Group Anti-Entropy – Achieving Eventual Consistency in Mobile Service Environments

Klaus Herrmann
University of Stuttgart
Institute of Parallel and Distributed Systems (IPVS)
Universitätstraße 38, 70569 Stuttgart
klaus.herrmann@acm.org

Abstract

Data consistency protocols are vital ingredients of mobile data management systems. Notable research efforts have been spent to find adequate consistency models for allowing mobile and nomadic users to share mutable data. Recently, mobile Ambient Service infrastructures that pose somewhat different requirements have entered the focus of attention. Ambient services are not as loosely coupled as the afore-mentioned systems, but they still need flexible consistency protocols that may adapt to the current dynamics in the system. We propose an extension to the well-known anti-entropy protocol that makes use of the nature of Ambient Service environments to allow for a flexible consistency management among arbitrary groups of mobile service replicas. We will show that our protocol can exploit the concept of group updates to increase its efficiency in terms of bandwidth usage. Furthermore, we prove that it avoids costly state transfers by means of a simple rule that limits the divergence within the overall set of replicas. Finally, we introduce two parameters for adjusting the level of consistency in system, and we present experimental results that show the effectiveness and the efficiency of the protocol.

1 Introduction

Data consistency is a fundamental requirement in any mobile data management application. Systems like Bayou [13] and Coda [11] have been devised for supporting mobile and nomadic users in their efforts to share data with each other. These systems provide optimistic replication strategies and protocols to keep distributed mobile data stores (file systems and databases) consistent. The underlying system model is based on a very loose coupling between the participants. Mobile users basically operate in isolation and have irregular pairwise encounters during which they reconcile their data. Or they switch between an offline mode and an online mode such that any synchronization happens when they are online. The respective consistency models and protocols are optimized for this type of highly decoupled operation.

However, in recent years the idea of Ambient Intelligence AmI [2] raised new research questions that do not fit into the systems models presented above. One possible implementation of an AmI environment assumes that the physical surrounding of the user is enriched with computing resources that enable service provisioning. As the user enters a physical location, he is able to use these services to enhance his interaction with the location. One example scenario is a shopping mall that offers Ambient Services to customers, enabling them to navigate through the mall, find certain products quickly, and optimize the contents of their shopping cart according to the overall price or quality. For this purpose, we have developed the so-called *Ad hoc Service Grid* (ASG) infrastructure [5] that is based on multi-purpose computers (so-called *Service Cubes*) being distributed in such a way that they can set up an ad hoc network and cover the respective facility (e.g. the shopping mall). Mobile users can access this network via their mobile devices and use its resources. Service Cubes represent modular building blocks providing both networking and computing resources. The overall ASG can quickly be scaled to the desired size by adding, removing, or re-positioning individual Service Cubes. The basic concept that we are trying to realize with the ASG may best be characterized as *Drop and Deploy*: Apart from the physical setup of the infrastructure (the distribution of Service Cubes), all other aspects of operating the system shall be organized by the ASG software without requiring excessive manual intervention. One particular challenge in this respect is that of service distribution: Services are *injected* into the ASG at an arbitrary Service Cube and shall replicate and position themselves such that the overall network load on the ASG and the client response times are reduced. Replication is an important aspect of the ASG since it increases availability (even if the network is partitioned), distributes processing load, and reduces network load since client requests can be served by a near-by service replica instead of being routed through the whole network. However, replication also raises the problem of data consistency among the replicas of a stateful service. Strong consistency models imply a strong coupling between replicas due to the mechanisms (e.g. locking) that are necessary for implementing them. This is not desirable in a dynamic environment like the ASG. However, some degree of consistency is required in most applications.

In this paper, we propose an extension to the well-known anti-entropy consistency protocol introduced by Bayou [9]. This extension exploits the relaxed conditions in the ASG (compared to the Bayou scenario) to achieve *eventual consistency* among an arbitrary group of replicated, stateful Ambient Services. The *Group Anti-Entropy Protocol* (GAP) exhibits a natural adaptivity to the dynamics in an ASG system: When the dynamics in the system are low, the complete group of replicas can synchronize their data stores quickly and efficiently to realize a high level of consistency. When the dynamics are high (services replicate and migrate quickly), the same protocol exploits its inherent epidemic nature to achieve synchronization through a series of reconciliation processes within smaller sub-groups of replicas.

The paper is structured as follows: In Section 2, we discuss related work. Section 3 investigates the concepts for optimistic replication introduced by the Bayou mobile database system. In Section 4, we present the GAP that builds on these concepts and extends them to support Ambient Service environments. We analyze the results of our experimental investigations in Section 5 before we give our conclusions and an outlook on future work in Section 6.

