *viewpoints* about what data is factual.

For instance, in bioinformatics, the same results from microarray experiments may be analyzed by different tools, or curated by different people, yielding different values. This is inevitable because most data curation is somewhat uncertain in nature, and thus correctness becomes a matter of interpretation. Therefore, bioinformatics data warehouse curators often reach different conclusions, and even occasionally revise these conclusions. Biologists have a sense of the authority of the different databases: for instance, SWISS-PROT [2] is generally more reliable than NCBI GenBank [24] because it is human-curated (see, e.g., [22] for a query language that incorporates such policies). To help control the quality of the data they use (and also to ensure their queries are not seen by competitors), most biologists create, query, and curate their own local database instances, which they populate with downloaded data. A biologist will typically not publish the contents of this database until he or she submits a paper for publication.

These situations result in a mismatch with existing methods of sharing data. If sharing is based on a common DBMS (whether distributed or client-server), the DBMS concurrency control and integrity checking routines will mandate that a single version be put into the database. Even in situations where local caching and optimistic concurrency control are used, ultimately one version of the state will exist. If, instead, the collaborating parties share information through a data integration [11, 20] or peer data management [13, 15] system, the results will be either inconsistent, or filtered through a "repair" scheme [1, 19]. Other strategies have similar drawbacks; e.g., file synchronization [26, 7] typically results in conflicts that must be manually resolved in order to restore consistency.

The solution to the needs of collaborating scientists is not to provide one globally consistent data instance, but rather to give each participant a *custom*, internally consistent instance with the data that this particular source accepts as authoritative. This instance should contain data "overlapping" with that at other sites, as well as certain data items that diverge from (many of) the others. In keeping with the needs of scientists as described above, data instances should be loosely coupled: each participant queries and manipulates a local database instance and may occasionally *publish* its database, including a log of recent changes; it may *selec-*

[*]This research has been funded by NSF CAREER award #IIS-0477972 and grant #IIS-0513778, and a seed grant from Penn ISTAR.



Participant p_2
p_2: Updates from p_3 get priority 1
p_3: Updates from p_2 get priority 1
Participant p_3
ProteinFn
ProteinFn
p_2: Updates from p_1 get priority 2
Δ_2
I_2(ProteinFn)
p_1: Updates from p_3 get priority 1
Δ_3
ProteinFn
I_3(ProteinFn)
p_1: Updates from p_2 get priority 1
Δ_1
Participant p_1
I_1(ProteinFn)

Figure 1: Collaborative data sharing system with three bioinformatics data warehouse participants sharing data on protein functions.

tively import others' modifications according to some policy. Today such capabilities can only be provided through custom import tools, which are expensive to develop and generally do not provide well-defined conflict resolution semantics.

The challenge in supporting this data sharing model lies in the fact that every database instance is subject to updates, in the form of transactions. Not all of these transactions may originate from sources trusted by all parties; hence, each site must decide whether to *accept* an update (in effect, synchronizing a portion of this site's data instance with the origin of the update), to *reject* it as being untrusted (causing a divergence), or in some cases to *defer* processing an update (if there are multiple conflicting updates with no clear preference). An update-centric model is important, as opposed to simply querying the resulting data values, because it allows sites to reject removals or replacements. However, it adds significant complexity, because one update may depend on a chain of other updates, some of which are trusted and others of which are not. A clean semantics for update propagation must consider these interactions, as well as different levels of trust.

At the University of Pennsylvania, we are developing a generic platform to support this type of data sharing; we refer to this platform as a *collaborative data sharing system* or CDSS. Our overall CDSS architecture was first presented in an overview paper [14], which sketched out our vision of confederations of autonomous data sharers who publish and import updates. The process of selectively importing updates (in particular, those that do not conflict with existing data) is termed *reconciliation*. The ORCHESTRA system represents our efforts to realize such a CDSS, and although our system is broadly applicable, we are particularly focusing on the needs of bioinformatics and biomedical researchers.

An ORCHESTRA system (Figure 1) consists of a number of collaborating participants ($p_1, \ldots, p_3$ in the figure), each of whom controls and edits its own data instance (denoted $I_1, \ldots, I_3$), and each of whom has a policy about what external data it is willing to trust and accept (labels on the arcs). As data is modified at different sites (denoted $\Delta_1, \ldots, \Delta_3$), ORCHESTRA publishes and propagates the updates to all sites that are willing to accept them. Our end goal is to facilitate update propagation in situations with both disparate schemas (i.e., each participant may have a different schema, with some attributes that may not have corresponding aspects in the other schemas) as well as disparate instances (i.e., each participant may have tuples with values that may not exist in other participants' instances).

This paper represents our first step towards addressing these goals: it defines a model and methodology for reconciling instances of a *single* database schema, in the presence of transactions, disagreement, and trust relationships. The complementary problem of translating updates across schemas in order to incrementally maintain data instances is ongoing work.

We make the following contributions:

- A formalization for reconciliation that emphasizes local autonomy, making forward progress, and providing intuitive behavior. We support *coexistence* of multiple instances, with consistency defined by *trust policies*.
- A model for reconciliation where individual transactions are assigned trust levels, and the goal is to propagate the highest-trusted, longest chains of transactions.
- New algorithms to reconcile database instances while maintaining consistency .
- A performance analysis of these algorithms on bioinformatics-based workloads.

This paper is structured as follows. We discuss related work in distributed data sharing in Section 2. We describe our CDSS model in Section 3. In Section 4, we define a semantics for reconciliation. Section 5 presents algorithms for implementing reconciliation. We validate our initial prototype implementation in Section 6, and finally we conclude and discuss future work in Section 7.

2. RELATED WORK

The problem of sharing structured data among multiple sites has a long history, and it has been revisited in many different settings. However, as described in the previous section, there has generally been an assumption that the end goal is to create a single consistent data instance (perhaps with multiple views) across all sources. We briefly outline previous work.

File synchronizers [26, 10, 16] take two files, and they attempt to reconcile the changes into a maximally consistent instance. When a conflict exists, both versions are typically preserved and the user must resolve the conflict before continuing to work. The Bayou [9] disconnected file system provides mechanisms for supplying *application-specific conflict detectors* and *conflict resolvers*. The Coda [30] and Ivy [23] file systems provide a mechanism for detecting conflicts, and disable updates to them until a *repair* is made.

Distributed databases provide a single administrative domain, database schema, and instance to all users. Each participant may independently make edits to the database instance. Through either *lock-based* or *optimistic* concurrency control schemes [17, 8, 5], the system interleaves updates while maintaining consistency. *Version vectors* [25], vector clocks [18] and other mechanisms are often used to determine causality. An alternative approach is that of Lotus Notes [21], which, upon a conflicting update, splits the data into two versions that are no longer synchronized.

Data integration (including federated databases and peer data management) is a middleware layer above autonomously maintained data sources. Here, each participant updates its own data sources in a self-contained fashion. When a participant wishes to access data from "the outside world," it poses a query over the virtual *mediated schema* that the data integration system builds over all data sources. In traditional data integration, there is one mediated schema; peer data management systems [13, 15, 4] allow for multiple mediated schemas, interrelated by a network of schema mappings. Data consistency is generally managed in terms of *certain answers*: inconsistent data values are removed through a "repair" procedure [1]. No facilities exist here for *importing* data into a given data source.

