Institute of Parallel and Distributed
High Performance Systems (IPVS)

Department of Computer Science

University of Stuttgart

Breitwiesenstr. 20-22
D-70565 Stuttgart
Germany

Consistent Update Diffusion in Mobile Ad Hoc Networks

*Kurt Rothermel, Christian Becker,
Jörg Hähner*[*]

Technical Report 2002-04

July 2002

[*] The work of Jörg Hähner is supported by the Gottlieb Daimler and Karl Benz Foundation

Abstract

Applications of mobile ad hoc networks (MANETs) occur in situations, where networks need to be deployed immediately but network infrastructures are not available. If MANET nodes have sensing capabilities, they can capture and communicate state of their surroundings, including environmental conditions or other nodes in proximity. If the sensed state information is propagated and collected in a database, this allows for a variety of promising automatic monitoring, tracking and navigation applications, using global state information to built up models of reality.

Since state changes represent events happening in reality, applications are typically interested to see the most recent state. Also the order of state changes should be consistent with the corresponding order of events in reality. In particular, preserving consistency becomes a challenging research problem if there are multiple nodes sensing the same object, either subsequently or even concurrently.

In this paper, we introduce a generic model of state propagation in MANETs and propose two consistency levels for this model. For each of these consistency levels, we define a state propagation algorithm based on information diffusion. Our simulations show, that for typical scenarios the additional synchronization overhead for achieving the proposed consistency levels is low. In terms of communication overhead and state propagation latency one of the proposed algorithms shows a similar performance as the underlying flooding mechanism.

1 Introduction

In the near future, advances in processor, memory and radio technology will enable small mobile nodes that are capable of communication, computation and sensing. The integrated sensing devices may detect a node's position and orientation as well as measure and test for changes in its environmental conditions, including hazardous emission, temperature, smoke or dust. For proximity and presence sensing various promising technologies, such as Radio Frequency Identification (RFID) are rapidly emerging. Active RFID tag technology has been designed to locate and track mission-critical people and assets. Although this technology is still in an early stage, read ranges of currently available active RFID tags are already up to 35 feet.

Their sensing capabilities allow nodes to monitor their physical context and gather context information. Although context-aware applications can already profit from locally gathered state information, the benefit is much greater if they have access to global state information, i.e. the collection of locally captured states. Access to global state enables to maintain more or less complex models of reality which can be accessed by each node. If not only the observed state but also – if possible – the relation to other observations with respect to their temporal ordering is propagated, applications can derive additional information about the environment. For an example consider a mobile object that has been observed by two consecutive observations. If the ordering of the observations is known, the heading of the object can be deduced.

To motivate our research, we consider the following fire fighting scenario. A number of fire fighters have been assigned to extinguish a fire in a building. Among the fire fighters are a number of officers, who are in charge to collectively manage the entire operation.

Each fire fighter carries an active RFID tag and a small communication device, which is equipped with appropriate environmental sensing devices. RFID tags are also attached to the mission-critical equipment of the fire fighters' (e.g., water pumps), some of which may signal their current state (e.g., pump pressure, energy level). In order to locate both fire fighters and devices, officers carry RFID receivers with communication capabilities. When the fire fighters move into the building, those receivers are also placed at neuralgic points in the buildings, like entrances or hall ways. Other relevant objects within the building, e.g. a tank for gas or other chemicals, might be provided with a sensor indicating their condition, e.g. temperature.

From a system's point of view, the fire fighters and their equipment are "perceivable" objects that are associated with state information. Each object has a relative location (e.g., in team of officer A) and an absolute location (e.g., in wing B of second floor). Objects are typically associated with further state information, such as pump pressure or temperature. If the officers have access to the state of all perceivable objects, they get a global picture of the entire operation. That is, they know where fire fighters and their equipment are located, how they are grouped into teams, and what are the fire fighters' current environmental conditions and the equipment's state. If consecutive observations indicate a rise of temperature in a gas tank either cooling or evacuation of the surrounding area can be forced. This information may significantly increase the efficiency of the operation as well as the safety of the fire fighters.

Applications of mobile ad hoc networks (MANETs) occur in situations like the one described above, where networks need to be deployed immediately but (operational) base stations or fixed network infrastructures are not available. To allow access to global state information in MANETs one can follow one of two approaches, *update diffusion* or *query diffusion*. In the case of update diffusion, sensed state changes cause updates to be propagated to a database, which is typically replicated over a number of nodes. The entire state information is stored in the database, and for state retrieval, applications access an available database copy. In contrast, with query diffusion, state information is only stored at the nodes, where it has been sensed. For state retrieval an application broadcasts or geocasts a query specifying the requested state information. State information matching the query is sent back to the application. While this approach has its merits in sensor networks with stationary nodes, it has two serious disadvantages in MANETs. Firstly, for partitioned networks the availability of state information is poor since each state item is stored on a single node only. Of course, caching can be used to increase availability, however at the cost of introducing the problem of stale caches. Secondly, in many situations, as in the scenario described above, applications should learn about state changes as fast as possible. This is hard to achieve with pull approaches like query diffusion. Therefore, we decided for the update diffusion approach. In our fire fighting scenario, on each officer's device there exists a copy of a database that stores the entire state information. Whenever a state change occurs, the new state information is propagated to each database copy.

The problem that comes with update diffusion is the danger of inconsistencies. In the realm of databases numerous replication management algorithms have been proposed to achieve strong consistency (e.g., see [BHG97]). However, applying those algorithms in MANETs, where network partitioning is supposed to occur frequently, would result in reduced availability of data, while also incurring unacceptably high communication overhead. For many applications the availability of data is more important than strong consistency, e.g., an officer querying the states of the fire fighters located in a given wing would prefer (slightly) outdated state information to the alternative of getting no information. This fact has lead to the definition of weaker consistency levels in various areas, such as directory systems (e.g., [Nee93]) or distributed file systems (e.g., [Sat93]). In the context of sensing, ideally all DB copies of an object should reflect this object's state in reality. Here, weak consistency has been defined to be given if all copies of an object converge to the most recently propagated state, where it may happen that a fraction of copies are up-to-date, while others are stale [KMP99]. A precise definition of our notion of weak consistency will be given below.

If there only exists a single source of updates for each object, the most recently propagated state can be easily determined by an appropriate versioning scheme. However, this becomes a more challenging task if there are multiple update sources per object and synchronized clocks are not available. For example, while moving through the building, a fire fighter will enter the read range of several RFID receivers, propagating an state update message each time. Two "current pressure" signals successively communicated by a water pump may be received by different communication devices, each propagating an update. Multiple update sources can occur in many realistic sensing environments.

In this paper, we address update propagation taking into account that there might exist multiple update sources per object, propagating updates subsequently or even concurrently. In particular, the contributions of the paper are as follows: We propose a generic model for update diffusion and define two consistency levels, called local observation consistency and global observation consistency. For each of these consistency levels, we describe an update diffusion algorithm based on flooding. Finally, simulations compare the algorithms with flooding in three typical

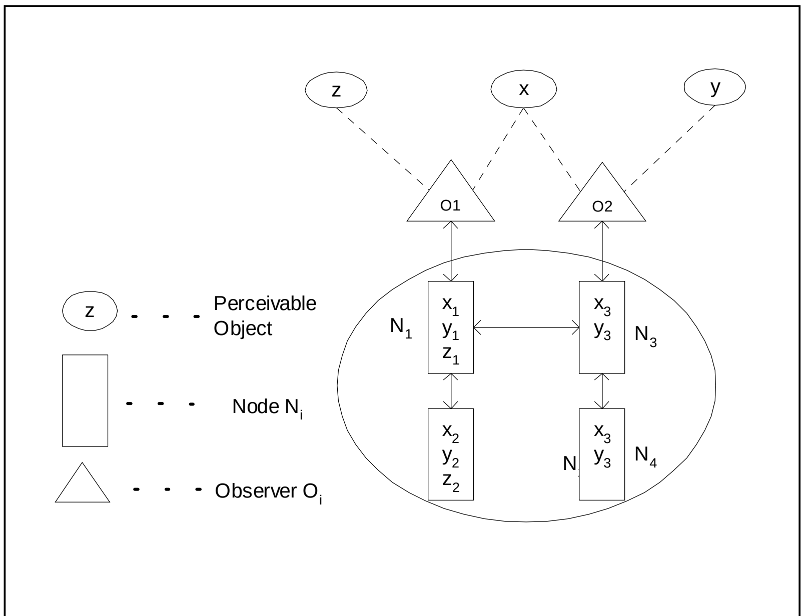


Figure 1: Observation Model

scenarios. While consistency in the local observation consistency results in 50% more messages and a doubled latency global observer consistency performs the same as flooding. However, additional effort is spent on maintaining ordering information on the node.