2 Related Work

Optimistic replication mechanisms [10] have been researched intensively in the past. The approaches most relevant for our research are those dealing with nomadic and mobile users accessing shared data. Mobile databases like Bayou [13] and object replication systems like Deno [7] use the anti-entropy approach to achieve eventual consistency through pairwise updates. In these systems, updates of objects or database records are submitted locally and spread in an epidemic way. Bayou uses a primary copy approach to create a global order among write operations whereas Deno uses a voting mechanism to avoid having a primary and increase availability. Another domain where optimistic replication has been applied in numerous ways are mobile file systems like Coda [11] and Rumor [3]. While Coda uses a hoarding mechanism to cache files on mobile computers, Rumor replicates entire volumes and reconciles them on a pairwise basis. Even though mobile databases and mobile file systems have very similar goals, their assumptions about connectivity are different. Bayou assumes that mobile nodes are only able to connect temporarily and that the reconciliation processes during these connections may even be interrupted prematurely. Coda and Rumor, on the other hand, assume that disconnections are only temporary. Our approach lies somewhere in the middle: Distributed services may be able to connect to each other for extended periods of time while, at other times, extended disconnections occur due to topology changes in the underlying network. Therefore, we take a database approach that is tailored to this requirement through an extension of the basic Bayou approach exploiting the somewhat relaxed conditions.

Most approaches to the dynamic placement of services within a network [8, 1] do not consider data replication and consistency as a major problem. Moreover, these systems are not targeted at mobile networks.

3 Optimistic Replication – The Bayou Model

Bayou [13, 9] is a database system for weakly connected mobile users. Each user has a replica of the database on his mobile device, and he may execute write operations (also called *Writes* hereafter) on this replica, even while he is not able to connect to other replicas in order to synchronize. These Writes are tentatively applied to the data store, and they are stored in a write log. If two replicas meet, they mutually exchange their Writes and apply them to their respective data stores. For this purpose, tentative Writes are rolled back (their effects on the data store are being undone), newly received Writes are merged into the existing write log according to their timestamps, and the operations in the new write log are then applied to the data store again. In this way, new Writes are propagated via a series of pairwise reconciliation processes, such that they spread through the system similar to the way in which an epidemic spreads. It can be shown that if no new Writes are created for some time and if no permanent partitions occur, then all databases will eventually have the same Writes in the same order in their logs. Consequently, the resulting data stores are also consistent. This model is called *eventual consistency*.

In order to eventually create the same Write order, each Write is given a timestamp consisting of a logical clock value (Lamport clocks) and the unique ID of the replica on which it was originally created. For example, the Write with timestamp 7.3 was received by the replica with ID 3 at (logical) time 7. Each replica r_i maintains a *version vector* $r_i.v$ that stores the timestamp of the most recent Write that r_i holds from r_j at index j ($r_i.v(j)$). Each time two replicas meet, the sending replica (S) fetches the receiving replica's (R's) version vector. By comparing R's version vector with its own, S can find all Writes that R still needs. It sends these Writes to R, and subsequently the two replicas switch roles such that R can send its Writes to S. This algorithm is called *anti-entropy* since it constantly decreases the *disorder* among the different data stores.

Indefinite write log growth is avoided by electing a primary replica that has the exclusive right to commit Writes when it receives them. When a Write is committed, it is given a globally unique *Commit Sequence Number* (CSN) that becomes part of the timestamp. Committed Writes (also called *stable Writes*) are also propagated using anti-entropy. However, they are applied permanently to any data store. Thus, a committed Write can never be rolled back and can be removed from the write log.

4 Group Anti-Entropy

Compared to the original Bayou model, the ASG environment allows us to relax some of the constraints for a consistency protocol. Service replicas may contact each other much more frequently. Each replica has the possibility to contact at least a subset of its current fellow replicas most of the time. Services may be disconnected by temporary network partitions, and the dynamics inherent to the creation and the removal of replicas may result in a specific replica having incomplete information about its current fellows. Thus, instead of having to deal with a set of isolated nodes that have irregular, pairwise encounters, we can assume that the contact between the replicas is much tighter.