The *collaborative data sharing system* was first introduced in [14], which proposed (but did not implement) an architecture for supporting the exchange of atomic updates in a distributed setting. Our work in this paper builds upon these ideas, but focuses on the complexities introduced by transactions and different trust levels.

3. COLLABORATIVE DATA SHARING

As shown in Figure 1, a *collaborative data sharing system* consists of a confederation of loosely coupled *participants* (or peers), each of which is autonomous, but each of which wishes to share its data and updates with the other participants. In general, *update translation mappings* (not shown) relate instances of one published schema to instances of "semantically neighboring" published schemas, and provide a means of translating updates from one schema to another; however, in this paper, we do not consider update mappings, and instead focus on reconciliation in a single-schema setting. Therefore, in the figure, all participants share the same schema, though they may of course trust different data. Participants (p_1, p_2, p_3 in the figure) make updates to their local database instances, and they publish (a subset of) these updates ($\Delta_1, \Delta_2, \Delta_3$) as well as a *published database instance* (shown in the figure as I_1, I_2, I_3) with a matching schema. Finally, a series of *acceptance rules* (labels along the arcs between participants) define, for each participant, a *trust priority level* for updates from other participants.

The central problem in a CDSS is the propagation of updates among sites. We term this problem *reconciliation*: given the acceptance rules and updates published by participants, the reconciliation operation determines which updates should be applied to ("accepted by") the *reconciling participant p*. All updates that satisfy the acceptance rules and do not mutually conflict (or conflict with existing state) should be accepted; for conflicting updates, priorities are used to determine which (if any) updates are to be applied.

Conflicts. In our initial implementation, we adopt a fairly simple conflict model. A conflicting update is any update that: (1) when applied to the current database instance, results in an instance that is inconsistent with its integrity constraints; or (2) is mutually incompatible with some other published update that also satisfies an acceptance rule. Instances of the latter case are updates that change a single antecedent data value into two different values, updates that simultaneously remove and replace a data value, and updates that result in a data instance that violates a constraint (though the last can also be caused by a single update to a database instance).

We assume that reconciliation is an operation that is done frequently but not in real-time, by each specific participant: the participant will accept and apply a subset of all "recently published" updates to its data instance. Note that reconciliation is a matter of *importing* data, and therefore it can be done more or less frequently than publishing, though we assume that the two are performed together.

3.1 Preliminaries

Before providing an example of reconciliation in a CDSS, we begin with a description of our high-level goals and basic mechanisms.

- *Maximal progress and monotonicity.* Each reconciliation should make maximal use of all published updates available at the time. Once an update has been accepted by a participant, a future reconciliation may result in *changing* the results of the update, but the update itself will not be rolled back from the data instance.
- *Least interaction.* Update sequences made at participant p should not interact with update sequences made at site q in unexpected ways: in particular, if q makes a modification that conflicts with p, but revises its modification so it no longer conflicts, *before* p imports its changes, then p should consider q's update sequence to be compatible.
- *Trust policies.* In many bioinformatics and other settings, some sources are known to be more credible than others. (As mentioned previously, SWISS-PROT is human-curated, making it more authoritative than GenBank, which is not.) We allow for each site to provide a partial ranking of authority for such cases — allowing the system to automatically resolve certain conflicts.

We now describe how each of these principles guides the functionality of the CDSS, before we present an example.

Maximal progress and monotonicity. We define reconciliation in a *participant-centric* way, and assume a global ordering on when participants reconcile (and optionally publish). We term each such step an *epoch*. At each reconciliation, a single peer imports updates from outside. While it may also publish its own updates, no other participant will receive these until *it* reconciles. Thus, information flow is inherently (1) relative, as the updates participant p sees from participant q are those published since p last reconciled; and (2) asymmetric, as q will not immediately receive p's updates. Moreover, a reconciling participant has no way of knowing whether the updates it "sees" now will be revised in the future, or whether some other participant will publish a conflicting update in the future. We believe that a causality model in which every participant makes maximal progress based on what it has seen, and never "changes its mind," has many desirable properties.

A second mechanism for ensuring progress is the notion of *update deferral*. If several updates conflict and the participant has no way of ranking them, it will mark them as being deferred until a user resolves the conflict. Any future updates that might conflict with an unresolved conflict are themselves deferred — ensuring that the user does not inadvertently render them inapplicable.

Least interaction. The mode of information exchange (and hence the cause of dependencies occurring among updates) will solely be via acceptance of updates from other participants. Since two participants may reconcile at different frequencies, we believe that any *intermediate* states of tuples should not interact, i.e., if different participants make successive modifications to a tuple while not in contact with one another, any intermediate states should be disregarded, and only the final updates should be considered.

Trust policies. *Acceptance rules* assign a numeric priority level to a set of updates, based on predicates over the *content* as well as the *origin* of these updates. We assume that higher priority levels result in larger numbers, and that in situations where conflicts arise, a participant will accept an update with a higher update priority over a conflicting one with a lower priority. When multiple updates have equivalent (and highest) priority, our semantics is to adopt a "certain answers," open-world model in which *none* of the conflicting updates will be applied until a user intervenes. Such updates are termed *deferred*, and any future updates that conflict with a deferred update are themselves deferred.

In order to support acceptance predicates over update origins, we assume that every update is annotated with the identity of its origin. While this model is not as expressive as some notions of lineage [3, 6, 32], it is adequate for our acceptance rules. This model resembles the Information Source Tracking method of [29] and the multi-viewpoint formalism of [14].

3.2 CDSS Example

We refer to the example CDSS in Figure 1, where participant p_1 has a policy to accept update sequences from either p_2 or p_3, assigning them equal priority. In contrast, p_2 prefers updates from p_1 versus p_3, and p_3 only accepts updates from p_2. An exception to this rule, which we describe later, is that p_2 may make revisions to updates that originated from p_1 — in this case, p_3 must *transitively* accept this portion of p_1's data.

Notation. In this paper we assume that all updates are described in terms of *changes to values*, and they are annotated with the identifier of a single originating participant. We consider the following operations: insert tuple (denoted $+R(\bar{a}; i)$ for an insertion of the tuple $\bar{a}$ by participant i into some relation R with a schema compliant with $\bar{a}$); delete tuple ($-R(\bar{a}; i)$); modify tuple ($R(\bar{a} \rightarrow \bar{a}'; i)$, where $\bar{a}'$ is a new set of attribute values conforming to schema of R). We also assume that updates may be grouped into transactions, denoted $X_{i:j}$, where i represents the identity of the originator of the transaction, and j represents its unique local transaction identifier. We assume that transaction identifiers are assigned in increasing order.

Figure 2 illustrates reconciliation over four epochs within this CDSS, for a single relation F(*organism,protein,function*), where (*organism,protein*) is a key. At time 0, each participant p_i's instance of this relation, denoted $I_i(F)|_0$, is empty. In epoch 1, participant p_3 applies two transactions (one of which revises the other), and then it publishes and reconciles its data. Since no other updates have been published, p_e ends Epoch 1 with state $I_3(F)|_1$, obtained by applying its own update sequence.