The remainder of the paper is structured as follows. The next section will present our system model introducing the system model and the two consistency levels. Section 3 discusses the assumptions regarding the underlying dissemination algorithm. The two protocols ensuring the consistency levels are introduce in section 4 and informal correctness proofs are given. Simulations comparing the protocols with plain flooding are presented in section 5 before related work is discussed. The paper closes with a conclusion and an outlook to further work.

2 Update Diffusion Model

The major components of our model are observers, nodes and perceivable objects (see Fig. 1). Each ***perceivable object*** (or object for short) has a globally unique identifier and is associated with state information, which may change over time. Perceivable objects may be stationary or mobile. For example, the fire fighters are equipped with RFID tags reflecting identifying them individually. Sensors for smoke or temperature are placed at neuralgic places providing environmental information. Even image recognition might be used to track objects not being equipped with RFID tags, e.g. people still residing in the building, and propagating this information

An ***observer*** can sense perceivable objects that reside in its observation range. For each observed object, an observer captures the object's identifier and the state information associated with it. There might be multiple observers that observe the same object, consecutively or even concurrently. Observers also may be stationary or mobile. The sensors for environmental conditions and position monitoring will be typically placed at appropriate locations within the mission area. However, some fire fighters might be equipped with such sensors as well in order to provide additional information about the environment as they move.

When an observer notices a state change of an object, it propagates the new state to a database (or DB for short). For each observed object, this DB stores a record consisting of the object's identifier and its current state. As will be seen below, the DB is replicated over multiple mobile ***nodes*** (see Fig. 1). Applications running on those nodes access the local DB to read the observed objects' states. In other words, observers update the DB, while applications only read it. A device can support both roles observer and node, e.g. the officer's device being equipped with sensors as well as running the tracking and management application.

In our model, we distinguish between observers and ***observations***. When an object moves into the observation range of an observer, a new observation is started, and this observation ends when the object leaves the observation range. Consequently, an object entering the observation range of a given observer several times results in multiple observations. Each observation observes a single object, for which it maintains a state record. If the state information captured by the observer's sensors significantly differ from the recorded state, the latter is updated accordingly and propagated to the DB, where the definition "significant" is application-dependent.

An observation can be modelled as a sequence of transitions of the observed object's state, where each state transition is propagated to the DB. We will use the following notation for denoting an observation, say *o*: $s^o_1 \rightarrow s^o_2 \rightarrow s^o_3 \rightarrow \ldots \rightarrow s^o_{j-1} \rightarrow s^o_j$, where edge $s^o_{i-1} \rightarrow s^o_i$ represents a state transition from state s^o_{i-1} to state s^o_i. We call this a ***local observation graph***.

Our notion of consistency introduced below is based on the ordering of observations. Let *o* and *o'* be two observations of the same object, say *x*. We define *o* to ***occur before*** *o'* if *o* was finished before *o'* was started, and will use notation $o <_x o'$ to express this relationship. Note that if $o <_x o'$ or $o' <_x o$, observations *o* and *o'* do not overlap in time. We define *o* and *o'* to be ***concurrent*** if neither $o <_x o'$ nor $o' <_x o$ holds, and will use notation $o \|_x o'$ to denote this relationship. Note that concurrent observations at least partially overlap in time.

There might exist multiple observations of the same object, concurrently or consecutively. The collection of observations of an object, say *x*, is modelled by a so-called ***global observation graph***, which is composed of the local observation graphs of all observations of *x*. This graph links *x*'s local observation graphs according to the $<_x$-relationship. More precisely, for each observation *o* and *o'* with $o <_x o'$, the global observation graph's set of edges includes $s^o_{Last} \rightarrow s^{o'}_1$, where s^o_{Last} is the last state in *o*'s state transition sequence and $s^{o'}_1$ is the first state in the sequence of *o'*. Fig. 2 shows three examples of global observation graphs of an object. The graph in Fig. 2a links observation *o1* and *o2*, where $o1 <_x o2$. The graph in Fig. 2b includes observations *o1*, *o2* and *o3*, where $o1 <_x o2$, $o1 <_x o3$ and $o2 \|_x o3$. In the graph given in Fig. 2c, observations *o2* and *o3*, which are concurrent, occur before *o4*.

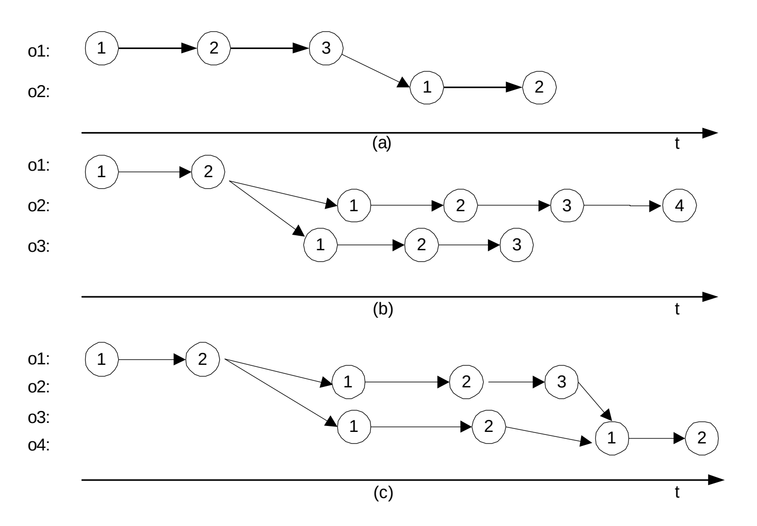


Figure 2: Observation graphs

As stated above, the DB is supposed to be replicated over a number of mobile nodes, and hence multiple copies of an observed object's state record exist. We will use x_i to denote a copy of *x*'s state record and *Copies(x)* to designate the set of all state record copies of *x*. When a state change is observed for some object *x*, the new state must be propagated to all *Copies(x)*. Since there may exist several (potentially concurrent) observations propagating state updates for the same object, an update algorithm has to obey some rules to avoid that the copies diverge. Those rules depend on the consistency level that is to be achieved. In the following we will define two consistency levels, local observation consistency and global observation consistency.

The DB is defined to be ***local observation consistent*** if for each object *x* in the DB and for each of $x_n \in Copies(x)$ the following holds:

(C1) x_n will eventually converge to the most recently propagated state of *x*.

(C2) Once x_n has reached state s^o_i, it will no longer accept a state s^o_j with $j < i$.