The Group Anti-Entropy Protocol (GAP) makes use of these relaxed conditions by assuming that each replica knows a subset of all replicas that currently exist, albeit, this knowledge may be incomplete. The GAP deals with this circumstance by running independent reconciliation processes among groups of replicas. Each replica has a *view* of the current replica group. If a replica chooses to reconcile then it does so with the group of the replicas in its current (possibly incomplete) view. In the ASG, a replica acquires this view by querying a distributed Lookup Service [6] that maintains approximate information about the current set of service replicas. Like the original anti-entropy protocol, the GAP achieves eventual consistency by exploiting the epidemic nature of successive reconciliation processes among several subsets of replicas.

This protocol has a nice property: If the dynamics and disturbances in the systems are low, a single reconciliation process is sufficient to bring all replicas of a service up-to-date since all replicas have a complete view. If, however, the dynamics is high and views tend to be rather incomplete, the protocol still achieves synchronization, albeit, this may take more reconciliation actions among smaller subsets of replicas. Additionally, the synchronization may not be as *tight* since the dynamics avoids complete synchronization among all replicas at any point in time. Thus, the consistency among the overall group of replicas may degrade as the dynamics increases, but it does not break down. Furthermore, the fact that a whole group takes part in a reconciliation is exploited to reduce the number of bytes transmitted between the replicas and to speed up the reconciliation.

The dynamics (degree of mobility) displayed by users does not influence the reconciliation process presented here since it only affects which replica a client submits its operations to. The general necessity of propagating each operation among all replicas remains the same. The effects of the users' mobility on their access to data can be mitigated through the session guarantees introduced for the original bayou system [12]. The implementation of these guarantees is orthogonal to the process described here.

4.1 The Protocol

Like in the original Bayou protocol, each replica r_i holds a version vector covering all replicas known to r_i. An example of a version vector for a replica r_1 in an overall group of four replicas could be

$$r_1.v = (5.1, 5.2, 7.3, 11.4)\,. \tag{1}$$

The GAP is started by one replica that is called *active* hereafter. All remaining replicas in its view are *passive*. In the first phase, the active replica requests all passive replicas to send their current version vectors. For demonstration purposes, let us assume that r_1 is active, that it has three other replicas (r_2, r_3, and r_4) in its view, and that the version vectors are the following:

$$\begin{aligned} r_1.v &= (\ \underline{5.1},\ \ 5.2,\ \ 7.3, 11.4\ ) \\ r_2.v &= (\ 3.1, \underline{11.2}, 10.3, 15.4\ ) \\ r_3.v &= (\ 3.1,\ \ 8.2, \underline{17.3}, 15.4\ ) \\ r_4.v &= (\ 3.1,\ \ 8.2, 10.3, \underline{15.4}\ ) \end{aligned} \tag{2}$$

After having collected all vectors, the active replica r_1 tries to retrieve all Writes that it does not already hold from the passive replicas. It tries to minimize the number of messages needed to reconcile by comparing the vectors and by calculating the best order in which to synchronize with the group of replicas. To do this, r_1 applies a greedy strategy by selecting the passive replica that promises to produce the best *update progress*. Note that, unlike Bayou, the ASG does not synchronize its logical clocks with the system clock in any way. The reason for this is that unsynchronized clocks allow an educated guess about the number of events that happened between two timestamps since a logical clock is only advanced if some event occurs. Thus, based on the timestamps in the version vectors, r_i can calculate a *preference value* $pref_{ik}$ for every replica r_k in its view. This value is an indication of how much progress r_i could make by reconciliating with r_k. To calculate the preference value,

r_i first compares its version vector with the vectors of the other replicas in its view. It does so component-wise and calculates the distance $d_{ik}(j)$ between the version vectors of replicas r_i and r_k for every vector component j. The result is the distance vector d_{ik} of r_i with respect to r_k:

$$d_{ik}(j) = \begin{cases} \max(0, r_k.v(j) - r_i.v(j)) : \exists_{kj} \wedge \exists_{ij} \\ r_k.v(j) : \exists_{kj} \wedge \nexists_{ij} \\ 0 : \nexists_{kj} \wedge \exists_{ij} \end{cases} \tag{3}$$

The difference between two timestamps is defined as the difference of their logical clock components. The predicate $\exists_{ij}$ is true iff the version vector of r_i has a component j, i.e., an entry for replica r_j. If a component j exists in both vectors, then both replicas have operations originating from r_j. In this case, we simple subtract the two timestamps. Note that if r_i has a more recent Write from r_j than r_k, the difference is negative. Thus, r_i cannot benefit from a reconciliation with r_k with respect to the Writes produces by r_j. Therefore, we set the distance to zero in this case. If component j only exists in $r_k.v$, then r_i has never received any operation originating from r_j. Thus, the distance between the two timestamps is equal to the timestamp of $r_k.v(j)$. Finally, if component j only exists in $r_i.v$, then r_k does not provide any operations originating from r_j, and r_i would not gain anything from reconciliating with r_k, concerning operations from r_j.