In the next epoch, participant p_2 introduces two new tuples and then reconciles. Its resulting state, $I_2(F)|_2$, is the result of applying its own updates. Although p_3 published two updates that p_2 trusts, these updates conflict with p_2's own updates — hence, it rejects them. In Epoch 3, p_3 reconciles a second time. Now it applies the *mouse* update from p_2; it rejects the *rat* tuple that is incompatible with its own local state. Finally, in the last epoch, p_1 reconciles. It trusts p_3 and p_2 equally. Hence, it accepts the nonconflicting *mouse* updates, but it must *defer* the remaining *rat* update transactions because they all conflict.

Given our intuitions from the basic principles and the preceding example, we now proceed to define a formal semantics for reconciliation.

4. RECONCILIATION

We begin by specifying the collaborative data sharing system formally. In the larger scope of ORCHESTRA, we intend to support reconciliation across multiple schemas. However, for purposes of this paper, we will define the CDSS for a setting in which all participants share a single schema.

DEFINITION 1 (COLLABORATIVE DATA SHARING SYSTEM). *A* collaborative data sharing system *(CDSS) includes the following components:*

- *Σ, a schema representing the relations in the system.*
- *P, a set of participants, $\{p_1, \ldots, p_n\}$.*
- *$\mathcal{A}$, a mapping from each $p_i \in P$ to a set of* acceptance rules*, each of which is a pair (θ, v) where θ is a predicate on updates in Δ over some relation R and v is an integer priority that p_i assigns to tuples satisfying θ.*
- *Δ, a sequence of transactions of updates of the form $+R(\bar{x}; i)$, $-R(\bar{x}; i)$, $R(\bar{x} \rightarrow \bar{x}'; i)$, over each relation R and published by each participant p_i.*
- *$I(\Sigma) = \{I_1(\Sigma), \ldots, I_i(\Sigma), \ldots, I_n(\Sigma)\}$, the public database instances controlled by each p_i.*
- *e, an integer clock or* reconciliation epoch *counter. It is incremented each time a different participant publishes data. We assume that the first publication or reconciliation step defines the beginning of epoch 1. We denote the subset of Δ published in epoch e as $\Delta|_e$.*

Suppose we are given a CDSS as in Definition 1. Let us denote an update made to relation R as δ_R. We define the *priority relative to participant p_i of a transaction X*, $pri_i(X)$, as follows:

- 0, if any $\delta \in X$ is untrusted, i.e., there is no $(\theta, v) \in A(p_i)$ such that $\theta(\delta)$ is satisfied and $v > 0$.
- $max(\{v \mid (\theta, v) \in A(p_i) \wedge \theta(\delta) \wedge \delta \in X\})$, otherwise.

We say that two updates δ_R, δ'_R conflict iff

- δ_R, δ'_R are both insertion operations with the same values for their key attributes, but different values for at least one other attribute, or
- one of δ_R, δ'_R is a deletion and the other is a replacement or insertion operation, and they have the same values for their key attributes, or

Epoch	**Participant p_3**	**Participant p_2**	**Participant p_1**
0	$I_3(F)\|_0 = \{\}$	$I_2(F)\|_0 = \{\}$	$I_1(F)\|_0 = \{\}$
1	$X_{3:0} : \{+F(rat, prot1, \textit{cell-metab}; 3)\}$ $X_{3:1} : \{F(rat, prot1, \textit{cell-metab} \rightarrow rat, prot1, immune; 3)\}$ <publish and reconcile> $I_3(F)\|_1$: $\{(rat, prot1, immune)\}$		
2		$X_{2:0}\{+F(mouse, prot2, immune; 2)\}$ $X_{2:1}\{+F(rat, prot1, \textit{cell-resp}; 2)\}$ <publish and reconcile> $I_2(F)\|_2$: $\{(mouse, prot2, immune), (rat, prot1, \textit{cell-resp})\}$	
3	<reconcile> $I_3(F)\|_3$: $\{(mouse, prot2, immune), (rat, prot1, immune)\}$		
4			<reconcile> $I_1(F)\|_4$: $\{F(mouse, prot2, immune)\}$ DEFER: $\{X_{3:0}, X_{3:1}, X_{2:1}\}$

Figure 2: Reconciliation of $F(organism, protein, function)$, with key $(organism, protein)$, among the participants of Figure 1, over four epochs. Each participant p_i may apply transactions ($X_{i:j}$), which it publishes and reconciles according to the policies in Figure 1. The resulting instance for each epoch e is denoted with $I_i(F)|_e$. When transactions conflict, the participant always picks its own version first, or else the highest-priority one and its antecedents (if this is unique). It *defers* any transactions that have no unique "winner."

- δ_R, δ'_R are both replacement operations with the same source tuple value, where the replacement tuples have different values.

An update δ_R may also be incompatible with an instance I if applying δ_R to I would violate an integrity constraint. We generalize this to say that two transactions X, X' conflict iff an update $\delta \in X$ conflicts with an update $\delta' \in X'$, and that a transaction X is incompatible with an instance I iff an update $\delta \in X$ is incompatible with I. Finally, we assume a function $apply(\delta_R(\bar{x}; i), R)$ that applies the updates in $\delta_R(\bar{x}; i)$ to relation R, and returns the resulting relation. Within this setting, we define two versions of the reconciliation problem: first, the problem in an append-only setting, and then the problem in its full generality.

4.1 Append-Only Reconciliation

In the append-only case, every transaction in a given epoch can be considered independently. An insertion may be applied so long as it does not conflict with a previously applied insertion, nor does it conflict with a transaction of equal or higher priority.

For any epoch e, let $\Delta_{acc}(i)|_e$ be the set of transactions from $\Delta|_e$ acceptable to p_i. We define $\Delta_{acc}(i)|_0$ to be the empty set, and for all other epochs let

$$\Delta_{acc}(i)|_e = \{X \in \Delta|_e : (\nexists X' \in \Delta|_e : X, X' \text{ conflict and } pri_i(X') \geq pri_i(X)) \text{ and } (\nexists X'' \in \Delta_{e'} : e' < e \text{ and } X, X'' \text{ conflict})\}$$

From this, it is straightforward to define the reconciliation problem for a given participant in the append-only model.

Definition 2 (append-only reconciliation). *Let $I_i(R)|_e$ be the instance of relation R at participant p_i in epoch e. The* append-only reconciliation problem for participant p_i *is to compute $I_i(R)|_e$ for every $R \in \Sigma$, given some initial $I_i(R)|_{e_0}$ and Δ from e_0 to e. For each relation R, let $Res = I_i(R|_{e_0})$. A tuple t must appear in instance $I_i(R)|_e$ iff it appears in the instance Res resulting from recursively applying, for each τ from e_0 to $e-1$ (in increasing order), $Res = apply(\delta_R, Res)$ for every δ_R in $\Delta_{acc}(i)|_{(\tau+1)}$.*

Append-only reconciliation is very simple to compute algorithmically: during epoch τ, for each p_i, we simply consider each published transaction X in isolation and determine whether it is in $\Delta_{acc}(i)|_\tau$, meaning that it uniquely has the highest priority of any transaction with which it conflicts. If so, we apply the transaction.

