

Reconsidering Consistency Management in Shared Data Spaces for Emergency and Rescue Applications

Thomas Plagemann, Ellen Munthe-Kaas, Vera Goebel

Department of Informatics, University of Oslo
P.O.Box 1080, Blindern, N-0316 Oslo, Norway
{plageman, ellemk, goebel}@ifi.uio.no

Abstract: Efficient information sharing is very important for emergency and rescue operations. These operations often have to be performed in environments where no communication infrastructure exists. Therefore, Mobile Ad-Hoc Network (MANET) technologies need to be leveraged for generic information sharing services. However, MANETs are using the shared medium of wireless networks, i.e., IEEE 802.11, which means that bandwidth is limited and mobility might cause network partitions. In order to achieve high availability of important information and increase overall system performance, replication can be used. The drawback of aggressive replication is the problem of inconsistencies and the costs of possible conflict resolution strategies. Since emergency and rescue operations are only operating for a short time compared to classical database applications, storage space might not be considered a bottleneck even for small devices: Instead of deleting and updating data, data versions are created, and each application can influence the choice of consistency model for its data and use its own policy to resolve conflicts. Such an approach does not only allow for application specific conflict resolution, but also enables traceability of what happened and has been registered, in the aftermath of the emergency and rescue operation.

1 Introduction and Motivation

Organizations involved in rescue operations of recent disasters have recognized that “*the flow of information throughout the disaster cycle is crucial for effective humanitarian operations*” [RC05]. It is obvious that data networks can help to increase the efficiency of the collaborative work of the rescue personnel. Communication networks for this application domain need fast deployment in the affected area and must not make any assumptions about the available infrastructure in the area. Therefore, mobile ad-hoc networks (MANETs) that are formed by the wireless devices brought into the area and carried by the rescue personnel, are the most promising approach for this application domain. It is the aim of the Midas project [G06] and the Ad-Hoc InfoWare project [PA+04] to develop middleware solutions that can provide applications with a virtual shared data space in such environments. However, MANETs are using the shared medium of wireless networks, i.e., IEEE 802.11, which means that (1) bandwidth is limited, and (2) the layout of the affected area, physical obstacles in the area, and the mobility of the network nodes might lead to frequent and/or long term network partitions. We call this kind of networks Sparse MANETs, which can be seen as a combination of MANETs and Delay Tolerant Networks. To enable applications to

continue their work also in the advent of longer term network partitions, shared data must be replicated in the data space to increase its availability. Well-designed replication has also the advantage of improving the overall system performance. While the two research projects Midas and Ad-Hoc InfoWare pursue different approaches to implementing the data space, they both are confronted with the same fundamental problem: consistency management of data in Sparse MANETs. Neither classical transaction-based solutions nor server-based pessimistic consistency management can be applied, because of possible communication loss and network partitions. Applications in different network partitions must be allowed to independently read and write their local replica, otherwise partitioning might halt all applications, which is not acceptable for emergency and rescue systems. When two network partitions merge, the middleware has to detect inconsistencies and resolve them. The most intuitive solution would be to regard always the newest value as the correct one. However, such an approach requires a global clock, which is not feasible, because clock synchronization between all mobile devices can never be perfect and GPS cannot be assumed available for all devices. It is the aim of this paper to discuss an entirely different approach for consistency management in Sparse MANETs for emergency and rescue applications.

One important insight for our approach is the fact that emergency and rescue operations are only operating for a short time compared to classical database applications, i.e., some hours up to a few days. During this short time, new data will be created and stored in the shared data space, but storage space does not represent a bottleneck for this data even on small devices. Disk-less mobile devices can already today use memory cards with several Gigabytes capacity, and devices with hard disks have several hundreds Gigabytes of storage space available. Therefore, we propose to never delete and update data, but create data versions instead. By this, each application can use its own policy to resolve a conflict. Such an approach does not only allow for application-specific conflict resolution, but also enables traceability of what happened and has been registered, in the aftermath of the emergency and rescue operation.

The rest of the paper is structured as follows: We give in Section 2 a description of emergency and rescue applications and their requirements. In Section 3, we discuss in detail the problem of consistency management in Sparse MANETs and give an overview of the state-of-the-art. Finally, our approach is presented in Section 4. We want to emphasize that it is not the goal to present design and implementation of a shared data space, but only to discuss a new and unconventional solution for consistency management. Conclusions are given in Section 5.

