Consistent Context Management in Mobile Ad Hoc Networks

Jörg Hähner, Christian Becker, Pedro José Marrón
Institute of Parallel and Distributed Systems (IPVS)
Universität Stuttgart, Germany

Abstract: John entered his office. John left his office. The order of these updates to a context-aware system is important to reflect the state in the real world. Context information obtained by sensor systems requires consistency concepts which reflect the chronological ordering in which context information has been captured. This paper introduces a consistency concept which allows to express the ordering of events which happened outside of a computer system.

1 Introduction

Context-aware applications rely on the state of the physical world and react accordingly in order to adapt to the preferences of users in their current situation. Examples are navigation systems adapting their presentation to the media a user is travelling on, e.g. as pedestrian, car driver, or on a public transport. Cell-phones which determine their context, i.e. being in a meeting based on the noise and light conditions in the surroundings, can forward incoming calls to a mailbox in order not to interrupt the meeting. Sensors in a storehouse can provide current information about the goods and their location allowing for an up-to-date inventory. All these examples have in common that the applications have to rely on a number of data fetched by sensor systems. The information may be collected by autonomous sensor nodes as well as there may be no centralized storage of the context information. Consistency concepts which ensure the logical order of events, such as Lamport's logical clocks, cannot cope with events which are created outside of the computer system, such as observations made by sensors. Especially applications in mobile ad hoc networks are additionally challenged by frequent and unpredictable network partitions. As a result, updates of the sensor information can be delayed and may overwrite more current ones. In cases where applications have to rely on the most current information or a history, e.g. to determine the track of an object, this will lead to wrong interpretations.

This paper presents a novel consistency concept that ensures the ordering of events based on the chronological order in which they have been observed. An algorithm ensuring this consistency concept in mobile ad hoc networks is sketched. This provides applications the capability to react upon the order of events happened in the physical world. Important examples are the determination of the direction of a mobile object based on the time of two sightings, e.g. moving from observation place 1 to observation place 2 or in the opposite direction. More generally, the consistency concept ensures, that no older observation overwrites a more current one. The paper is structured as follows. Next, we discuss related work. In Section 3 we introduce our system and consistency model followed by a discussion of how replication algorithms that guarantee the consistency model may be designed. The paper closes with a brief summary.

2 Related Work

The ordering of events in distributed systems has been subject of research for many decades. The seminal work of Lamport [La78] introduces means for ordering events that can be specified and observed *within* a given system. The *happened-before* relation as defined by Lamport utilizes the order between sending and receiving of a given message at different processes in the system by merging the causal history of the sender and receiver processes at the receiver. Our consistency concept enforces the chronological ordering of update operations caused by events *external* to the system rather than their causal ordering within the system. Strong consistency based on the concept of serializability [HR83] has been addressed in the domain of distributed databases extensively. Since the level of consistency is a trade-off to availability [DGMS85], strong consistency may result in poor availability in the presence of frequent node and network failures. Weaker consistency levels have been proposed to increase the availability of data. The authors of [DGH$^+$87] propose epidemic algorithms to update copies of replicated data in fixed networks. Their concept of consistency ensures that all copies converge to a common state. The Deno system [KC00] implements an epidemic replication algorithm based on weighted-voting ensuring that each copy commits updates in the same order. The authors of [LHE03] present a collection of protocols for probabilistic quorum based data storage in MANETs. Read operations will return the result of the latest update operation independent of the order in which these updates have been executed. Both, the work in [KC00] and [LHE03] do not guarantee that updates are applied in chronological order.

3 Consistency Model

Our system is comprised of a set of mobile or stationary nodes which are networked in an ad hoc manner using wireless communication technology. Each node may be in one or more of the following roles: observer, database node (DB node), or client. An *observer* is equipped with sensors that are used to observe the state of *perceivable objects* in its surroundings. Whenever such an observer senses a significant state change of an object it is responsible for creating a so-called *update request* which is then sent to one or more DB nodes. The *DB nodes* maintain a database that stores one state record for each perceivable object in the system. The database may be distributed over the set of DB nodes in different ways. Each DB node may, for example, hold a copy of every state record to ensure high availability. Applications run on *clients* which only read state records. This restriction makes sense, because each state record reflects the state of a real-world object, which cannot be altered if no actuators are used in the system. A single physical node in the network may be in several roles at a time, e.g., as a client and a DB node.

3.1 Ordering of Update Operations

As observers are used to capture the state of real-world objects, the chronological ordering of update requests created by observers plays a crucial role for applications to reason about the state of an object. An application may, for example, need to determine the direction of movement of an object based on a series of location changes. Due to the lack of global time in distributed systems, the ordering of state changes perceived by different observers may only be done with limited accuracy. Therefore, we define the *occurred-before* relation for two update requests as follows.

Definition 1 (occurred-before): Let u and u' be two update requests. Then u occurred-before ($<$) u' iff $\mathrm{t_{obs}}(u') - \mathrm{t_{obs}}(u) > \delta$, where $\delta > 0$ and $\mathrm{t_{obs}}(u)$ denotes the real time at which the observation leading to the generation of u occurred.

In the definition, parameter δ describes how accurate update requests can be ordered. If two update requests are created by two different observers with a temporal difference of less than δ, their order cannot be determined and these update requests are said to be *concurrent*. If an algorithm relies on synchronized clocks, for example, δ would be determined by the accuracy of the clock synchronization algorithm used.