4.2 Replacement and Deletion

If we allow for replacement and removal, the semantics of reconciliation must change significantly: now an update may be *dependent* on other, *antecedent* updates (where the result of the antecedent update is used by the dependent update). These antecedent updates may have originated from participants *other than* the participant who published the most recent update, and the original source of that update might not itself be trusted by the peer who is reconciling. We therefore adopt the semantics that a participant who trusts an update u must *transitively* trust any antecedent updates made by other participants, at (at least) the priority level of u, provided the transaction has not been explicitly rejected by the reconciling participant. A transaction that depends upon or conflicts with any *deferred* update, or whose antecedents do so, is deferred, until the potential conflict is resolved by explicit user interaction.

In a reconciliation model with deletions, one transaction may introduce a conflict, but a succeeding transaction may remove that conflict. For instance, to continue the example of Figure 2, suppose that in epoch 2, participant p_3 first introduced a sequence of transactions:

$$X_{3:2} : \{+F(mouse, prot2, \textit{cell-resp})\}$$
$$X_{3:3} : \{F((mouse, prot2, \textit{cell-resp}) \rightarrow (mouse, prot3, \textit{cell-resp})\}$$

where initially the wrong protein was given the function *cell-resp*. In this case, while transaction $X_{3:2}$ clearly conflicts with $X_{2:0}$, intuitively p_3 should accept $X_{2:0}$, since this does not conflict with its state after applying the full transaction sequence above. In general, given a transaction sequence, one can take the constituent update sequence and "flatten" it into a set of direct updates by removing intermediate steps, as described in [12, 14]. The sequence $[X_{3:2}, X_{3:3}]$ above can be minimized to $\{+F(mouse, prot2, cell\text{-}resp)\}$.

Let $applied(p_i, e)$ be the set of updates that have been applied by participant p_i from epoch 1 through epoch e. Also, for a transaction X published in epoch e, we define the *antecedent* set, $ante(X)$, to contain any transaction $X' \in \Delta_\tau$, $1 \leq \tau \leq e$, where X' either inserts a new tuple, or makes a modification to a tuple, which X directly deletes or modifies.

Definition 3 (transaction extension). *We define p_i's* transaction extension *of transaction X, reconciled in epoch e, to be the transitive closure of X's antecedents, so long as those transactions have not yet been accepted by p_i in epoch e:*

$$te_{i|e}(X) = \{X \cup ante(X)\}$$
$$te_{i|e}(X) = te_{i|e}(X) \cup \{X' | \exists X'' \in te_i(X) : X' \in ante(X'') \wedge X' \notin applied(p_i, e)\}$$

Henceforth we will assume that the transaction extension is sorted by the order of each transaction in Δ.

We say X *subsumes* some other X' if its transaction extension is a superset of X''s transaction extension. Given a list of transactions L, sorted by the order of their application, we can define their *update footprint* to be:

$$uf(L) = [\delta \in X \text{ for each } X \in L]$$

Now, assume we are given a function $flatten(s)$, which, given an input sequence s of updates, produces a set of mutually independent updates with all dependency chains removed, as in [14]. For a transaction X, a subset of its antecedents L, and a reconciling participant p_i, we define an *update extension* $U_i(X, L)$ to include the following components:

- an *operation*, the set of updates in $flatten(L)$.
- a *root*, which is the original transaction X.
- a *source*, which is the contents of L.
- a *priority* level equal to $pri_i(X)$.

The update extension represents the set of changes made by transaction list L as "seen" by peer p_i, with all intermediate steps removed. Our goal is to consider conflicts between update extensions, and to choose to apply the highest-priority update extensions.

To do this, we must consider when transactions' extensions conflict; here we should only consider interactions between updates that are not shared between the transactions.

Definition 4 (direct conflict). *Two transactions X, X'* directly conflict *iff $\exists\ \delta \in U(te_{i|e}(X) - S), \delta' \in U(te_{i|e}(X') - S)$ s.t. δ and δ' conflict, where $S = \{X'' : X'' \in te_{i|e}(X) \wedge X'' \in te_{i|e}(X')\}$.*

We can now define the general reconciliation problem for updates that include deletions and replacements. A solution to the general reconciliation problem must also maintain information about whether prior reconciliation operations marked certain transactions as rejected or deferred: as discussed previously, we similarly reject or defer (resp.) any future transactions that depend upon these.

Definition 5 (general reconciliation). *We define the* general reconciliation *problem for participant p_i as follows. During epoch e, given an initial $I_i(R)|_{e_0}$, a set of previously deferred transactions $deferred(p_i, e_0)$, and previously rejected transactions $rejected(p_i, e_0)$, and all Δ_i from e_0 to e, compute:*

- *A new instance $I_i(R)|_e$ for every $R \in \Sigma$, defined as follows. Let App be the set defined as follows. For each epoch τ from e_0 to e, for each X in $\Delta|_\tau$, App must contain those transactions in $te_{i|e}(X)$ that:*
  1. *have a priority $pri_i > 0$,*
  2. *can be completely applied to $I_i(R|_{e_0})$ without violating its integrity constraints,*
  3. *do not directly conflict with some other X' of equal or higher priority, which is not subsumed by X,*
  4. *do not have a transaction in $te_{i|e}(X)$ that is in $rejected(p_i, e_0)$, and*
  5. *do not delete, modify, or insert a tuple whose key matches any update in $deferred(p_i, e_0)$.*

  For each R, $I_i(R)|_e$ must contain tuple t iff t appears in Res as defined next. Initialize $Res = I_i(R)|_{e_0}$ and create an empty set Used for transactions that have been applied. For each transaction X in App that is not antecedent to any other transaction X' (i.e., it is not in $te_{i|e}(X')$ for any X'), apply all updates $Res = apply(\delta_R, Res)$ for all δ_r in $flatten(uf(te_{i|e}(X) - Used))$. Add all transactions in $te_{i|e}(X)$ to Used.
- *A new deferred set $deferred'(p_i, e)$, which adds to $deferred(p_i, e)$ every X that directly conflicts with some update in $deferred(p_i, e_0)$ or any other, non-subsumed, X' of equal priority.*
- *A new rejected set $rejected'(p_i, e)$, which adds to $rejected(p_i, e_0)$ every X that directly conflicts with some other X' of equal or higher priority, or whose extension $te_{i|e}$ contains a transaction in $rejected(p_i, e)$.*

Proposition 1. *A solution to the general reconciliation problem will always accept transactions and their antecedents for which there exist no other directly conflicting, non-subsumed transactions of equal or higher priority.*

Proof sketch. Assume for the purpose of contradiction that a transaction has not been accepted, despite being of higher priority than any transaction with which it directly conflicts, not depending an a rejected or deferred transaction, and being conformant with integrity constraints. Then the update extension of the transaction must directly conflict with a transaction accepted in the same reconciliation operation at a higher priority. But the definition only rejects or defers transactions that conflict with transactions of equal or higher priority. Hence the update extension must conflict with that of a higher-priority transaction, which is a contradiction.

A greedy algorithm that closely matches the above definition, processing items in decreasing order of priority, is provided in the next section.