For every passive replica r_k, r_i sums up all distance vector components $d_{ik}(j)$ and gets the total preference $pref_{ik}$ that r_i has for reconciliating with r_k:

$$pref_{ik} = \sum_j d_{ik}(j) \tag{4}$$

Finally, r_i selects the replica r_l with the highest preference value because a reconciliation with this replica has the highest potential benefit. For the reconciliation, r_i sends its current version vector to r_l, and r_l sends all Writes that are missing on r_i. Then, r_i updates its version vector to reflect the newly received operations and repeats the procedure until $\forall k : pref_{ik} = 0$. That is, r_i has every Write existing on the replicas in its view. Now, r_i merges these operations into its data store (roll-back, merge, and roll-forward). In the final phase of the protocol, r_i sends all Writes missing on the other replicas in its view to the respective replicas who in turn merge them with their data stores. Now, all replicas in the group are perfectly synchronized.

For the setup in our example, the replicas' initial version vectors are given in Equation 2. Calculating the difference vector d_{1k} for every k results in:

$$\begin{aligned} d_{12} &= (\ 0, 6,\ \ 3, 4\ ) \\ d_{13} &= (\ 0, 3, 10, 4\ ) \\ d_{14} &= (\ 0, 3,\ \ 3, 4\ ) \end{aligned} \tag{5}$$

Since r_1 always has the most recent operations originating from itself, the first component of each preference vector is 0. If r_1 would reconcile with r_2, it could potentially receive 6 $(11 - 5)$ new operations submitted at r_2, 3 operations submitted at r_3, and 4 submitted at r_4. Note that it is not guaranteed that r_1 makes this progress in terms of the number of new operations. This is due to the way in which the logical clocks are incremented: A replica does not only increment its clock if a new operation is submitted, but also when it communicates with other replicas.

In our example, the preference values are the following:

$$\begin{aligned} pref_{12} &= 13 \\ pref_{13} &= 17 \\ pref_{14} &= 10 \end{aligned} \tag{6}$$

Thus, r_1 choses r_3 first and requests an update. Note that r_3's version vector completely covers that of r_4: For every component, r_3 provides at least the operations held by r_4. Thus, after updating its write log with the operations received from r_3, there is no need for r_1 to request an update from r_4 anymore. If r_a covers r_b, then $pref_{ia} \geq pref_{ib}$ and r_a is ranked before r_b. Therefore, covering replicas are always chosen before covered ones and, thus, no updates are requested from covered replicas at all. After the update from r_3, the situation is as follows:

$$\begin{aligned} r_1.v &= (\ \underline{5.1},\ \ 8.2, 17.3, 15.4\ ) \\ r_2.v &= (\ 3.1, \underline{11.2}, 10.3, 15.4\ ) \\ r_3.v &= (\ 3.1,\ \ 8.2, \underline{17.3}, 15.4\ ) \\ r_4.v &= (\ 3.1,\ \ 8.2, 10.3, \underline{15.4}\ ) \end{aligned} \tag{7}$$

$$\begin{aligned} p_{12} &= (\ 0, 3, 0, 0\ ) &\Rightarrow\ & pref_{12} = 3 \\ p_{13} &= (\ 0, 0, 0, 0\ ) &\Rightarrow\ & pref_{13} = 0 \\ p_{14} &= (\ 0, 0, 0, 0\ ) &\Rightarrow\ & pref_{14} = 0 \end{aligned} \tag{8}$$

Now, r_2 is the only replica with a preference value greater than 0. r_2 potentially still holds 3 operations (submitted locally at r_2) that r_1 has not seen. After receiving an update from r_2, r_1 is consistent with the union of all replicas in its view. At this point, r_1 calculates the differences between its own log and that of the other replicas and updates each of them accordingly. Afterwards, all replicas hold the same set of operations. Thus, they are all consistent with each other.

4.2 Stable Writes and Log Truncation

Like Bayou, the ASG consistency protocol employs the principle of *stable Writes* to be able to truncate write logs. A replica may decide to remove any stable Write from its log. This raises the problem that the sender may have removed committed Writes that the receiver has not seen yet due to extended periods of disconnection. Thus, there is no way to reconcile by exchanging Writes since some of them are missing. In Bayou, the solution for this problem is a complete transfer of the data store's state up to the first omitted Write after having rolled back to that position in the log. In the ASG, we try to avoid this situation because complete state transfers can be very costly. Before we give a detailed discussion of the mechanism employed to avoid complete state transfers, we define some new notations.