Condition (C1) ensures that all copies of *x* converge into the most recently propagated state. We assume here that observations observing the same object at the same time deliver equivalent states, where the definition of state equivalence is application-dependent. In the observation graphs depicted in Fig. 2a, 2b and 2c, the most recently propagated state is s^{o2}_2, s^{o2}_4, and s^{o4}_2, respectively. Condition (C2) ensures that a state propagated by a given observation is never overwritten by an older state propagated by the same observation. In other words, the ordering of state transitions need to be consistent with *x*'s local observation graphs in the sense that a state transition from *s* to *s*' is only allowed if *s*' follows *s* in one of these graphs. Note, however, that there are no restrictions concerning states propagated by different observations. For example, for the observation depicted in Fig. 2a the sequence state updates s^{o1}_1 s^{o2}_1 s^{o1}_3 s^{o2}_2 applied to some copy is local observation consistent. Notice that although *o1* occurred before *o2*, a state propagated by *o2* is overwritten by a state of *o1*. As this might not be acceptable for various applications, we introduce a stronger type of consistency. When two subsequent observations of the same object happen, applications can deduce distinct properties, e.g. the heading of an object when it passes different observers. A fire fighter first passing the receiver in the first floor and then one in the second floor can be considered to be moving upstairs.

A DB is defined to be ***global observation consistent*** if it is local observation consistent and additionally for each object *x* in the DB and for each $x_n \in Copies(x)$ the following holds:

(C3) Once x_n has reached state $s^{o'}_j$, it will no longer accept a state s^o_k for $o <_x o'$ and any *j* and *k*.

Condition (C3) ensures that a state propagated by a given observation is never overwritten by a state propagated by an observation that occurred before. In other words, the ordering of state transitions need to be consistent with *x*'s global observation graph in the sense that a state transition from *s* to *s*' is only allowed if *s*' follows *s* in this graph. For the global observation graph depicted in Fig. 2a, the sequence of state updates s^{o1}_1 s^{o1}_3 s^{o2}_1 s^{o2}_2 applied to some copy is global observer consistent. Note, however, that there are no restrictions concerning the ordering of states belonging to concurrent observations. For example, the state update sequence s^{o1}_1 s^{o1}_2 s^{o2}_1 s^{o3}_1 s^{o2}_2 s^{o3}_3 s^{o2}_4 applied to some copy is global observer consistent for the graph depicted in Fig. 2b.

3 Assumptions

In the following, we will introduce the assumptions, the algorithms described in the next section are based upon.

We will assume that object states are replicated on each node, i.e., each node contains a replica of the DB. Diffusion mechanisms are applied to propagate state changes from node to node, which update their copies accordingly. Nodes and observers are assumed to form a mobile wireless ad hoc network (MANET). Two devices (node or observer) are called neighbours while they are in each others communication range. On each device we assume the existence of a discovery mechanism that maintains a *Neighbours* list to keep track about the device's current neighbours.

Our state diffusion algorithms are based on a flooding mechanism. For MANETs several such mechanisms have been introduced in the literature (e.g., see [Ni+99, Ho+99]). To avoid so-called broadcast storms various scoped flooding techniques have been proposed in [Ni+99]. For example, probabilistic, distance-based, or even location-based schemes can be applied to restrict the scope of flooding. Furthermore, "hyper flooding" [OTV01] can be used to increase the packet delivery ratio if network partitioning happens frequently [Ho+99]. With this approach, references to previously updated DB records are stored in a cache, where cached states are re-broadcasted when a new neighbour node becomes available. Either the entire cache is broadcasted or some heuristic can applied to select an appropriate subset of states. A more complex approach is to synchronize databases when two nodes become neighbours, and flood the updates resulting from synchronization. Note that flooding techniques incorporating state caching or DB synchronization techniques increases the "lifetime" of update messages as compared to plain or scoped flooding [OTV01].

It has been shown that the chosen flooding technique can significantly impact both communication overhead and packet delivery ratio [HKB99][Ho+99]. Whether or not a given flooding mechanism is appropriate depends on the application and the characteristics of the underlying MANET. That is the reason why we have tried to make our diffusion algorithms independent on the underlying flooding technique. We assume a communication primitive ***f-forward***, which forwards a message to the sender's neighbour nodes according to the underlying flooding scheme. It is also assumed that f-forward applies an appropriate randomisation technique to reduce the probability of collisions. Moreover, we will assume that a state f-forwarded by an observer will eventually reach each node. Note that this assumption is a prerequisite for condition C1 to be fulfilled for all copies. It is important to mention, however, that our diffusion algorithms also work if an object's most recent state can only be f-forwarded to a fraction of copies. In this case, of course, C1 can be only fulfilled for this fraction of copies.

Any kind of sensor technology can be used to perceive objects, and we make no assumptions about the sensed objects itself. In particular, we do not require objects to support versioning of state information. Clearly, an object sensed visually or identified by RFID technology cannot perform state versioning itself. Therefore, we assume that versioning of states is done by observers.

Finally, we do not require the clocks of observers and nodes to be synchronized. Clock synchronization is at least a difficult task in frequently partitioned networks and consumes additional bandwidth and energy.

4 Update Diffusion Algorithms

In the first part of this section, we will present a diffusion algorithm that supports local-observer-consistency. This algorithm is extended in the second part to also provide for global-observer-consistency. While removal of DB objects are not considered in the first two parts the last part of this section, the last part is devoted to this problem.

4.1 Supporting Local Observer Consistency: Algorithm 1

We will first introduce the data structures needed for observers and nodes before describing the observer/node and node/node protocol. Finally, informal correctness arguments are given.

4.1.1 Data Structures

For each active observation, an observer maintains a so-called *observation record* in data structure *Obs*. When an object enters an observer's observation range a new observation is created and an observation record added to data structure *Obs*. When an observation ends, the corresponding record is removed from *Obs*. An observation record is composed of the following components:

- ***Obj***: the observed object's unique identifier.
- ***State***: the observed object's propagated-state.
- ***VN***: version number of the propagated-state. *VN* is incremented whenever a new version of *State* is generated.
- ***ObsId***: This is the globally unique identifier of the corresponding observation. It has the form *ObserverId.ObservationCounter,* where the counter is incremented whenever a new observation record is created.

We will use notation *Obs(o)* to denote the observation record of observation *o*, while *Obs(o).c* denotes component *c* in this record.

A node's local DB contains an observation record for each locally known object. We will use *DB(x).c* to designate component *c* of object *x*'s observation record, while *DB(x)* denotes the entire DB entry of *x* and *DB* the entire local copy of the database.

4.1.2 Observer/Node Protocol

An observer broadcasts state information to its neighbour nodes whenever it observes a new object or a state change of an already observed object The state observed by an observation, say *o*, is transmitted in an *O_Update* message, which includes the observation record *Obs(o)*. A randomization technique is used to make sure that concurrent observers forwarding the same state information becomes a rare event. Each observer waits a random time before it transmits an object's new state. If in the meanwhile the observer receives another *O_Update* including equivalent state information for the same object, the transmission is withdrawn. Again note that the decision whether two states are equivalent is application-dependent. This mechanism is based on the valid assumption that in most cases observers observing the same object are close to each other and sense state changes at approximately the same time.

When a neighbour node receives an *O_Update* message, including observation record *r,* it proceeds according to the following cases:

(O-1) Object *r.Obj* is not in *DB*: *r* is included in *DB* and f-forwarded in an *N_Update* message to its neighbour nodes.

(O-2) Object *r.Obj* is in *DB*: The following subcases must be considered:

(O-2.1)*DB(r.Obj).ObsId* = *r.ObsId*: *r* and the record in *DB* are from the same observation, again two subcases must be considered:

(O-2.1.1) *DB(r.Obj).VN* < *r.VN*: *r* is younger than the record in *DB*. Therefore, the old record is replaced by *r* in *DB*, and *r* is f-forwarded in an *N_Update* message to its neighbour nodes.

(O-2.1.2) *DB(r.Obj).VN* ≥ *r.VN*: *r* is not younger than the record in *DB* and thus is ignored.

(O-2.2)*DB(r.Obj).ObsId* ≠ *r.ObsId*: *r* and the record in *DB* are from different observations, two subcases must be considered:

(O-2.2.1) *DB(r.Obj).VN* < *r.VN* : *r* replaces the old record in *DB* and is f-forwarded in an *N_Update* message to its neighbour nodes.