Once a number of items have been deferred, the process of *conflict resolution* makes use of the solution to the reconciliation problem stated above. To resolve a conflict, the

	Client-Centric Reconciliation	Network-Centric Reconciliation
Distributed Store	Pros: No central store, medium communication Cons: Needs stable base of connected peers, reconciliation work all at one peer	Pros: No central store, distributed reconciliation work Cons: Highest communication, needs stable base of connected peers
Central Store	Pros: Low communication, high reliability Cons: Needs reliable central server, reconciliation work all at one peer	Pros: Distributes reconciliation work across many peers, high reliability Cons: High communication, needs reliable central server

Figure 3: Comparison of different combinations of reconciliation algorithms and update stores.

user specifies some number of transactions to remove from the deferred set and reject. The remaining transactions are removed from the deferred set and treated as recently published transactions, and the reconciliation solution is re-run to apply those that no longer conflict.

5. RECONCILIATION ALGORITHMS

A general reconciliation algorithm, executed by participant p_i, first determines which transactions have been published since p_i's previous reconciliation (the *relevant* transactions), and p_i's priority assignment to each of these newly published transactions. It then computes the update extensions for these transactions, and determines which updates can be applied without violating the requirements given in Definition 5. Finally it applies the transactions it has selected, records the set of transactions that it must defer, and rejects those that remain.

This process can either be centralized or distributed. If the work is centralized on the reconciling participant (peer), we call it *client-centric reconciliation*, since it is typically the reconciling participant that retrieves all of the relevant transactions and decides which to apply. An alternative is *network-centric reconciliation*, in which computation is distributed across the entire network of peers. While the network-centric approach puts less load on the reconciling participant by distributing almost all of the work across the network, the client-centric approach generates less network traffic, and it allows for a considerably simpler reconciliation algorithm. It also may allow potentially sensitive information, like the trust conditions, to be kept private from other participants.

The reconciliation algorithm needs to access several different kinds of data to perform the operations outlined above. It must access the series of published transactions operations, and the instance of the reconciling participant. It also needs to read and modify the sets of applied, rejected, and deferred transactions for the reconciling peer. We define an *update store* module to provide a general interface to much of the aforementioned state. We have explored using both a central server and a distributed store in which the peers themselves store the state.

Each combination of reconciliation algorithm and update store implementation has its own unique benefits, as shown in Figure 3. Our initial implementation uses client-centric reconciliation, which is considerably simpler both to understand and to implement; we couple that with either central or distributed storage. As future work we intend to implement network-centric reconciliation.

In order to implement an algorithm for the general reconciliation problem given in Definition 5, we introduce several new concepts:

- *Dirty values* are key values that are modified (i.e. read or written) by a deferred transaction. Any transaction that reads or writes a value whose key is in the dirty value set must be deferred, in order to ensure that a previously-deferred transaction can always be accepted later.
- *Conflict groups* are groups of conflicts with the same type that involve the same key value; the reconciliation algorithm groups conflicts for each reconciliation into such groups.
- *Options* are groups of transactions within a conflict group that make the same modification to the key value. At most one option can be accepted for each conflict group when conflicts are resolved; the transactions from the other groups are rejected.

5.1 Client-Centric Reconciliation

The core of the client-centric reconciliation algorithm is the ReconcileUpdates procedure. It determines which updates the participant can apply or reject during a particular reconciliation, and assigns the deferred transactions into conflict groups. When a participant reconciles, it first queries the update store to fetch the newly relevant transactions, their trust priorities, and their update extensions. The algorithm then determines which transactions to apply, reject, or defer; for deferrals it records conflict groups. When the user resolves one or more conflicts, this rejects the transactions in the options he or she did not select; then ReconcileUpdates is re-run for the earliest point of conflict resolution.[1] ReconcileUpdates reconsiders all previously deferred transactions, and it accepts or rejects those for which conflicts have been resolved.

The core of ReconcileUpdates is given in Figure 4, and the various helper functions appear in Figure 5. ReconcileUpdates begins by computing the flattened update extension of each trusted transaction. The call to CheckState at line 7 determines which transactions much be rejected or deferred because of the reconciling participant's dirty value set or materialized state. The call to FindConflicts at line 9 discovers conflicts between the flattened update extensions of trusted transactions. The algorithm then calls DoGroup at line 11 to consider each group of transactions with the same priority, in decreasing order of priority; the decreasing order allows the algorithm to proceed greedily and consider each group only once. Within each group, transactions that conflict with higher-priority accepted transactions are rejected, and those that conflict with higher-priority deferred transactions are themselves deferred; if conflicts are found between two non-rejected transactions within a group, both are deferred. Once all priority groups have been considered, ReconcileUpdates has made decisions for all trusted transactions. Line 13 records which transactions the client

[1]If information has been cached from the previous invocation of the algorithm, no calls to the update store should be needed; otherwise, it queries the update store in exactly the same manner as before.

```
ReconcileUpdates ($recno$)
1   $txns \leftarrow$ The IDs of the undecided fully trusted transactions
2   $prio \leftarrow$ Mapping from index in $txns$ to priority
3   $prios \leftarrow$ Set of all transaction priorities
4   Sort $prios$ in decreasing order
5   for $t \in txns$ do
6       $upEx[t] \leftarrow$ The flattened update extension of $t$
7       $decision[t] \leftarrow$ CheckState($recno, upEx[t]$)
8   endfor
9   $conflicts \leftarrow$ FindConflicts($txns, upEx$)
10  for $txnPrio \in prios$ do
11      $decision \leftarrow$ DoGroup($txnPrio, conflicts, prio, decision$)
12  endfor
13  Record $decision$ at $recno$
14  for $t \in txns$ do
15      if $decision[t] =$ ACCEPT then
16          $upEx[t] \leftarrow$ The flattened update extension for $t$
17          Apply $upEx[t]$
18      endif
19  endfor
20  $deferred \leftarrow \{txn \mid decision[txn] = \text{DEFER}\}$
21  UpdateSoftState($recno, deferred$)
```

Figure 4: The main client-centric reconciliation algorithm. Helper methods are in Figure 5.

has decided to accept or reject. Lines 14-19 update the state of the local database; it is necessary to recompute the update extension since the antecedents of the trusted transactions may overlap. Line 21 updates the client's dirty value set and list of conflicts for the current reconciliation.

Suppose that during a particular reconciliation there are t relevant transactions, each of which has at most a undecided antecedents. Further suppose that each transaction contains at most u component updates. In this case, computing the flattened update extensions will take time $O(tua)$, since that much time is needed even to read through the updates for the relevant transactions. Checking for pairwise conflicts between the update extensions will take time at most $O\left(t^2 + tua\right)$, if a hash table-based conflict detection algorithm is used. This conflict detection step asymptotically dominates all other work done afterwards by the ReconcileUpdates procedure, giving a combined running time $O\left(t^2 + tua\right)$.

By considering the trusted transactions in decreasing order by priority, ReconcileUpdates greedily ensures that the conditions given in Definition 5 are satisfied; since lower priority transactions can never affect whether higher priority transactions are accepted, the lower priority ones can be considered independently in subsequent iterations.