Each replica stores the most recent commit sequence number it has seen. This the point up to which the replica's data store is stable. CSN_i^j denotes the CSN of replica r_j as it is currently perceived by replica r_i. Note that this value can be outdated due to the fact that r_i may not have had contact with r_j for some time. Thus, $CSN_i^j \leq CSN_j$. CSN_i denotes the actual current CSN of r_i. CSN_i^{min} is the minimal value of all CSNs currently perceived by r_i: $CSN_i^{min} = min_{1 \leq j \leq n} \left(CSN_i^j \right)$. CSN^{min} is the actual, current, minimal value over all CSNs. CSN_i^{max} and CSN^{max} are defined analogously. To denote the value of any of these parameters at some time index t, we write $CSN_i^j(t)$, $CSN_i(t)$, $CSN_i^{min}(t)$ etc. respectively. OSN_i denotes the *Omit Sequence Number* of replica r_i. The OSN is only known to the replica itself and indicates the CSN of the most recent omitted write operation.

Let $\{r_1, \ldots, r_n\}$ be the group of replicas that take part in a reconciliation. Let S be the set of the Commit Sequence Numbers of the Writes that are sufficient in order to do a reconciliation without state transfer:

$$S = \left\{ s \mid CSN^{max} \geq s > CSN^{min} \right\} \tag{9}$$

To avoid state transfers during a reconciliation among the replicas $\{r_1, \ldots, r_n\}$, the following property must hold:

$$\forall s \in S : \exists i \in \{1, \ldots, n\} : CSN_i \geq s > OSN_i \tag{10}$$

In other words, for every required Write (all CSNs $s \in S$), there has to be some replica in the group that still holds the operation. We call this property the *Bounded Divergence (BD) Property*. Figure 1 depicts an example of a reconciliation group of 4 replicas that fulfills this requirement. However, if r_4 was not present in the group, then the Write with the commit sequence number 7 would not be present on any replica. r_1 and r_3 have not yet received this Write, and r_2 already removed it to save space. Thus, r_2 would have to transfer its state at least up to the result of the execution of Write number 7 to both r_1 and r_3.

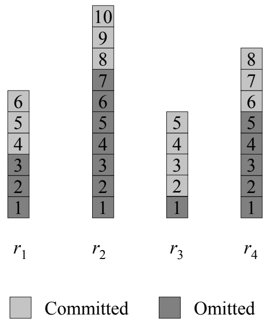


Figure 1. Group of four replicas with different write log states.

4.3 Bounded Divergence

In the following, we describe an extended version of the GAP that exploits the relaxed conditions in the ASG to avoid state transfers completely. We call this protocol the *Bounded Divergence Group Anti-Entropy Protocol* (BD-GAP). We introduce an additional timestamp vector which is called the *CSN vector*. Each replica r_i maintains such a vector that holds the values CSN_i^j. Whenever i and j take part in the same reconciliation process, they mutually exchange their CSN vectors after all the write logs in the group have been synchronized. That is, if r_i takes part in a reconciliation with a total of n replicas, then r_i receives all $n-1$ CSN vectors of the other participants. Subsequently, r_i updates its own CSN vector as follows:

$$CSN_i^j := max_{1 \leq k \leq n} \left(CSN_k^j \right) \tag{11}$$

Thus, CSN_i^j now holds the highest CSN for replica r_j that was perceived by any of the participants in the current reconciliation process. This represents the best guess of r_j's current CSN. Note that r_j itself may not be participating in the current reconciliation. Therefore, this guess may be based on a CSN received by one of the participants in an earlier reconciliation.

The goal is to limit the divergence of the write logs such that state transfers do not happen. The basic idea is to set a limit to the number of Writes that a replica may omit, such that property (10) is never violated. To achieve this, we introduce a simple rule for the removal (omission) of Writes from the write log:

Definition 4.1 (Omission Rule)**.** *r_i must never omit a committed Write that has a CSN higher than CSN_i^{min}.*

This means, we take the smallest known CSN among all known replicas (not only the participating ones!) and set it as the *omit limit*. Thus, we only omit Writes that have been received by all known replicas. Before we state the two central theorems that express the validity of this approach, we give the fundamental assumptions that must be fulfilled:

Definition 4.2 (Assumptions)**.**

1. *All replicas of a given service stem from a single, initial replica either directly or indirectly.*

2. *A replication at time t always produces a spawned replica r_s with a CSN equal to the CSN of the parent replica r_p: $CSN_s(t) = CSN_p(t)$.*

3. *The CSN of a replica never decreases as time progresses: $\forall t_0, t_1 | t_0 < t_1 : CSN_i(t_0) \leq CSN_i(t_1)$.*

4. *There is a single primary replica with the exclusive right to commit Writes.*

Theorem 4.1. *The Omission Rule guarantees that the BD Property (Equation 10) holds even if partitions occur.*

For spatial restrictions we can only sketch the proof here. The full proof can be found in [4].