2 Emergency and Rescue Applications

In emergency and rescue operations, rescue teams with personnel from different organizations, like policemen, firemen, physicians, and paramedics, typically arrive independently at the scene. The rescue personnel cooperate to rescue human lives and to reduce any harm to humans, the environment, and any other kind of destruction in the area. Since cooperation requires communication, much time is spent on this. Thus, any tool that reduces the time that is needed for communication, increases the efficiency of

the rescue personnel. We assume that wireless computing devices will be used as the basic technical means for information sharing between rescue personnel. It is not possible to make any assumptions about the available networking infrastructure at the scene. This is valid for wired networks and for wireless networks like 3G networks, because the infrastructure might have been destroyed, including 3G base stations; or it is not accessible, like in tunnels, etc. Therefore, the mobile devices carried by the rescue personnel at emergency sites, form Sparse MANETs with all their well-known properties, like heterogeneous nodes, unpredictable reachability of nodes, network partitions, etc. A shared data space that is tailored to operate on top of Sparse MANETs is supposed to facilitate the information sharing. Applications for rescue personnel that leverage the shared data space include command and control applications, as they are implemented in the Midas project [KG+06]. Here, management of incidents in the Paris Metro is supported by regularly acquiring position information on all localizable professional users and visualizing their positions; by capturing, dispatching and viewing incident details; and by executing the applicable incident procedure. Another important application is patient management and remote health status monitoring in cases where a few paramedics have to treat many patients at different locations [SS+07]. Such data is aggregated at sensors and provided to the application at regular intervals or when a threshold is reached. Besides these mission critical applications, others could be envisioned to support the logistics, to collect evidence, etc. Instead of aiming here for an extensive list of useful applications, we want to point out that the applications have different requirements with respect to reliability, availability, data quality, and also general Quality of Service.

With respect to establishing and maintaining a Sparse MANET and information sharing services, we distinguish six phases in such a scenario [MD+06]: *A priori*, before the accident, the different organizations will exchange information on data format and make agreements on working methods. After an accident has happened, the first step is *briefing* of the different rescue teams, involving gathering of information about the disaster, e.g., weather, location, number of people involved, and facilities in the area. The next phase is the *bootstrap* of the network where events such as registration of nodes and electing leaders take place. During the *running* of the network different events may happen that will affect the middleware services: a node may join or leave the network, the network may be partitioned, and network partitions may be merged again. The middleware must support sharing of information during this phase, which typically lasts a few hours to at most a few days. During this time, devices can lose contact to other mobile devices due to network partitioning or power drain, but groups that are portioned off from other parts of the Sparse MANET should still function as good as possible. Therefore, replication is necessary to achieve the required level of availability. At the *end* of the rescue operation, all services must be terminated. *After* the rescue operation it could be useful to analyze resource usage, user movements, how and what type of information is shared, etc., to gain knowledge for future situations and to improve the middleware and the applications. Furthermore, it is important that it is possible after the operation to trace in detail what has happened, especially in case any (human) failure happened. Such an auditing is required for example to identify the responsible person and the reason for failure, etc. Therefore, data that is important for auditing must not be deleted or altered during the running phase.

3 Consistency Management in Sparse MANETs

The most important properties of Sparse MANETs for emergency and rescue operations that impact consistency management are the following: (1) there is no global clock. Device clocks might be synchronized during the briefing phase, but there will always be a certain drift between the clocks in the running phase. (2) Bandwidth is a scarce resource and propagation times might be long due to network partitioning. Other network dynamics include merging of partitions and late joining nodes. It is important that applications do not stop to work just because they might cause inconsistencies or because they might not have the *correct* value available. Furthermore, there are many different applications that communicate through the data space. In such a data centric approach a single conflict resolution strategy that fits the needs (and semantics) of all applications might be very hard to develop.

Saito and Shapiro [SS05] provide an overview of replication system design choices. In the following, we build on their terminology and classification to position our application domain requirements and design choices.

Pessimistic vs. optimistic replication: Since pessimistic replication blocks access to replicas that are not provably consistent, operations will block indefinitely in the presence of an unavailable site. This is not acceptable given the performance requirements of the application domain. Optimistic replication relies on the assumption that consistency problems are infrequent and can be fixed after they happen. Practical experiments support this assumption, e.g., for file sharing [RH+94]; the assumption is likely to hold also in our approach where relational tables are the unit of shared data. Simulations are needed to substantiate this assumption. Optimistic replication allows sites and users to remain autonomous and permits quick feedback by releasing tentative operations as soon as they are submitted.