5.2 Update Store

The update store's fundamental role is to publish and retrieve updates, and to associate each published transaction with a client reconciliation time. The latter ensures that no transaction ends up in multiple reconciliations for the same client, and that no new updates for a specific reconciliation appear or disappear after it is recorded. As mentioned above, all other state, such as deferred transactions, conflicts, client state, trust conditions, and which transactions each participant has accepted or rejected, can remain private data to each participant.

Such a system, however, would require a great deal of communication across the network, as each update needed

```
CheckState ($recno, upEx$)
1   if $upEx$ contains a value dirty at $recno$ then
2       return DEFER
3   else if $upEx$ contains an already decided transaction then
4       return REJECT
5   else if $upEx$ is incompatible with the instance at $recno$ then
6       return REJECT
7   else if $upEx$ conflicts with the delta for $recno$ then
8       return REJECT
9   else
10      return ACCEPT
11  endif

FindConflicts ($txns, upEx$)
1   $conflicts \leftarrow \emptyset$
2   for $t, t' \in txns$ do
3       if $upEx[t]$ conflicts with $upEx[t']$ then
4           if neither $t$ nor $t'$ subsumes the other then
5               $conflicts[t] \leftarrow conflicts[t] \cup \{t'\}$
6               $conflicts[t'] \leftarrow conflicts[t'] \cup \{t\}$
7           endif
8       endif
9   endfor
10  return $conflicts$

DoGroup ($txnPrio, conflicts, prio, decision$)
1   $prioGrp \leftarrow$ Values in $prio$ that map to $txnPrio$
2   $higher \leftarrow$ Values in $prio$ that map to a priority $> txnPrio$
3   Remove rejected transactions from $prioGrp$
4   for $t \in prioGrp$ do
5       for $c \in (conflicts[t] \cap higher)$ do
6           if $decision[c] =$ ACCEPT then
7               $decision[t] \leftarrow$ REJECT ; $prioGrp \leftarrow prioGrp - \{t\}$
8           else if $decision[c] =$ DEFER then
9               $decision[t] \leftarrow$ DEFER
10          endif
11      endfor
12  endfor
13  for $t, t' \in prioGrp$ do
14      if $t$ conflicts with $t'$ then
15          $decision[t] \leftarrow$ DEFER ; $decision[t'] \leftarrow$ DEFER
16      endif
17  endfor
18  return $decision$

UpdateSoftState ($recno, deferred$)
1   Clear all soft state from reconciliation $recno$
2   for $t \in deferred$ do
3       $upEx[t] \leftarrow$ The flattened update extension of $t$
4       Remove from $upEx[t]$ clean updates inapplicable at $recno$
5       Mark $upEx[t]$ dirty at $recno$
6   endfor
7   $conflicts \leftarrow$ FindConflicts($deferred, upEx$)
8   $conflictGroups \leftarrow \emptyset$
9   for $t \in deferred,\ t' \in conflicts[t]$ do
10      for conflict $\langle type, value\rangle$ between $t$ and $t'$ do
11          Add $\{t, t'\}$ to $conflictGroups[\langle type, value\rangle]$
12      endfor
13  endfor
14  for $\langle type, value\rangle \in conflictGroups.keys$ do
15      Combine compatible txns for $\langle type, value\rangle$ into same option
16  endfor
17  Record $conflictGroups$ as conflict set for $recno$
```

Figure 5: Helper methods for the ReconcileUpdates method given in Figure 4.

during reconciliation would have to be requested individually. Our implementations, therefore, move the sets of applied and rejected transactions from the participant into the update store; this allows the update store to be determined remotely, and thereby reduces that amount of network traffic. An additional result of this approach is that each client contains only soft state; it is possible to reconstruct the entire state of the participant, up to his or her last reconciliation, from the update store.

For these reasons, we implement an update store with the following basic operations: publish transactions from a peer, and record those it has already accepted,[2] record that a peer has accepted and rejected certain transactions, record that a peer has decided to reconcile and associate with that reconciliation a particular set of published transactions, retrieve the current reconciliation number of a peer, and retrieve all of the transactions that a peer may need to see in order to perform its more recent reconciliation, along with the priorities associated with the fully trusted transactions in that set. In order to perform these operations efficiently, the update store must log all of the updates published and their epoch, what transactions each peer has accepted or rejected, the current epoch, the epoch corresponding to each peer's previous reconciliation, and the trust conditions for each peer.

Early prototypes of our system showed it was vital to reduce the number of messages sent between the update store and each participant. In the current interface, a constant number of procedures are invoked during each reconciliation. In the centralized server implementation, each of those sends a small number of messages across the network; in the distributed implementation, each trusted transaction requires a request message, though antecedent transactions will be sent automatically. The system is still limited by network bandwidth, since many transactions may need to be sent to the reconciling peer, but the reduced number of 'round-trips' between the update store and the client gives a great performance improvement in many situations.

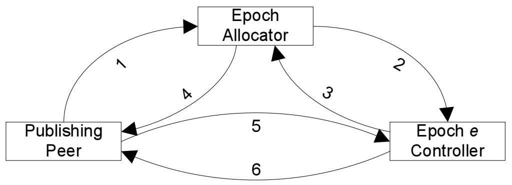
5.2.1 Relational Database Update Store

Relational database technology provides an efficient way to implement a centralized update store. Commercial RDBMSs offer high performance and durability, both important characteristics in a system such as ours. We highlight some of the more interesting and innovative aspects of our design.

In our implementation, an epoch count (implemented using an SQL sequence) is used to timestamp each batch of transactions that it is published. Since publishing is not instantaneous, each peer records when it has started publishing, and also when it has finished. We decouple publishing from reconciliation to support greater concurrency: when a peer requests to reconcile after publishing, it determines the latest epoch not preceded by an "unfinished" epoch, and it uses this as its reconciliation epoch. No additional transactions will be published by any participant prior to this point. The inputs to reconciliation, then, are any transactions whose epoch number lies between the participant's prior reconciliation epoch and this new epoch.

Implementing this approach requires care to avoid sacrificing performance. The series of epoch numbers can contain gaps if reconciliations are rolled back or aborted; therefore

[2]Some of the published transactions may be deferred because they modify dirty values.



The messages sent are request epoch (1), begin epoch e (2), confirm epoch begun (3), begin publishing at epoch e (4), publish transaction IDs for epoch e (5), and confirm epoch finished (6). After this the publishing peer can send the transactions for epoch e to their transaction controllers.

Figure 6: The procedure by which a peer publishes an epoch in the DHT-based store.

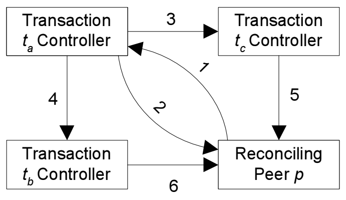
each publishing peer must record when it has finished writing all transactions to the database, as mentioned above. However, we also want to allow as many peers as possible to publish updates simultaneously. Repeatable read isolation at the DBMS level prevents race conditions: when the reconciling peer determines the epoch to associate with its reconciliation, it immediately stores that value in the reconciliations table and commits the transaction, releasing all locks. Thus it holds an exclusive lock on the epochs table just long enough to determine the largest stable epoch number; thereafter reconciliation operations are decoupled from the epochs table. By minimizing the time that lock is held, we enable maximum concurrency in publishing updates, as well as in the operation of reconciling. As long as no peer is recording its decision to reconcile, there is no limit on the number of peers that can simultaneously start reconciliation, publish updates, or record that they have finished publishing. Additionally, application of trust predicates and determination of update extensions take place inside the DBMS, meaning that only relevant transactions and transactions that contribute to their transaction extensions are sent over the network.