Proof (sketch). Let $g(t)$ be the time-dependent group of all existing replicas, and let $g_i(t)$ be the subgroup known to replica r_i. Then $\overline{g_i(t)} = g(t) \setminus g_i(t)$ is the set of replicas that r_i has no knowledge of. In the following, we omit t for brevity, and we write, for example, g instead of $g(t)$. If

$$\forall r_i \in g : CSN_i^{min} \leq \min_{r_j \in g} CSN_j \qquad (12)$$

then the BD Property holds. Proving that (12) holds for $r_j \in g_i$ if the Omission Rule is obeyed is straight forward since $CSN_i^{min} = \min_{r_j \in g_i} CSN_i^j$ and $CSN_i^j \leq CSN_j$. For $r_o \in \overline{g_i}$, we must show that

$$\forall r_o \in \overline{g_i} : CSN_o \geq CSN_i^{min}. \qquad (13)$$

Due to our assumptions, r_o must have been replicated indirectly by r_i, or r_i and r_o must have a common ancestor $r_w \in g_i$. In both cases, we can show that (13) holds since any replication is also a reconciliation. Thus, (13) is a consequence of the assumptions stated in Definition 4.2. □

Theorem 4.2. *If partitions among the overall set of existing replicas are only temporary, then the* Bounded Divergence GAP *does not let write logs grow indefinitely large.*

The basic idea of the proof is as follows: We assume that a replica exists that can never remove any Write from its log and show that this can only happen if permanent partitions occur. This is the reverse implication of Theorem 4.2.

Proof. We assume that a replica r_i exists that can never remove any Write from its log due to the Omission Rule. This must be a result of r_i's CSN^{min} value never being increased. For CSN^{min} to remain at the same value, r_i's CSN vector must be invariant in at least one position. That is, one replica r_k must exist whose element CSN_i^k in r_i's vector never increases. This can be directly concluded from the Omission Rule. We assume that new Writes are constantly being submitted somewhere in the system. Otherwise, indefinite growth could never occur. If r_i's value for r_k's CSN never increases, then either r_k never receives any new committed Write, or it is unable to propagate its new CSN to r_i. In the first case, r_k must be permanently separated from the primary replica. In the latter case, r_i must be permanently separated from r_k. Thus, in both cases, a permanent partition must exist. □

4.4 Triggering Reconciliations

In the GAP, replicas decide autonomously when to start an active reconciliation process. This is done based on two simple parameters. A replica starts an active process

- if the number of Writes received from clients since the last reconciliation exceeds the *write threshold* W_{rec}, or

- if the time that has passed since the last reconciliation exceeds the *reconcile timeout* T_{rec}, and at least one Write has been received locally during that time.

These two predicates are called *reconciliation rules*. The first rule sets a limit to the level of inconsistency that may occur within a group of replicas. If N is the number of replica in the group, then they may at most reach the inconsistency level equal to having $N \cdot W_{rec}$ different Writes in the group. This upper limit is reached if all replicas in the group receive an equal amount of W_{rec} Writes in the time period T_{rec}. However, it is more likely that they experience different load conditions. Thus, a new reconciliation process will be started by one replica that reaches the limit first while other replicas may have less Writes.

The second rule ensures that even if it takes very long for a replica to reach the write threshold, reconciliation processes will take place in regular intervals as long as Writes are received at all. If this rule was not applied, there could be long-lasting periods of growing inconsistency.

5 Experimental Results

5.1 Setup

We have implemented a simulation of the complete ASG. To produce a benchmark scenario for our consistency protocol, we modeled the data stores of the replicas as bit strings of ℓ bits each ($\ell = 100$ in the following). Each Write toggles one of the bits in the data store associated with the respective replica. A w-fraction of all operations produced by clients are Writes. w is called the *write ratio*.