Single and multi-master systems: A *master* is a node with a local data replica and write permission. In a single master system, in case of network partitioning, each partition may have to establish a master, e.g., to allow cooperative work to continue in partial isolation. In case of network remerging, we then face a multi-master situation and we need to elect a new single master. Thus, the benefits from the simplicity of design offered by single master systems are lost. Multi-master systems face operation scheduling problems and conflict management, but provide higher availability and performance, which are important requirements of the application domain. Our aim is therefore to design a multi-master system where master replica nodes are chosen dynamically in accordance with network topology and node resource profiles, balancing the number of replicas against propagation and synchronization costs.

State vs. operation transfer: To minimize networking overhead, we avoid transferring complete tables. It is sufficient to transfer information about changes to a table, i.e., we support operation transfer.

Propagation strategy: Since lives may be at risk, it is vital that changes are pushed efficiently to all reachable replicas. Due to network partitioning, we cannot expect to

reach all replicas during the push phase, thus this must be complemented with epidemic propagation, rendering a hybrid propagation strategy. Appropriate propagation protocols are the subject of ongoing work and outside the scope of this paper.

Consistency guarantees: In our application domain, lost operations are unacceptable. On the other hand, applications cannot stop working just because some remotely submitted operations are not yet fully propagated and might perhaps cause an inconsistency. In optimistic replication systems, tentative operations are made available locally at their submission and propagated to remote sites in the background. Such systems can at most offer *eventual consistency*, i.e., the state of replicas will converge only eventually.

Since the life time of the system is in the range of hours to few days, storage space does not represent a problem. Additionally, auditing processes require that some kinds of data never be deleted or altered. The basic idea of our approach is therefore to never delete or overwrite data, but instead just to append new versions. Thus, two or multiple replicas can be synchronized by adding to each replica missing data item versions. By this, all data, respectively all versions, are after the rescue operation available for auditing. Furthermore, such an approach enables each application to use its own policy for conflict resolution. In traditional application scenarios, the overhead of such a solution in terms of storage space that is needed to store all versions, is unacceptable.

The special characteristics of the emergency and rescue application domain, impose two important consequences for which our solution differs from most of the existing solutions: (1) there is no *primary* master replica, and (2) the replicas do not necessarily have their origin in one initial replica. Assume two rescue teams that approach the emergency site from opposite sides. In order to work immediately, both establish a MANET and a shared data space as illustrated in Figure 1-(a). In order to register injuries, they might in both network partitions each create a table called `Patients` and add to each their local version of new data. Before network merging, the system even does not know that there are two instances of the same table with different values. Obviously, when the MANETs merge (Figure 1-(b)), the system needs to identify the two instances (replicas) of the same table and to update each other by adding the missing data versions.

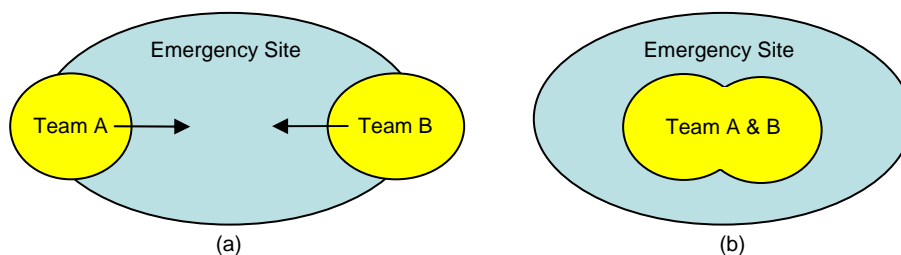


Figure 1: (a) Independent partitions and (b) merging of partitions

Two notable mobile information systems that use optimistic replication, are Bayou [TT+95] and Roam [RR+04]. Bayou is a multi-master system that supports eventual