5.2.2 DHT-Based Store

Our distributed update store is based on FreePastry, an open-source implementation of a distributed hash table [28, 27, 31]. In this version, work (both storage and computation) is spread over the entire network of peers, using transaction identifiers and epochs as keys. The peers store three kinds of data, and each responds to several kinds of messages, which are described below in detail. In this implementation, we assume successful message delivery and postpone a study of fault-tolerance to future work.

One peer, the owner of a predesignated key, keeps track of the epoch count. When a participant wants to publish updates, it requests the next epoch count from this *epoch allocator* (see Figure 6). The epoch allocator informs the *epoch controller* for this epoch (the DHT peer who "owns" the hash value of the epoch) that this participant wants to publish updates, and then returns the epoch count to the requesting peer. After sending the epoch number to the requesting participant, the epoch allocator increments its epoch counter. We observe that, if this peer were to fail, its data could be reconstructed by polling for the largest epoch present in the system.

After a participant publishes its set of transactions during an epoch, which it sends to the peer who owns the hash of its transaction ID (the *transaction controller*), it trans-


Transaction t_a Controller
Transaction t_c Controller
Transaction t_b Controller
Reconciling Peer p
1
2
3
4
5
6

In this example p requests reconciliation information for transaction t_a. p has already applied t_b, but has not decided t_a or t_c. t_b and t_c are antecedents of t_a; t_b has other antecedents, but t_c has none. The messages sent are request t_a (1), send t_a (2), request t_c for p (3), request t_b for p (4), send t_c (5), and t_b not relevant (6).

Figure 7: An example of retrieving a transaction for reconciliation in the DHT-based store.

mits their IDs to the epoch controller. The epoch controller concludes by marking the epoch as complete.

Now the participant needs to determine an epoch for the second step, reconciliation. It requests the most recent epoch from the epoch allocator, and uses that information to request the contents of all epochs since its last reconciliation from their respective epoch controllers. It uses this information to determine the most recent "stable" epoch, and records this as its reconciliation epoch at its *peer coordinator*. Then, for each epoch since the participant's prior reconciliation, it requests the set of transactions published in that epoch from the epoch controller, and then requests that set from the transaction controllers. Each transaction controller either sends back the requested transaction, its priority, and a set of antecedents; or a notification that the transaction is untrusted or irrelevant. The reconciling peer maintains a pending transactions set, to which it adds antecedents and from which it removes received (or irrelevant) transactions. This procedure is visualized in Figure 7. When the pending transactions set becomes empty, the peer begins running the reconciliation algorithm. That algorithm notifies the appropriate transaction controllers when it accepts, rejects, or defers transactions.

6. IMPLEMENTATION & EXPERIMENTS

We have implemented the reconciliation algorithm of Section 5 above in Java, and also constructed a centralized update store, built in Java over a major commercial RDBMS, and a distributed store, based on FreePastry [28]. Since we expect ORCHESTRAsystems to consist of tens of participants, we explore configurations of up to fifty peers.

Given that no comprehensive workload already exists for bioinformatics data sharing, we developed a synthetic workload generator based on the SWISS-PROT bioinformatics database, which contains organisms, proteins, and protein functions. The simulator mimics the process of updating a curated database like SWISS-PROT: each transaction consists of a series of insertions or replacements over the Function relation, where update values are chosen according to a heavy-tailed Zipfian distribution with characteristic $s = 1.5$, sampled over the protein functions listed in SWISS-PROT. When a new key is inserted, a secondary table of database cross-references is updated to include a reference for the new key; on average, 7.3 such tuples are inserted into the secondary table for each value inserted into the primary table. All participants in the simulation trust each other at the same priority for all updates, which will result in conflicts that must be manually rather than automatically resolved.

We focus on two metrics that measure the feasibility, efficacy, and scalability of our system. The first is running time versus the amount of data and number of peers; clearly any data sharing system needs to scale well. The second metric is what we term the *state ratio*, the average number of values in all participants' states for a key (including lack of a value). This measure ranges from one (all the peers have exactly the same state) to the number of peers (there is no overlap at all between the peers' states). Since a lower ratio indicates more shared data, we consider a smaller value for this metric to generally indicate higher quality sharing.

Our experiments examine the effects of transaction size, number of participants, and reconciliation frequency on these metrics. Experiments were run at least five times, and 95% confidence intervals are given in all figures. Unless indicated, all experiments were performed with ten participants. All of the running times quoted in this paper are the averaged over all trials over all peers.

For the experiments using the central store, the RDBMS ran on a dual Xeon 3.0GHz with 2 GB of RAM running Windows Server 2003, and the client ran on a 2.8GHz Pentium 4 with 1 GB of RAM running SuSE Linux 9.2. The computers were connected via switched 100Mb Ethernet. For experiments with the distributed store, all nodes were run on the Windows server, with a delay of at least 500 microseconds added to every message (and reply) transmission. All machines used Sun's JRE 1.5.0.

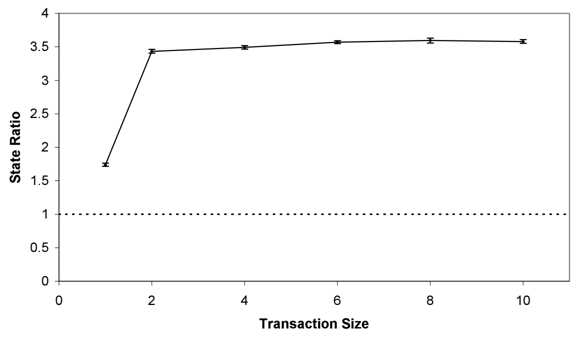


Figure 8: The effect of varying transaction size on state ratio, while holding the number of updates between reconciliations constant.

6.1 Transaction Size

Figure 8 examines the relationship between transaction size and the overall amount of data shared between peers. As one might expect, increasing transaction size increases the state ratio: larger transactions increase the number of transactions that conflict with each other, thus result in more overall (inconsistent) state within the CDSS. Surprisingly, while going from single-update transactions to transactions with two updates greatly increases the state ratio, further increases in transaction size have negligible effect, at least for our synthetic workloads. This suggests that large transactions do not cause undue fragmentation of data instances in the system.

6.2 Reconciliation Frequency

Figure 9 measures the effect on state ratio of varying the reconciliation interval. As reconciliation interval increases, i.e. we reconcile less frequently, the state ratio increases

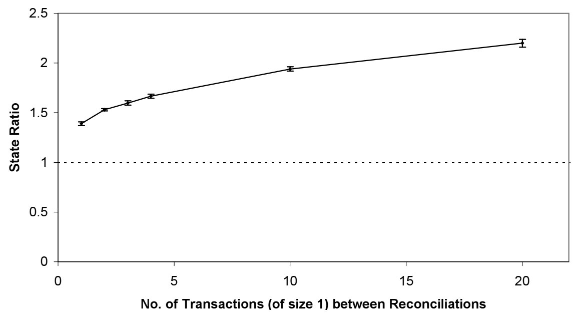


Figure 9: The effect on state ratio of varying reconciliation interval.