This artificial application (setting bits in a bit string) enables us to come up with a simple measure for the consistency among an arbitrary number of replicas. For this purpose, we introduce a global data store represented by the bit string B_G. This data store is the result of all operations from all clients being executed in the correct order on a single bit string. In reality, this string would not exist, but in the simulation, we can easily construct it. Let $|B|$ denote the number of 1-bits in the bit string B. Our consistency measure κ is defined as follows:

$$\kappa = 1 - \frac{\sum_{r \in R} |B_r \oplus B_G|}{|R| \cdot \ell} \qquad (14)$$

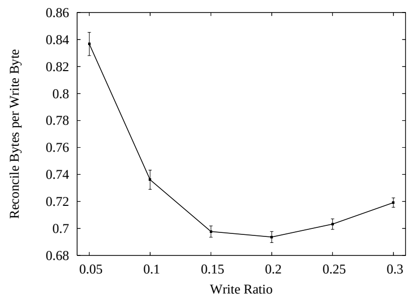
The numerator denotes the number of differences between B_G and all the bit strings B_r found in the set of existing replicas R. The denominator is the number of bits existing in all data stores. The fraction measures the inconsistency found in the system. To find the consistency, we subtract it from 1. κ is 1 if all replicas hold bit strings identical to B_G, and it is 0 if each B_r is the negation of B_G. One can easily verify that, if the individual bit strings are independent, the expected value of κ is 0.5. Thus, if κ is close to 0.5, this indicates that no correlation exists among the data stores. From a reasonable consistency protocol, we would expect values much high than 0.5. An ideal protocol would constantly maintain κ at 1.

We denote the time-dependent consistency function as $\kappa(t)$. $\kappa(t)$ is simply computed from the bit strings and the value of $|R|$ at time t. The average consistency $\overline{\kappa}$ for a complete simulation run is defined as follows:

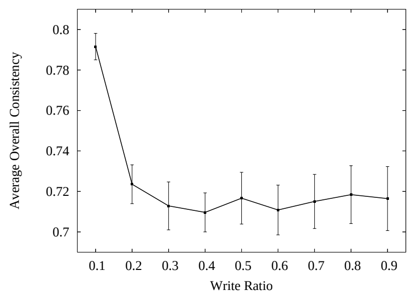
$$\overline{\kappa} = \frac{\sum_{t \in [0, T-1]} \kappa(t)}{T} \qquad (15)$$

5.2 Results

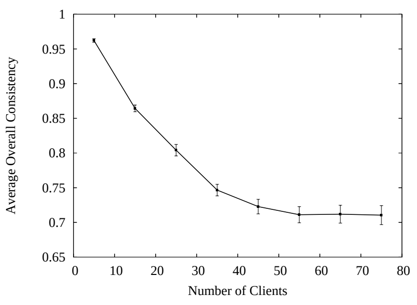
In the following figures, vertical bars indicate the 95% confidence intervals of the measurements. Each measurement represents the average of 100 experiments. Figure 2(a) depicts how many bytes have to be transmitted during reconciliations for one byte of a write operation. We denote this ratio as w_s. This shows how efficient the GAP works for different write rates w. As the write ratio w is increased, w_s drops until it reaches a minimum at about $w = 0.2$. The decrease is caused by the fact that the GAP relies on collecting a number of Writes and transmitting them together in one message in a specific order that minimizes the traffic. For a low write ratio, reconciliations will mostly be triggered by the timeout T_{rec}. Therefore, the overhead is higher since more messages are needed to transmit the same number of Write operations. As w increases beyond 0.2, transmissions tend to be triggered by the write threshold. Thus, the time interval between reconciliations gets shorter. The consistency protocol takes an optimistic approach concerning conflicts that may occur between concurrent reconciliation attempts. At high write ratios, conflicts occur more often and lead to messages being dropped. Therefore, w_s increases for values higher than $w = 0.2$.



(a) Protocol efficiency against w



(b) $\overline{\kappa}$ plotted against the write ratio w



(c) $\overline{\kappa}$ for increasing number of clients

Figure 2. Experimental results.