(O-2.2.2) *DB(r.Obj).VN* ≥ *r.VN*: *r* is not accepted. Instead, the observer is requested to synchronize its version number by sending a *O_Sync* message. This message includes a tuple (*r.ObsId*, *NewVN*,), where the latter component is the requested new value of the version number, which is set to*DB(r.Obj).VN*+1.

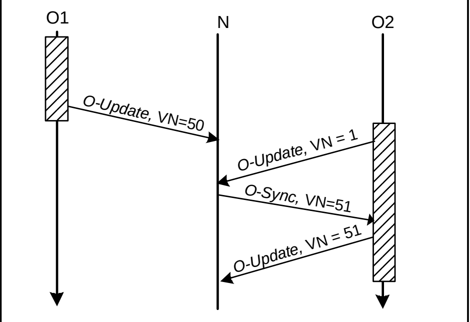


Figure 3: Synchronization of Version Numbers

When an observer receives *O_Sync(o, NewVN)* for observation *o*, it sets *Obs(o).VN* to max(*Obs(o).VN*, *NewVN*). Moreover, it f-forwards the updated observation record within an *O_Update* message to its neighbours.

Let's take a closer look at case (O-2.2.1). Since condition C3 does not have to be fulfilled, the node can accept the propagated state without knowing the ordering between *DB(r.Obj).ObsId* and *r.ObsId*. However, to satisfy condition C1, we have to make sure that the most recent state will eventually end up with the highest version number. That is why in case (O-2.2.2) the node requests the observer to increase its version number to *DB(r.Obj).VN*+1. Fig. 3 shows a scenario, where synchronization of version numbers is to be performed. In this scenario, observation *o2* follows observation *o1*, whose final *O_Update* message forwards an observation record with version number 50. The first observation record propagated by *o2* includes version number 1. Because of the returned *O_Sync* message, *o2* synchronizes its version number counter and f-forwards an observation record with version number 51, which then is accepted by the node.

4.1.3 Node/Node Protocol

When a node receives an *N_Update* message, including observation record *r*, it proceeds according to the following cases:

(N-1) Object *r.Obj* is not in *DB*: *r* is included in *DB* and f-forwarded in an *N_Update* message.

(N-2) Object *r.Obj* is in *DB*: The following two subcases are to be considered:

- (N-2.1) *DB(r.Obj).VN* < *r.VN* : Two subcases are to be considered:
  - (N-2.1.1) *DB(r.Obj).ObsId* = *r.ObsId*: *r* is younger than the record in *DB*. Therefore, *r* replaces the old record in *DB* and is f-forwarded in an *N_Update* message.
  - (N-2.1.2) *DB(r.Obj).ObsId* ≠ *r.ObsId*[1]: Since the ordering between these both observations is unknown, it cannot be decided whether *r* or the *DB* record is more up to date. Since, however, *r* can be accepted without violating C2, the old record in *DB* is replaced by *r*, which then is f-forwarded in an *N_Update* message.
- (N-2.2) *DB(r.Obj).VN* ≥ *r.VN*: *r* is ignored.

4.1.4 Correctness Arguments

Let us briefly argue why this algorithm ensures local observation consistency. Condition C2 is obviously fulfilled due to (N-2.2) and (O-2.1.2), which prevent a node from accepting an observation record that is older than the one in the DB.

To show C1 to be fulfilled we assume that f-forwarded observation records will eventually reach each node (see Sec. 3). The fact that observers increment version numbers with each state change in combination with the synchronization of version numbers in (O-2.2.2), ensures that the most recent state will eventually end up with the highest version number. Finally, the state with the highest version number will eventually be accepted by all nodes due to (O-2.1.1), (O-2.2.1), and (N-2.1). Consequently, condition C1 is also satisfied.

Although this algorithm achieves the required level of consistency, it suffers a performance problem in the presence of concurrent observations. As soon as there are multiple observations at the same time for the same object, the corresponding observations will indirectly hear from the version numbers generated by each other. Due to (O-2.2.2) they will cause themselves to mutually increase their version numbers constantly as long as they overlap in time. One way to avoid this problem is to let a node check whether the state included in an *O_Update* message received from an observer actually differs from its local DB state[2]. If the received state and the DB state are equivalent, the received state is not accepted and the version number not modified. Since we assume that concurrent observations capture equivalent states while they overlap in time, the above mentioned problem is avoided.

4.2 Supporting Global Observer Consistency: Algorithm 2

In this section, we will modify Algorithm 1 to provide for global observer consistency. Again we will first introduce the required data structures, then describe the observer/node and node/node protocol, and finally give informal correctness arguments.

[1] To speed up synchronization of version numbers, the following can be performed: If the observer performing *DB(r.Obj).ObsId* is in the node's *Neighbours* list, an *O_Sync*(*DB(r.Obj).ObsId, r.VN*+1*)* is sent to this observer. If this observation is still active, an *O_Update* with a synchronized version number will be triggered.

[2] This check is only to be performed if the received state an DB state are from different observations.

4.2.1 Data Structures

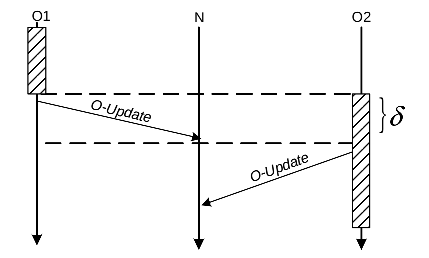


Figure 4: Deferring initial

Nodes and observers maintain data structures *DB* and *Obs*, respectively, which are structured as described for Algorithm 1. In addition, nodes have to maintain information about the ordering of observations to be able to fulfil condition C3.

Ordering relationships can be derived from interactions between nodes and observers. Assume that *o1* and *o2* are observations of the same object, say *x*, and a given node learns about *o1* before *o2*, i.e. the node receives the first *O_Update* of *o1* before the first *O_Update* of *o2*. We conclude from the ordering of these receive events that *o2* did not occur before *o1*, i.e., $o1 <_x o2$ or $o1 \|_x o2$. Of course, this conclusion is only valid if we can guarantee that the first *O_Update* of *o2* never arrives before the first *O_Update* of *o1* at any node if $o1 <_x o2$. This can be ensured by deferring the first *O_Update* of an observation by δ, where δ is the maximum message delay on a link (single hop). The scenario in Fig. 4 shows that the first *O_Update* of *o2* cannot arrive at the node before the one of *o1*.

For each object in its DB a node maintains a so-called ***ordering graph***, which specifies the locally known ordering relationships between observations of this object. Let $G_x=(V_x,E_x)$ denote the ordering graph for object *x*. The set of vertices, V_x, contains the locally known observations of *x*, where a node learns about observations by means of *O_Update* or *N_Update* messages, as will be seen below. The set of edges, E_x, defines the locally known ordering relationships between observations. A directed edge $o \rightarrow o'$ exists in the ordering graph if and only if either $o <_x o'$ or $o \|_x o'$ holds, i.e., *o'* did not occur before *o*.

Now, we will describe how observation graphs are constructed. When a node receives an *O_Update* message from an observation, say *o*, it performs the following graph management operations:

(GM1): $V_x \cup \{o\}$, and $\forall\, o' \in V_x \setminus \{o\}$: $E_x \cup \{o' \rightarrow o\}$.

Fig. 5 shows versions of an ordering graph G_x maintained by some node *N*. The graph in Fig. 5a indicates that $o1 <_x o2$ **or** $o1 \|_x o2$. Now assume that *N* receives an *O_Update* from observation *o3*, which causes G_x to be modified as depicted in Fig 5b.