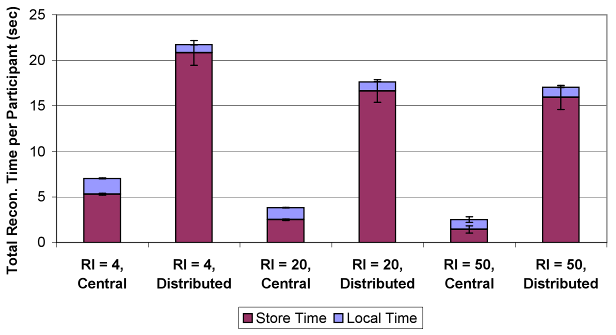


Figure 10: The effect on execution time of varying reconciliation interval, while holding transaction size at one. RI is the number of transactions published between reconciliations.

slightly; this is because more conflicts are likely to arise between longer transaction chains.

Figure 10 shows the corresponding execution times for an average participant over several reconciliation intervals from Figure 9. With a centralized update store, smaller reconciliation intervals are significantly more expensive; in contrast, for the distributed store, where the requests to follow antecedent transaction chains dominate the running time, the penalty for more smaller reconciliation intervals is negligible. Overall, we conclude that with the distributed store, unless application-level semantics require otherwise, frequent reconciliation is preferable, since it improves synchronicity at little cost. For a central store the needs of the system users will determine whether the improvement in quality is worth the cost in performance. However, more frequent reconciliations put a heavier load on overall system resources, potentially reducing performance during a large number of simultaneous reconciliations.

6.3 Number of Participants

Figures 11 and 12 show the effects of increasing the number of participants (peers), on state ratio and execution time. Increasing the number of participants has several effects. First, it increases the number of transactions that need to considered and compared with one another. Second, for the distributed store, it increases the number of peers that must be contacted to retrieve data, and therefore the number of messages that must be sent across the network. Both of these increase the time needed for reconciliation. However, even for large numbers of peers, reconciliation is an inexpensive operation, as we see in Figure 12. Finally, a large number of peers increases the maximum state ratio,

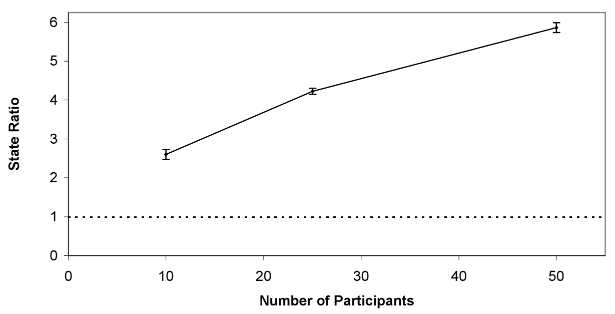


Figure 11: The change in state ratio when the number of peers is increased.

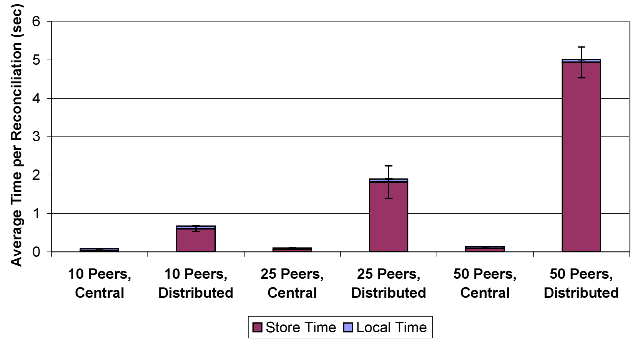


Figure 12: The effect on execution time when the number of peers is increased.

since it also increases the total number of participants who make updates; observe that the state ratio grows decidedly sublinearly, indicating a high level of sharing among even large numbers of peers.

7. CONCLUSIONS AND FUTURE WORK

This paper represents a new means of providing distributed data sharing services. Rather than requiring all sites to agree on a single data instance that is globally consistent, instead we allow each site to have its own autonomous — but carefully coordinated — data instance, with a series of *acceptance rules* specifying its policy for importing updates from others. Our contributions have been the following:

- A formalism for reconciliation, including a notion of consistency, that emphasizes local autonomy, making forward progress, and providing intuitive behavior.
- New algorithms to reconcile database instances.
- Demonstration that these algorithms perform acceptably for sharing data in mid-sized confederations.

The work discussed in this paper represents a first step in the ORCHESTRA project. In the future, we plan to address techniques for schema translation, implement a fully distributed network-centric reconciliation engine, and perform real-world evaluation of ORCHESTRA in bioinformatics data sharing. We are optimistic that our approach of supporting overlapping, yet divergent, database instances will facilitate improved data sharing practices.

Acknowledgments

The authors would like to express their gratitude to Alon Halevy, Val Tannen, TJ Green, Grigoris Karvounarakis, Olivier

Biton, the members of the Penn Database group, and the anonymous reviewers for their comments and suggestions.

8. REFERENCES

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[2] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28, 2000.

[3] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[4] D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Logical foundations of peer-to-peer data integration. In *PODS*, 2004.

[5] S. Ceri, M. A. W. Houtsma, A. M. Keller, and P. Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(3), 1995.

[6] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[7] Concurrent versions system. Available from `www.cvshome.org`.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *PODC '87*, 1987.

[9] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *UIST '97*, 1997.

[10] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. Technical Report MS-CIS-004-15, University of Pennsylvania, July 2004.

[11] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2), March 1997.

[12] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *TODS*, 21(3), 1996.

[13] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *ICDE*, March 2003.

[14] Z. Ives, N. Khandelwal, A. Kapur, and M. Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *CIDR*, January 2005.

[15] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *SIGMOD*, June 2003.

[16] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of diverging replicas. In *PODC '01*, August 2001.

[17] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 6(2), 1981.

[18] L. Lamport. Concurrent reading and writing of clocks. *ACM Trans. Comput. Syst.*, 8(4), 1990.

[19] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *KRDB '02*, April 2002.

[20] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[21] K. Moore. The Lotus Notes storage system. In *SIGMOD*, 1995.

[22] P. Mork, R. Shaker, A. Halevy, and P. Tarczy-Hornoch. PQL: A declarative query language over dynamic biological schemata. In *American Medical Informatics Association (AMIA) Symposium, 2002*, November 2002.

[23] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, 2002.

[24] National Center for Biotechnology Information. GenBank. Available from `www.ncbi.nlm.nih.gov/GenBank/`.

[25] D. S. Parker, Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. A. Edwards, S. Kiser, and C. S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3), 1983.

[26] B. C. Pierce, T. Jim, and J. Vouillon. UNISON: A portable, cross-platform file synchronizer, 1999–2001.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM '01*, 2001.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[29] F. Sadri. Aggregate operations in the information source tracking method. *Theor. Comput. Sci.*, 133(2), 1994.

[30] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4), 1990.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of ACM SIGCOMM '01*, 2001.

[32] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.