consistency in arbitrary and dynamic communication topologies. It lets mobile users replicate a database, modify it, and synchronize with any other replicas when inconsistent versions are encountered. Update operations in terms of SQL statements are propagated epidemically. Applications specify how to detect and resolve conflicts. A primary, centralized master is responsible for committing updates. Roam [RR+04] is a multi-master system for nomadic users (like PDAs) and combines a client-server and peer-to-peer model to provide the nomadic users with shared volumes of data. The replicas are clustered into peer-based dynamic groups called wards. Synchronization is performed between pairs of ward members; in addition a (dynamic) ward master keeps knowledge of all ward members and has some synchronization responsibility. Each ward forms a semi-structure of servers. Per-ward version vectors keep track of updates and are used to detect possible inconsistencies. Both Bayou and Roam rely on the presence of a primary master and require that the shared unit has one initial replica, thus not allowing more than one network partition to form replicas for a given shared unit initially. Also, since synchronization is not performed eagerly, our requirements to high availability of data are not met.

4 Append-Only Approach

To explain our approach in more detail, we make the following assumptions for our data space solution: The data space aims to resemble a relational database system or tuple space for the applications and provides SQL-like operations to create tables, insert, update, read, and delete records. The schema of the data space is defined during the *a priori* phase and known to all applications. The objects (units of shared data) that are replicated in the data space are the tables that are defined in the schema.

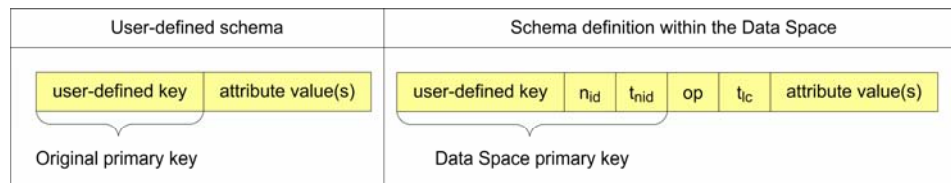


Figure 2: Generic record structure

Data Space schema definition: In order to support data versioning, each record not only contains the user-defined attribute values, but an identifier of the node that originally submitted the operation that lead to the value, denoted n_{id} , the local time the operation was submitted at n_{id} , denoted t_{nid} , and the type of operation, denoted op . Furthermore, we include the time the value was inserted into the local replica, denoted t_{lc} , in order to enable auditing processes after the operation to identify which information was known at which time and at which node and in which network partition. The resulting generic structure of each record comprises a user-defined key, the version specific timestamps, originator id, operation type, and the attribute values, as illustrated in Figure 2. The two fields n_{id} and t_{nid} together with the user-defined key constitute the primary key.

With these simple preparations at hand, we can now explain how the different operations work. The creation of a table has the classical semantics; the only restriction is that it has to follow the predefined schema. In contrast to this, the other operations have not anymore the classical semantics. In the following, we describe first the modify operations, i.e., insert, update, and delete, because they are handled quite similarly within the data space. Afterwards, we describe our approach for replica synchronization and the consistency model we support, before we present the read operation.

Insert, update, and delete operations: Applications invoke `Insert_record()` with the table name in which the record should be inserted and its user-defined key and attribute value(s). Internally, the data space adds the values for n_{id} , t_{nid} and t_{lc} and the type of operation (“insert”) to the record before it is actually inserted into the specified table. The `Update_record()` is very similar to `Insert_record()`. The main difference is that internally the `Insert_record()` operation is used (including also n_{id} , t_{nid} , and t_{lc} , but with $op=$ “update”) and normally an existing user-defined key is specified, while insert typically uses a new primary key because the application wants to insert a new record. The `Delete_record()` operation is internally also changed to an insert record operation and in addition to the values for n_{id} , t_{nid} , t_{lc} and op , it also sets all attribute value(s) to a tombstone value that marks the record as deleted. Table 1 summarizes the operations invoked by the application and the corresponding operations and their parameters performed within the data space.