In Figure 2(b), the average consistency $\overline{\kappa}$ is depicted for increasing write ratio w. As expected, an increasing write ratio reduces the consistency. However, it decreases only by about 10% from the maximal value at $w = 0.1$ to the minimal value at $w = 0.4$. Moreover, after an initial steep drop, the consistency stabelizes around a value of about 0.71. This behavior is the result of the combination of the *reconcile write threshold* W_{rec} and the *reconcile timeout* T_{rec}. Initially, (approximately for $w < 0.3$) the number of Writes received by a replica on average is not high enough to trigger the write threshold rule. Reconciliations are always triggered by the timeout rule. Therefore, an increasing write ratio causes more inconsistency in this phase as the time intervals between consecutive reconciliations are approximately equal. However, as w is increased beyond a certain value, the intervals start getting shorter as the write threshold rule takes over: After a certain number of Writes is received, a replica starts a reconciliation process, and for large w this point is reached before the timeout expires. Doing a reconciliation after a prescribed number of Writes have modified the data store limits the inconsistency experienced throughout the whole system. This keeps the system at a constant consistency level.

Figure 2(c) shows how $\overline{\kappa}$ develops as the number of clients in the system increases. In this experiment, w was kept constant at a value of 0.1. The same general behavior as for a growing write ratio can also be observed here: $\overline{\kappa}$ drops until it converges to a value of about 0.71. The reason is that a growing number of clients produce a growing number of write operations. Therefore, the timeout rule fires in the initial stages (for a low number of clients), and the write threshold takes over for a higher number of clients.

6 Conclusions and Future Work

In this paper we have introduced an extension to the well-known anti-entropy consistency protocol for a more tightly coupled Ambient Service environment. The *Bounded Divergence Group Anti-entropy Protocol* (BD-GAP) runs reconciliation processes among an arbitrary group of service replicas and exploits this fact to increase its efficiency in terms of network bandwidth. Due to this group reconciliation concept, the protocol can adapt to different degrees of dynamics. If the dynamics is low, a high degree of consistency can be achieved efficiently. In highly dynamic situations, the reconciliation groups tend to get smaller, and the epidemic nature of the original Bayou protocol comes into play. This leads to a graceful degradation of the overall consistency level without breaking down the system. We have shown that costly state transfers can be avoided completely by introducing the simple *Omission Rule* that limits the divergence of the individual write logs. Finally, our experimental results show that the interaction between the two tunable parameters of the system also limits the decrease in consistency suffered as the load (write ratio and client number) on the system increases.

If the vision of Ambient Intelligence shall be realized and intelligent environments shall interact with the user in a decentralized fashion, then flexible data consistency mechanisms are a key requirement. The adaptiveness of our approach is a clear advantage over existing technologies as it enables a wide range of working conditions. The class of applications that may be run in the ASG is a superset of those targeted by Bayou. However, eventual consistency still limits this class such that *critical tasks* (e.g. classical financial transactions) are prohibitive.

In our future work, we concentrate on designing a proper model for shaping the consistency characteristics by tuning W_{rec} and T_{rec}. The goal in this respect is to allow a purposeful tuning to adapt the system to different classes of services. Furthermore, the performance of the system in more realistic applications shall be investigated.

References

[1] S. Choi and Y. Shavitt. Placing Servers for Session-Oriented Services. Technical Report WUCS-2001-41, Department of Computer Science, Washington University, 2001.

[2] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman. Scenarios for Ambient Intelligence in 2010. Technical Report, The IST Advisory Group (ISTAG), 2001.

[3] R. G. Guy, P. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, Nov. 1998.

[4] K. Herrmann. *Self-Organizing Infrastructures for Ambient Services*. PhD thesis, Berlin University of Technology, July 2006.

[5] K. Herrmann, K. Geihs, and G. Mühl. Ad hoc Service Grid – A Self-Organizing Infrastructure for Mobile Commerce. In *Proceedings of the IFIP TC8 Working Conference on Mobile Information Systems (MOBIS 2004)*. IFIP – International Federation for Information Processing, Springer-Verlag, Sept. 2004.

[6] K. Herrmann, G. Mühl, and M. A. Jaeger. A Self-Organizing Lookup Service for Dynamic Ambient Services. In *25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 707–716, Piscataway, NJ, USA, June 2005. IEEE Computer Society Press.

[7] P. J. Keleher and U. Cetintemel. Consistency management in Deno. *Mob. Netw. Appl.*, 5(4):299–309, 2000.

[8] K. Liu, J. Lui, and Z.-L. Zhang. On Service Replication Strategy for Service Overlay Networks. In *Proceedings of the Network Operations and Management Symposium (NOMS 2004)*, pages 643–656, Apr. 2004.

[9] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, New York, NY, USA, 1997. ACM Press.

[10] Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[11] M. Satyanarayanan. Coda: A Highly Available File System for a Distributed Workstation Environment. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, Sept. 1989.

[12] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 140–150. IEEE Computer Society Press, 1994.

[13] 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, New York, NY, USA, 1995. ACM Press.