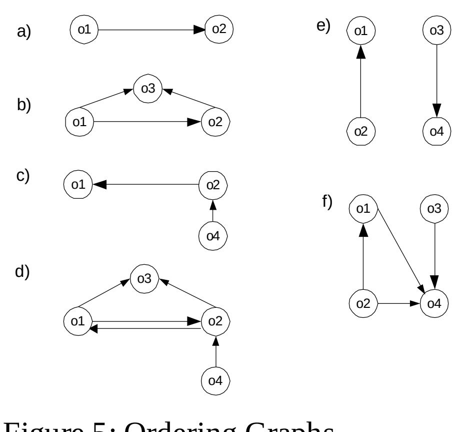


Figure 5: Ordering Graphs

Nodes forward ordering relationships by means of *N_Update* messages to their neighbouring nodes (for the details of this protocol see below). An *N_Update* message propagating an observation record for an object, say *x*, also includes the sending node's ordering graph G_x. A node receiving G_x in an *N_Update* merges G_x and its local ordering graph, say G'_x. It performs the following graph management operations:

(GM2): $V_x \cup V'_x$ and $E_x \cup E'_x$.

Note that due to the graph merging, it may happen that there are two oppositely directed edges between a pair of vertices, which indicates that these two observations are concurrent. For example, the merging of the graphs depicted in Fig. 5b and Fig. 5c results in the graph shown in Fig. 5d, from which we can derive that *o1* and *o2* are (definitely) concurrent.

A vertex $o \in V_x$ is defined to be a *sink* of G_x if there exists a path from each $o' \in V_x \setminus \{o\}$ to o in G_x. Moreover, we call G_x to be *ordering-complete* if either (1) V_x includes a single sink, or (2) V_x contains multiple sinks and for each pair of sinks, say o and o', the following holds: $o \rightarrow o' \in E_x$ and $o' \rightarrow o \in E_x$, i.e., all sink observations are concurrent. Note that an ordering-complete graph always has a single sink if there are no concurrent observations, while there might be multiple sinks in the case of concurrent observations. In the ordering graph in Fig. 5d, for example, there is a single sink observation, which is *o3*. Obviously, if G_x is ordering complete, the local copy of x converges to the state most recently propagated by one of the sink observations. Remember, observations observing a given object at the same time are assumed to deliver equivalent states.

The result of merging may be an ordering graph that is not ordering-complete. An example of such a graph is given in Fig 5e. An ordering graph G_x at some node can be completed due to later merging operations contributing the missing ordering information. However, it is definitely completed if this node receives an *O_Update* message of some observation in V_x. When the node maintaining the ordering graph depicted in Fig. 5e receives an *O_Update* from *o4*, it performs operations (GM1), which results in the ordering-complete graph depicted in Fig. 5f. Consequently, the following operations ensure that a node staying in the communication range of an observer currently observing x will end up with an ordering-complete G_x:

(GM3): If an ordering graph, say G_x, is not ordering-complete at a node, this node broadcasts an *O_UpdateReq* including the identifier of x. An observer receiving this message f-forwards an *O_Update* message only if it currently observes x. If no *O_Update* arrives, the node repeats (GM3) when new neighbours become available.

In addition to an ordering graph G_x, a node maintains for each observation in V_x the highest version number it has seen so far for this observation. We will use *VN(o)* to designate this version number for observation o.

4.2.2 Observer/Node Protocol

An observer broadcasts *O_Update* messages as described in Sec. 4.1.2.

When a node receives an observation record, say r, in an *O_Update*, it performs the graph management operations (GM1) described above on $G_{r.ObsId}$. Moreover, it acts upon r according to the following cases. This algorithm differs from Algorithm 1 only in case (O-2.2), for the other cases we refer to Algorithm 1.

(O-1) Object *r.Obj* is not in *DB*: Same as in Algorithm 1.
(O-2) Object *r.Obj* is in *DB*: The following subcases must be considered:
(O-2.1)*DB(r.Obj).ObsId = r.ObsId*: Same as in Algorithm 1.

(O-2.2) $DB(r.Obj).ObsId \neq r.ObsId$: According to our discussion in Sec. 4.2.1 this means that $DB(r.Obj).ObsId \rightarrow r.ObsId$. Two subcases must be considered:

(O-2.2.1) $VN(r.ObsId) < r.VN$: r replaces the old record in *DB* and is f-forwarded in an *N_Update* message to its neighbour nodes.

(O-2.2.2) $VN(r.ObsId) \geq r.VN$: r is ignored.

Note that since in this algorithm the ordering between observations is explicitly captured, the synchronization of version numbers of different observations is not needed anymore.

4.2.3 Node/Node Protocol

In this protocol, *N_Update* messages are used to propagate state information as well as information concerning the ordering of observations. In other words, an *N-Update* contains an observation record, say r, as well as the sender's ordering graph $G_{r.Obj}$. In some situations, the observation record can be omitted from those messages. To indicate this we use the notation N_Update^G.

When a node receives an ordering graph in an *N_Update* or N_Update^G message, it merges the received graph with the corresponding local ordering graph according to graph management operations (GM2) introduced in Sec. 4.2.1. In the case of an *N_Update*, it additionally acts upon received observation record, say r, according to the following cases:

(N-1) Object $r.Obj$ is not in *DB*: Same as in Algorithm 1.

(N-2) Object $r.Obj$ is in *DB*: The following subcases must be considered:

(N-2.1) $DB(r.Obj).ObsId = r.ObsId$: r and the record in *DB* are from the same observation, again two subcases must be considered:

(N-2.1.1) $DB(r.Obj).VN < r.VN$: r is younger than the record in *DB*. Therefore, r replaces the old record in *DB* and is f-forwarded in an *N_Update*.

(N-2.1.2) $DB(r.Obj).VN \geq r.VN$. r is not younger than the record in *DB* and thus is ignored. If $G_{r.Obj}$ was modified, it is f-forwarded in an N_Update^G message.

(N-2.2) $DB(r.Obj).ObsId \neq r.ObsId$: r and the record in *DB* are from different observations, two subcases must be considered:

(N-2.2.1) $DB(r.Obj).ObsId \rightarrow r.ObsId \in E_{r.Obj}$: In this case, $r.ObsId$ did not occur before $DB(r.Obj).ObsId$. Again two subcases are to be distinguished:

(N-2.2.1.1) $r.VN > VN(r.ObsId)$: r is younger than all observation records that have been seen from $r.ObsId$ before. Therefore, r replaces the old record in *DB* and is f-forwarded in an *N_Update*.

(N-2.2.1.2) $r.VN \leq VN(r.ObsId)$: r is ignored. If $G_{r.Obj}$ was modified, it is f-forwarded in an N_Update^G message.

(N-2.2.2) $DB(r.Obj).ObsId \rightarrow r.ObsId \notin E_{r.Obj}$: r cannot be accepted as potentially $r.ObsId <_x DB(r.Obj).ObsId$. In other words, accepting r could violate C3, and thus r is ignored. If $G_{r.Obj}$ was modified, it is f-forwarded in an N_Update^G message.

The algorithm described so far works fine if there are no concurrent observations. However, in the presence of concurrent observations, it cannot be guaranteed that all DB copies of an object converge to the most recent state of this object (condition C1). Since there is no ordering enforced between concurrent observations, say *o* and *o'*, a subset of copies may end up with the latest state of *o* and another subset with the latest state of *o'*. Note that this is no problem as long as both observations are active since both observe equivalent states by assumption. If, however, one ends before the other, some copies might reflect the (globally) most recent state, while others might not.

This problem can be solved by introducing an *incarnation* scheme. For this we require an observation to maintain an incarnation counter, being incremented whenever an new incarnation of this observation is created. Moreover, observation identifiers are to be extended to indicate the current incarnation. With this modification, the vertices of ordering graphs represent incarnations of observations. When a node receives an *O_Update* message of an observation incarnation, say *o.i*, after performing (GM1) it checks, whether *o.i* and some other observation, say *o'.k* are concurrent, i.e., whether edges *o.i→o'.k* and *o'.k→o.i* are in its corresponding ordering graph. If it finds such a concurrent observation, the node asks *o* to create a new incarnation and to re-broadcast *O_Update* within the new incarnation.