Operations invoked by applications	Operations executed within the Data Space
<code>Insert_record(tableName, key, value(s))</code>	<code>Insert_record(tableName, key, n_{id}, t_{nid}, t_{lc}, “insert”, value(s))</code>
<code>Update_record(tableName, key, value(s))</code>	<code>Insert_record(tableName, key, n_{id}, t_{nid}, t_{lc}, “update”, value(s))</code>
<code>Delete_record(tableName, key)</code>	<code>Insert_record(tableName, key, n_{id}, t_{nid}, t_{lc}, “delete”, tombstoneValue(s))</code>

Table 1: Summary of insert, update, and delete operations

One interesting question is how the semantics of `Update_record()` has to be defined in case the operation tries to update a deleted record. A naïve solution would be to let `Update_record()` return with an error message. In order to explain why this is not an acceptable approach, we assume two network partitions. In one partition a record is deleted and in the other partition the applications work happily with this record and even create new versions long after the record has been deleted in the other partition. As long as the partitions are not merged or a node storing the corresponding table moves from one partition to the other, this has no impact. The system can only work from its local knowledge and the conflict is not visible within the two partitions. However, if they merge or a node with the table moves to the other partition, the conflict is visible in at least one partition. As mentioned earlier, the naïve approach would be to let the tombstone version be the dominant version. This solution seems not very promising, because if some applications have the need to use the record, it should not be possible for another application to remove it. Obviously, it should be also asked whether there is a need at all to provide a delete operation to the application. If a delete operation is provided, it should also have some corresponding semantics. Currently, we use the

simple approach to allow such an update and just create a new version. The advantage of this simple solution is that conflict resolution has only to be performed in the read operation.

Thus, the design chosen is to *always append records for each modify operation*, and even stronger, *not to decide on behalf of an application how to interpret the semantics of a set of records that share a user-defined key*. The data space simply stores and propagates (the result of) all invoked operations, whether they originate from an insert, update, or delete. Now, the application can seek support from the data space in the action of interpreting the records, by supplying the data space with a strategy or even an operation describing how to choose the *correct* value(s) during read. Such *filtering*, in contrast to other comparable systems, is not done at the time of object/table synchronization, but rather when performing `Read_record()` operations. Thus, consistency checks and conflict handling both can be performed at the leisure of an application and do not delay the production and propagation of data/operations.

Replica synchronization: In order to propagate the changes, i.e., new records that are caused by insert, update, and delete record operations, we use both a push-based approach and a polling-based approach. The push-based approach is a very efficient means to propagate the new records to the known and reachable copies of the table. It basically means that to each node that hosts a replica, an insert record request is forwarded. However, we can neither assume that we know about all replicas nor that we can reach all known replicas. Therefore, we need to additionally employ an epidemic style of operation propagation. Epidemic propagation is slower than push-based propagation, but it is the only means to reach eventual consistency if all nodes have been for a sufficiently long time in communication range.

Only for the epidemic propagation, we need to detect conflicts, i.e., two or more replicas have different record (versions). To perform this efficiently, we could use a two-level synopsis-based approach. On the record level, we compute for all versions of one record/primary key a synopsis, by using e.g. a hash function. On the table level, all synopses are again combined into a single value with the help of a hash function. In order to compare whether two table replicas are out of sync it is sufficient to compare the two table synopses, and the record synopses can be used to identify quickly which records respectively record versions are missing.

Consistency model: Our solution supports *eventual consistency*, in the following sense:

- If no more insert, update, or delete operations with respect to a given table are invoked within a network partition, and the set of nodes in the network partition remains unchanged sufficiently long (no added nor removed nodes, and all nodes in the partition continue to function properly), then the propagation and synchronization protocols ensure that all the table replicas in the partition will eventually get identical contents when ignoring the t_c attribute.
- For each user-defined key and each replica, the set of records for that key grows monotonically in time.

- For every invoked insert, update, or delete operation with respect to a given table and user-defined key, the corresponding data space insert operation will eventually be performed on at least one node containing a replica of the table.

Read operation: Notice that eventual consistency as defined above, only ensures that all invocations of the insert, update, and delete operations are recorded and propagated; checking whether a collection of such versioned records for a given user-defined key can be (re-)organized to prove the absence of data conflicts, e.g., by demonstrating single-copy consistency or client-based consistencies for that key, is not the concern of the data space. The read operation might thus reveal that there are data conflicts even if all tables are synchronized, because there is more than one record with the same user-defined key in the table. We are currently investigating the following policies to address the conflict:

- Return all versions to the application, including n_{id} , t_{nid} , t_{lc} and op , to enable the application to use its own policy to decide on which value to use.
- Regard a conflict only as a conflict for the application if there are multiple versions for the most recent time window, i.e., 10 seconds, because the drift between the different time clocks is much smaller than the window.
- Add a conflict resolution layer to the data space: The application specifies a policy or supplies a procedure for how to handle conflicts (see Table 2), like in [JL06] the conflict resolution layer resolves the conflict on behalf of the application. Part of the conflict resolution layer's responsibilities is to manage vector clocks and other data structures for supporting application-specified consistency policies.