3.2 Consistency Definition: Update-Linearizability

Update-linearizability is a weak consistency model which ensures that clients never read a state for a logical object that is older than any other state the same client has previously read for the same object. The update requests for each logical object are ordered according to the *occurred-before* relation presented in the previous section. The following definition of *update-linearizability* captures the idea that all operations, both read operations done by clients and update requests executed by observers, may be serialized against a single logical image of the database.

Definition 2 (update-linearizability): An execution of the read and update operations issued by clients and observers is said to be update-linearizable if there exists some serialization S of this execution that satisfies the following conditions:
(C1) All read operations of a single client on a single object in S are ordered according to the program order of the client. For each object x and each pair of update requests $u[x]$ and $u'[x]$ on x in S: $u'[x]$ is a (direct or indirect) successor of $u[x]$ in S iff $u[x] < u'[x]$ or $u[x] \| u'[x]$.
(C2) For each object x in the database S meets the specification of a single copy of x.

a)	b)	c)
O1: $u[x]1$	O1: $u[x]1$	O1: $u[x]1$ $u[y]2$
O2: $u[x]2$	O2: $u[x]2$	O2: $u[y]1$ $u[x]2$
C1: $r[x]1$ $r[x]2$	C1: $r[x]1$ $r[x]2$	C1: $r[x]2$ $r[y]1$
C2: $r[x]2$	C2: $r[x]2$ $r[x]1$	C2: $r[y]2$ $r[x]1$

Figure 1: Sample executions: execution (a) and (c) are valid, while (b) is invalid

3.3 Examples of Executions

Figure 1 gives three examples for valid and invalid executions according to Definition 2. We use the notation $u[x]1$ to indicate an update request for object x that writes the state 1 and $r[y]2$ for a read operation that reads object y and returns state 2. The time axis runs from left to right.

The execution in Figure 1(a) is correct because client C1 reads the state of object x as 1 even though state 2 has already been written by observer $O2$. This is allowed because $C1$ has never read object x before, allowing to start $C1$'s program for object x anywhere in the serialization. Client $C2$ reads state 2 at the same time than $C1$ reads state 1. This is valid because executions of different clients may be interleaved in the serialization. In contrast to that, Figure 1(b) shows an invalid execution, because $C2$ reads state 1, which is the older state information, after it has already read state 2.

The example in Figure 1(c) is a valid execution with two objects, because update-linearizability is an object-local property and both clients read each object only once. $C1$ reads state 2 of object x before reading state 1 of object y and $C2$ vice versa.

3.4 Using the Consistency Model

From the perspective of a programmer the concept of update-linearizability is slightly different from sequential programming. Consider a program that monitors the state of an object, e.g. the temperature of an object. The task of the program is to send a notification to another process if a threshold is exceeded. Let $state_{below}$ be a value below and $state_{above}$ a value above a given threshold. Definition 2 guarantees that if the monitoring process reads a sequence $state_{below}$ then $state_{above}$ there was a state change from below the threshold to above the threshold. The same holds respectively for reading $state_{above}$ first and then $state_{below}$. This means that the monitored state of the object crossed the threshold in the observed direction, if the monitoring process sends a notification. In general, the opposite implication, i.e. if a notification is sent exactly one state transition has been observed, depends on how fast updates are being propagated to client processes compared to the update frequency. Computations that involve reading multiple objects may be regarded as concurrent clients where one object is read by each client, because update-linearizability is an local property for each object.

3.5 Implementation Issues

A multitude of algorithms that implement *update-linearizability* is possible. The spectrum of such algorithms includes different degrees of data replication and may function with or without the use of synchronized clocks. Depending on the algorithm, state information may be fully or partially replicated on a set of DB nodes. Clients and DB nodes may be co-located on the same device for high read availability. To achieve chronological ordering without the use of synchronized clocks, the according algorithms need to maintain additional ordering information, such as chronological ordering graphs.

4 Summary

Context-aware applications often need to reason about the state of real-world objects. One key aspect here is the order in which state changes of such objects occur. Therefore, we presented *update-linearizability*, a novel consistency model that takes into account the *chronological* ordering of update operations as they are created by so-called observers. Observers are simple devices that are equipped with appropriate sensors that allow for identifying objects and sensing their state. State changes of objects are propagated to DB nodes which maintain a distributed database of the most recent state of all objects in the system. Clients run context-aware applications that read the state of real-world objects in order to adapt themselves to changes in their environment.

References

[DGH+87] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., und Terry, D.: Epidemic algorithms for replicated database maintenance. In: *Proc. of the 6th Symposium on Principles of Distributed Computing*. S. 1–12. 1987.

[DGMS85] Davidson, S. B., Garcia-Molina, H., und Skeen, D.: Consistency in a partitioned network: A survey. *ACM Computing Surveys (CSUR)*. 17(3):341–370. 1985.

[HR83] Haerder, T. und Reuter, A.: Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*. 15(4):287–317. 1983.

[KC00] Keleher, P. J. und Cetintemel, U.: Consistency management in deno. *Mobile Networks and Applications*. 5(4):299–309. 2000.

[La78] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*. 21(7):558–565. 1978.

[LHE03] Luo, J., Hubaux, J.-P., und Eugster, P. T.: Pan: Providing reliable storage in mobile ad hoc networks with probabilistic quorum systems. In: *Proc. of the 4th ACM Int. symposium on Mobile Ad Hoc Networking and Computing*. S. 1–12. 2003.