With regard to (GM1) and (GM2), there is no change except that vertices represent incarnations rather than observations. In other words, a new (sink) vertex is added to a node's ordering graph when an *O_Update* of a newly created incarnation arrives at this node. Note that the execution of (GM1) and (GM2) may result in an ordering graph including vertices for different incarnations of the same observation. In this case, only the vertex for the latest incarnation remains in the resulting ordering graph, all the other ones including their ingoing and outgoing edges can be removed. Consequently, as before, an ordering graph includes at most one vertex for each observation.

Assume that *o.i* and *o'.k* are the latest incarnations of observations *o* and *o'*, respectively. Assume further, that a node learns at time *t* that *o.i* and *o'.k* are concurrent. When this node receives an *O_Update* including observation record *r* from *o.i* at time *t* or later, incarnation *o.(i+1)* is created, which then re-broadcasts *r*. With the proposed scheme, it is ensured that if *o'* was finished before *t*, a record propagated by *o'* will never replace *r*.

Since ordering graphs are propagated in messages, their size is a critical parameter. In an ordering graph with *n* vertices there are *n (n-1)* edges in the worst case, where each edge can be encoded with one bit. As mentioned above, there exists at most one vertex for each observation. Moreover, a vertex can be removed from the graph if the corresponding observation is not active anymore. There are situations, where this can be decided very easily. When a node learns about a new observation, it can remove all observations performed by the same observer from its ordering graph. Remember that the observer identifier is part of the observation identifier. This limits the number of vertices to the number of observers that exist for a given object. Moreover, if a node's ordering graph includes edge *o.k→o'.i* and the node knows that the observation ranges of *o* and *o'* do not overlap, it can conclude that *o* has been finished and hence can be removed from the graph. Therefore, *n* is supposed to be a small number.

4.2.4 Correctness Arguments

Here, we will briefly argue why this algorithm ensures global observation consistency. Condition C2 is obviously fulfilled due to (O-2.1.2), (O-2-2-2), (N-2.1.2) and (N-2.2.1.2), which prevent a node from accepting an observation record that is older than the one in its local DB. Condition C3 is satisfied due to the maintenance of ordering graphs and (N-2.2.2), which prevents a node from accepting an observation record from an observation that occurred before the current DB record's observation.

To show C1 to be fulfilled, we assume that f-forwarded observation records and ordering graphs will eventually reach each node (see Sec. 3). Let us first assume that there are no concurrent observations, and *o* is the most recent observation of object *x*. Consequently, there exists only one incarnation of *o*, denoted *o.1*. Whenever a node's local ordering graph G_x is changed, the graph is forwarded by means of *N_Update* or *N_Update*^{G} messages. Therefore, each node will eventually learn about all observations of *x*. Eventually, at least one such node will be in the communication range of the observer performing *o*. Graph management operations (GM3) ensured that this node's ordering graph will become ordering-complete with sink *o.1*. Since this graph is f-forwarded in *N_Update* or *N_Update*^{G} messages, eventually all nodes will know it and thus will accept observation records of *o.1*. Our version numbering scheme makes sure that *o.1*'s latest observation record has the highest version number. Finally, this state will eventually be accepted by all nodes due to (N-2.1.1), (N-2.2.1.1), (O-2.2.1) and (O-2.1.1). Therefore, condition C1 is fulfilled if observations are sequential.

Now assume that observations *o* and *o'* are concurrent, *o'* ends before *o*, and *o* is the final observation of *x*. Moreover, we assume that *o* exists long enough that edge $o'.k \rightarrow o.i$ is in at least one ordering graph, where *o.i* and *o'.k* are current incarnations of *o* and *o'*.

If *o'* is finished before an edge $o.i \rightarrow o'.k$ can be added to any ordering graph, each ordering graph will eventually be ordering-complete with *o.i* as the only sink. Otherwise, both edges $o'.k \rightarrow o.i$ and $o.i \rightarrow o'.k$ will eventually occur in each ordering graph. Consequently, an *O_Update* messages forwarded by *o.i* will cause a new incarnation *o.(i+1)* to be created. When a node receives the first *O_Update* from *o.(i+1)*, the execution of (GM1) results in a ordering-complete graph, which includes *o.(i+1)* as the only sink. If *o'* finishes before an edge $o.(i+1) \rightarrow o'.(k+1)$ or $o.i \rightarrow o'.(k+1)$ can be added to any observation graph, each ordering graph will be eventually ordering-complete with the only sink *o.(i+1)*. Otherwise, it can be shown that each ordering graph will eventually indicate that the latest incarnations of *o* and *o'* are concurrent. This continues until *o'* is finished. Then each ordering graph will eventually be ordering-complete with the latest incarnation of *o* as the only sink. Consequently, all copies will converge to the state most recently propagated by *o*.

4 Removal of DB Entries

So far, we have only considered the creation and update of DB entries. However, the fact that objects can also "disappear" and hence must be removed from the DB has not been considered in the algorithms described above.

For the removal of objects that are not under observation anymore we adopt a soft state approach, which can be easily integrated in the above algorithms. Each DB entry is associated with a TTL. Observers are responsible to periodically refresh the TTL of the objects they observe. To do so they must make sure that the version number of an object is incremented at least once within a given TTL period and this change is propagated to the object's DB copies. Whenever a node learns about a higher version number of an object, it refreshes the TTL of the corresponding DB entry. In order to save bandwidth, the state information can be omitted from *O_Update* and *N_Update* messages if the observed state is the same as the previously propagated one.

5 Simulations

In order to determine the overhead imposed by the algorithms ensuring local and global observation consistency we compare the performance of the algorithms based on a particular f-forward primitive with the f-forward mechanism itself. The f-forward mechanism used in the simulations is plain flooding.

We consider three scenarios which model characteristic cases. The simulations are conducted in order to determine the average number of messages produced by the algorithms and the state propagation ratio *spr* reflecting the percentage of nodes that have received the observation at a time t_0. after initiating an update.

5.1 Simulation Environment

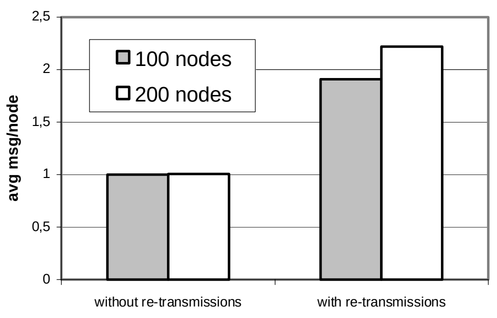
The simulations were done using a discrete time-step approach. At the MAC layer a simple collision avoidance mechanism prohibits one node to send if it is in the radio range of another node that is already sending. In this case an exponential back-off like algorithm is used to reschedule the message. If the retransmission fails for the third time the message is dropped. If both senders are out of each others radio range, simultaneous transmissions are allowed, though the message does not reach receivers in the intersection of both ranges. If two or more senders start sending simultaneously, again messages in the intersection of any two radio ranges are extinguished and do not reach their receivers. All message types are assumed to have a size of 128 bytes, the transmission speed is 128kbit/s with 250m transmission range. Mobility of the nodes is not a concern in scenario 1 and scenario 2 since times measured here are very small. In the third scenario the nodes follow the random waypoint mobility pattern [Bro+98] with a pedestrian speed of 3-5km/h.

All simulation scenarios are done on an area of 1000m x 1000m with different numbers of observers at fixed locations and either 100 or 200 nodes placed at random locations. The next sub sections present different scenarios with simulations results. All simulations have been run 25 times and were averaged.

Algorithm 1 will be referred to as *loc* (local observer consistency) and algorithm 2 as *goc* (global observer consistency). The notation of *loc-aaa x bb* or *goc-aaa x bb* refers to scenarios where either algorithm has been used with *aaa* nodes and *bb* observers.