Operation invoked by applications	Operation executed within the Data Space
Read_record(tableName, key)	Read_record(tableName, key, readPolicy _{appl})

Table 2: Read operation

5 Conclusions

In this paper, we have discussed a new approach for consistency management in shared data spaces for emergency and rescue operations. The goal of this work is to show how application specific requirements can be leveraged to solve the hard problem of consistency management for optimistic replication in Sparse MANETs. It should be noted that the fundamental problem to solve differs from most other approaches for optimistic replication and consistency management, because there cannot be any primary master copy of a table, and it cannot be assumed that different versions start from the same initial master copy. It is not the goal of this paper to present a full and perfect solution. We have only focused on the consistency management aspects; obviously many more elements are needed to realize the envisaged shared data space. The interesting insight is that the hard problem of consistency management during replica synchronization can be quite easily and elegantly solved by removing one fundamental assumption that is made for all data management systems (except append-only data management systems), i.e., data needs to be deleted and storage space is a scarce

resource. By never deleting data we do not only reduce the consistency problem to a synchronization problem, but also achieve three additional benefits: (1) auditing after the operation is automatically supported, (2) conflicts need only be resolved when an application reads a multi-versioned record, and (3) each application can apply its own policy to resolve conflicts.

Acknowledgements

This work has been performed in the context of the Midas project (funded by European Commissions 6th Framework Program, Contract no. 027055) and the Ad-Hoc InfoWare project (funded by Norwegian research Council IKT2010, Project No. 152929/431).

References

- [DG+88] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic Algorithms for Replicated Database Maintenance, ACM SIGOPS Operating Systems Review, Vol. 22, No. 1, Jan. 1988, pp. 8-32
- [G06] J. Gorman, The MIDAS Project: Interworking and Data Sharing, Interworking 2006, Santiago, Chile, January 2007
- [KG+06] E. Klintskog, P. Gründler, K. Skjelsvik, S. Pérez, Proof-of-Concept Application Requirements, Midas Deliverable D5.3, Nov. 2006
- [MD+06] E. Munthe-Kaas, O. Drugan, V. Goebel, T. Plagemann, M. Puzar, N. Sanderson, K. Skjelsvik, Mobile Middleware for Rescue and Emergency Scenarios, in: Mobile Middleware, P. Bellavista, A. Corradi (Eds.), CRC Press, Sept. 2006
- [PA+04] T. Plagemann, J. Anderson, P. Halvorsen, O. Drugan, V. Goebel, C. Griwodz, E. Munthe-Kaas, K. Skjelsvik, M. Puzar, and N. Sanderson, Middleware Services for Information Sharing in Mobile Ad-hoc Networks - Challenges and Approach, Workshop on Challenges of Mobility, IFIP TC6 World Computer Congress 2004, Toulouse, France, Aug. 2004
- [RC05] International Federation of Red Cross and Red Crescent Societies (IFRC), World Disasters Report 2005, Oct. 2005, <http://www.ifrc.org/publicat/wdr2005/index.asp>
- [RH+94] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, G. Popek, Resolving File Conflicts in the Ficus File System, Proceedings of the Summer 1994 USENIX Conference, Boston, MA, USA, 1994, pp. 183-195
- [RR+04] D. Ratner, P. Reiher, G.J. Popek, Roam: A Scalable Replication System for Mobility, Mobile Network Applications, Vol. 9, No. 5, Oct. 2004, pp. 537-544
- [SS05] Saito, Y. and Shapiro, M., Optimistic Replication, ACM Computing Surveys, Vol. 37, No. 1, March 2005, pp. 42-81
- [SS+07] K. Skjelsvik, J. Sørberg, V. Goebel, T. Plagemann, Using Continuous Queries for Distributed Event Notification, 11th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2007), Arizona, USA, March 2007
- [TT+95] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, C.H. Hauser, Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System, Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP'05), Copper Mountain, Colorado, USA, Dec. 1995, pp. 172-182
- [JL06] J. Jittamas, P. Linington, Using a Policy Language to Control Tuple-Space Synchronization in a Mobile Environment, Policy 2006, Ontario Canada, June 2006