5.2 Scenario 1: Flooding

In the flooding scenario one observer is placed in the middle of the simulation area forwarding exactly one O_Update message using either the *loc* or the *goc* algorithm. All nodes accept and forward the data when they receive it for the first time since their local DB is empty. Using either algorithm in this scenario means that an existing DB entry is never replaced since only one version number from one observer has been issued. The results in Figure 7 show the percentage of nodes that accepted the data over time for 100 and 200 nodes with either protocol. Figure 6 shows the average number of overall messages used per node receiving the data with and without repetitions caused by MAC collisions.



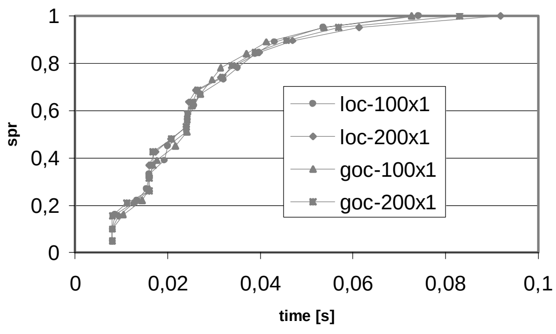


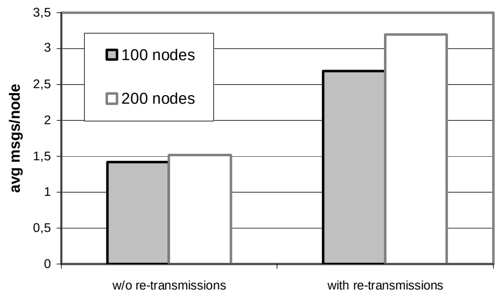
Figure 6: Average number of messages in the flooding scenario

Figure 7: *spr* over time in the flooding scenario

The figures show a very similar behaviour that is independent of the protocol used, supporting the interpretation that in all cases the algorithms are reduced to the underlying *f-forward* mechanism, which is plain flooding in this case.

5.3 Scenario 2: Concurrent Observations

In this scenario we compare the performance of both algorithms in the presence of concurrent observations with the results of scenario 1 in order to determine the cost of obtaining local and global observer consistency.



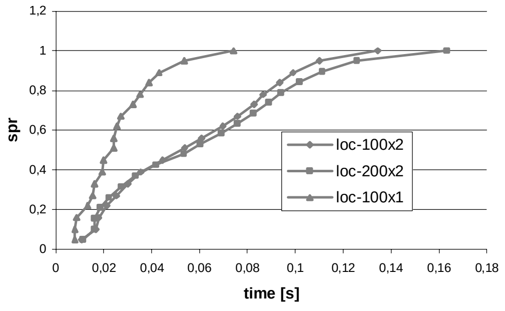


Figure 8: Average number of messages for *loc* with 2 concurrent observations

Figure 9: *spr* over time for *loc* with 2 concurrent observations compared to the flooding scenario

Figures 8 and 9 show the simulation results of the *loc* protocol in the presence of two concurrent observation that each *f-forward* the state of the same object once. Both observers use different version numbers for their observations. This causes some of the nodes to accept the state from the observer with the lower version number first, before they replace it according to the *loc* protocol with the state offered by the second observer using a higher version number. Some other nodes accept the higher version number first and do not replace it any more. The average number of messages per node in Figure 8 supports this interpretation since approximately 1.5 messages are sent by each node. Figure 9 shows a comparison of the *spr* compared to the scenario 1. The times for reaching a particular *spr* in scenario 1 are approximately 50-150% faster compared to the results when concurrent observations are present. This is caused by the higher number of messages that have to be sent in order to distribute the highest version number to nodes that accepted the lower version number first. Those nodes replace their DB entry and f-forward this a second time.

The *goc* algorithm performs in both message numbers and *spr* very similarly to scenario 1. This behaviour is due to the fact that the nodes very soon notice and distribute the concurrency of the

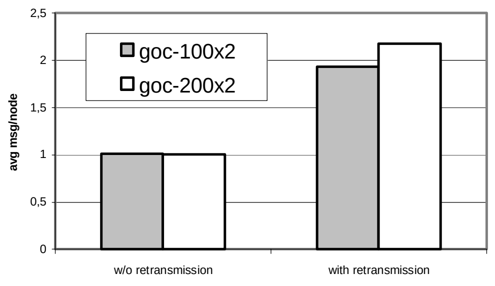


Figure 10: Average number of messages for *loc* with 2 concurrent observations

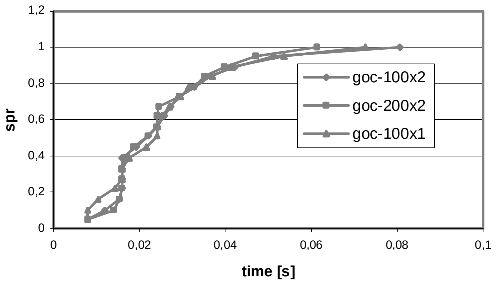


Figure 11: *spr* over time for *loc* with 2 concurrent observations compared to the flooding Scenario

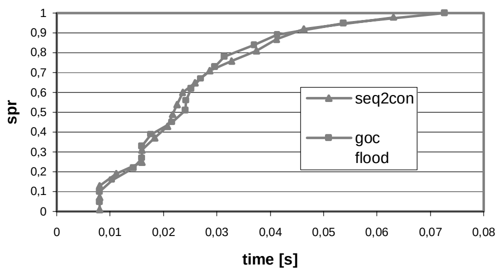


Figure 12: Transition from sequential to concurrent observations

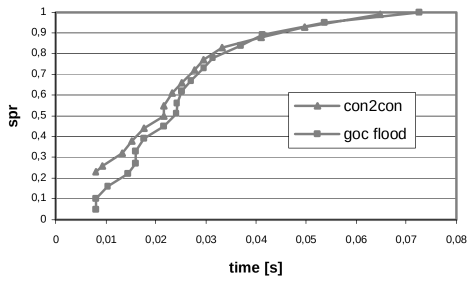


Figure 13: Transition from one state to the next in the presence of concurrent observations

two observations. This leads to the effect that nodes do not replace one object state with the other once they have accepted either one.

5.4 Scenario 3: Sequential and Concurrent Observations

This scenario determines the impact of a growing ordering graph on the spreading behaviour of the *goc* algorithm. Figure 15 shows the structure of the scenarios: it starts with two sequential observations followed by four concurrent and again two sequential observations over time. Figures 12, 13, and 14 show the information spreading at three different situations in the scenario: the transition from sequential to concurrent observations (see Figure 15, a), while observing concurrently (b), and the transition from concurrent to sequential observations (c). Every observation lasted for five seconds with an update of the version number every second.

With pauses between the sequential observations the overall duration of the scenario was about 30 seconds.

The times measured for the three situations have been normalized to compare them to the flooding scenario. 100 nodes at random positions and 8 observers at fixed positions were simulated.

The performance in all three cases is very close to the performance of the flooding scenario. In Figure 13 the concurrent scenario shows faster information spreading in around 10ms. The states of all concurrent observations are accepted as valid by the nodes due to the construction of the ordering graph whereas in the flooding scenario only one state at a time is exchanged.

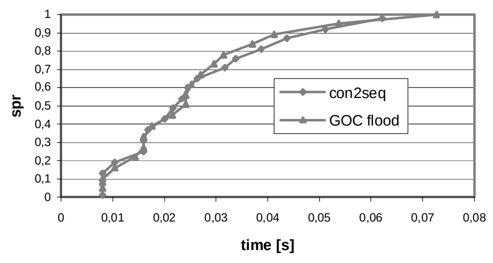


Figure 14: Transition from concurrent to sequential observations

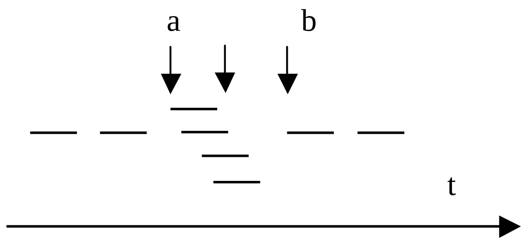


Figure 15: Observations in Scenario 3

5.5 Discussion

The conducted simulations cover realistic scenarios with respect to scenarios where observers are located near the observed object and the overall area is considerably small, i.e., 1000m x 1000m.. It is remarkable, that global observation consistency can be achieved with less message overhead and in less time than local observation consistency. This results from the additional information about the ordering of observations in the ordering graph. In the scenarios here, the ordering graph will easily fit within the simulated message size.

In future work we will ascertain the overhead imposed by larger observation graphs and the resulting message overhead as well as the state propagation ratio. Therefore we will consider scenarios in a larger area and a higher number of possibly mobile observers. Also, the impact of longer duration on the observation graphs during the simulations will be worthwhile to be investigated.

6 Related Work

A variety of flooding algorithms have been proposed to disseminate information in MANETs. Many scoping schemes have been proposed to avoid broadcast storms, e.g. in [Ni+99] and apply either probabilistic message exchange, e.g. see [Ho+99], [Ni+99], and hyper flooding approaches have been investigated to increase the packet delivery ratio, e.g. in [OTV01]. As pointed out in Sec. 3, our work builds on such mechanisms.

Strong consistency based on the concept of serializability [HR83] has been addressed in the domain of distributed databases extensively. Since, however, consistency is a trade-off to availability [DGS85], strong consistency may result in poor availability if the presence of frequent network partitioning. Therefore, weaker consistency levels have been proposed to increase the availability of data. The seminal work in [Dem+87] proposes epidemic algorithms to update copies in fixed networks. Their concept of consistency ensures that all copies converge to a common state. While this concept comes close to ours, it does not enforce state transitions to be consistent with the order the state changes has been observed in reality. However, consistency with the order of the corresponding events in reality is most important for many monitoring and tracking applications.

Data replication has been also addressed in the context of sensor networks. The adaptive broadcast replication protocol (ABR [XWC00]) covers explicitly the trade-off between consistency and transmission costs. ABR ensures weak consistency assuming that there exists a single update source per object. Deno [KC00] presents an epidemic replication algorithm based on weighted-voting for weakly connected and ad hoc networks. This algorithm ensures that each copy commits updates in the same order. However, there are no real-time constraints concerning this order.

The work reported [IGE00] also treats the diffusion of sensor data. However, it adopts an query diffusion approach rather than data replication.

For data replication in MANETs an replication algorithm has been described in [KMP99]. This work has similar objectives as ours: replicating the most current state of a data item and accepting a weaker consistency of replicas in order to increase availability under real-time conditions. However, it differs form our in the assumption that there exists a single update source for each object.

The replication protocols in [KC00], [DGS85], [Dem+87] provide means for synchronizing the copies in rejoined partitions. While our algorithms ensure that updates are applied according to our consistency constraints, we rely on the underlying (hyper) flooding mechanism to re-broadcasts states that are not available in each of the rejoined partitions. If a hyper-flooding scheme captures and re-broadcast the DB “difference” of nodes moving into each others communication range, the copies in the re-joined partitions move to the most recent state available there. However, depending on the underlying MANET and the application plain and even scoped flooding will already provide an acceptable state propagation ratio. Since our algorithms can work with any kind of flooding mechanisms they are tuneable with regard to communication cost versus state propagation ratio.

7 Conclusion and Outlook

We have addressed update propagation in mobile ad hoc networks. Updates can result from one or more observations of an object. These observations can happen concurrently. The contribution of this paper is the introduction of two consistency levels based on a generic model for update diffusion: local and global observation consistency. Based on forwarding primitive, f_forward, two algorithms were presented to ensure the corresponding consistency levels. Beside informal correctness proofs, simulations compare the algorithms with plain flooding as the underlying f_forward mechanism. While local observation consistency can be achieved without storing data structures besides the updated information and its source global observation consistency requires additional ordering information about observations. Compared to the underlying flooding, local observation consistency results in a 50% increase of messages and a doubled latency. Global observation consistency performs the same compared to the underlying flooding in both, latency and message overhead but requires the maintenance of the ordering graph.

Our current implementation of the f_forward primitive is based on plain flooding. Due to the performance of Hyper-Flooding or gossiping compared to plain flooding it is worthwhile to compare the resulting message overhead and latencies when the f_forward mechanism is changed. Additionally, the size of the ordering graph in different realistic and long-term scenarios will be investigated. Further optimisations in the graph storage and management should be based on characteristic occurrences.

References

[BHG87] Bernstein, P. A., Hadzilacos, V., Goodman, N.: "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987

[Bro+98] Broch, J., Maltz, D., Johnson, D., Hu, Y.-C., Jetcheva, J.: "A performance comparison of multihop wireless ad hoc network routing protocols", Proceedings of the Fourth International Conference on Mobile Computing and Networking (MobiCom), Dallas, Texas, 1998

[Dem+87] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: „Epidemic Algorithms for Replicated Database Maintenace", Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, 1987

[DGS85] Davidson, S. B., Garcia-Molina, H., Skeen, D.: "Consistency in Partitioned Networks", ACM Computing Surveys, Vol. 17, No. 3, September 1995

[HKB99] Rabiner Heinzelman, W., Kulik, J., Balakrishnan, H.: "Adaptive Protocols for Information Dissemination in Wireless Sensor Networks", Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking (MobiCom'99), Seattle, Washington, USA, 1999

[Ho+99] Ho, C., Obraczka, K., Tsudik, G., Viswanath, K.: "Flooding for Reliable Multicast in Multi-Hop Ah Hoc Networks", Workshop on Discrete Algorithms and Methods for Mobility at the Fifth Annual International Conference on Mobile Computing and Networking (MobiCom'99), Seattle, Washington, USA, 1999

[HR83] Härder, T., Reuter, A.: „Principles of Transaction-Oriented Database Recovery", Computing Surveys, Vol. 15, No. 4, pp. 287-317, 1983

[IGE00] Intanagonwiwat, C., Govindan, R., Estrin, D.: "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks", Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (Mobicom 2000), August 2000, Boston, MA.

[KC00] Keleher, P. J., Cetintemel, U.: „Consistency management in Deno", Mobile Networks and Applications, 5(2000) pp. 299-309, Baltzer

[KMP99] Karumanchi, G., Muralidharan, S., Prakash, R.: "Information Dissemination in Partitionable Mobile Ad Hoc Networks", Proceedings of 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Switzerland, 1999

[Nee93] Needham, R. M.: “Names”, in Distributed Systems, Ed. S. Mullender, 2nd Edition, Addison Wesley, 1993

[Ni+99] Ni, S.-Y, Tseng, Y.-C, Chen, Y.-S., Sheu, J.-P.: “The Broadcast Storm Problem in a Mobile Ad Hoc Network“, Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking (MobiCom'99), Seattle, Washington, USA, 1999

[OTV01] Obraczka, K., Tsudik, G., Viswanath, K.: “Pushing the Limits of Multicast in Ad Hoc Networks”, Proceedings of International Conference on Distributed Computing Systems (ICDCS), Phoenix, USA, 2001

[Sat93] Satyanarayanan, M.: “Distributed File Systems”, in Distributed Systems, Ed. S. Mullender, 2nd Edition, Addison Wesley, 1993

[XWC00] Xu, B., Wolfson, O., Chamberlain, S.: „Spatially Distributed Databases on Sensors“, Proceedings of the 8th ACM Symposium on Advances in Geographic Information Systems (ACMGIS’00), Washington, DC, USA