Using a Policy Language to Control Tuple-space Synchronization in a Mobile Environment

Vorapol Jittamas and Peter F. Linington
University of Kent Computing Laboratory
Canterbury, Kent, UK
vj7@kent.ac.uk, pfl@kent.ac.uk

Abstract

Any sharing of information using a distributed platform carries the risk of disconnection because of loss of network access. This is particularly the case when considering mobile communication, either using base stations or by forming ad-hoc networks. Replication of shared data is one way to increase data availability in such an environment, but leads to the problem of inconsistency between copies of data, and so requires some means of data synchronization.

This paper investigates how policies can be used to resolve data conflict in a way that can be tailored to meet the needs of different types of application in different situations. It discusses a range of application requirements, and describes a policy-based pervasive middleware to support the sharing of data using a tuple-space paradigm. Policies maintained within the middleware are used to trigger a wide range of synchronization options to restore the consistency of the data after periods of disconnected operation.

1. Introduction

Distributed applications are needed in a mobile environment both to allow interaction with fixed services and to allow direct communication between mobile peers. A shared tuple space provides a model for communication which is simple for the application designer to work with, but which results in apparent continuity for the end-user. Such a model can then be supported by middleware that handles communications problems, striking a balance between continuity of operation and consistency between distributed data copies.

In many situations, such as coordination of diaries, continuity of operation is more important than absolute consistency. However, isolated copies will gradually diverge as changes are made, and some form of compensation is essential. A synchronization process provides this. A set of rules is needed that re-establishes consistency while providing users with a predictable view of the process. One solution is to use a policy language to express these rules so that a manager can create policies causing the middleware to react to inconsistency in ways that meet user expectations. The aim is to increase the probability that synchronization is handled automatically by the middleware. The first step in designing a policy-based solution is to extract requirements from applications and so establish the constraints needed and the information they will reference.

This paper investigates the requirements for synchronization policies in a range of applications. It also presents a tuple-space-based mobile middleware called JavaSpace for Mobile Environment (JSFM), built at the University of Kent to experiment with synchronization policies.

2. Background and related work

The tuple-space paradigm was first introduced by the Linda language [5]. It defines a body of storage called a space and the groups of objects put into and retrieved from the space are called tuples. It is attractive because of its simplicity and its ability to provide time and space decoupling; many mobile middlewares. such as LIME [11], TOTA [9], and TIAMAT [10], have used this paradigm. However, these systems do not store local replicas of remote information; it is left to an application to make local copies explicitly. The middleware proposed here automates this process and manages synchronization when needed.

There are also many systems providing data synchronization at a file level; AdhocFS [3] is a file system for use in a mobile ad-hoc environment, which uses data replication to make data available within a group of devices. The Bayou replicated storage system [13] also provides conflict resolution during data synchronization in a mobile environment. Clients store data on servers and each server is responsible for propagating data and resolving conflicts between data items. In this design, users provide, along with the data, a definition of how conflict will be detected and resolved. In Coda [7], file caching is used to provide information to a user whenever the system is disconnected. The Coda file system tries to avoid user intervention during conflict reconciliation by using a file of rules to control the system behaviour.

Instead of trying to reconcile conflict, Mobisnap [12] prevents conflict from happening between different mobile devices by using reservation. Although this avoids the synchronization problem, it does not support the continuous disconnected operation we envisage here.

3. Application Requirements

Each application requires its data to be synchronized whenever nodes are connected, and requires a policy to resolve any conflicts, but the rules for conflict resolution vary. Review of a number of applications identifies requirements.

Basu and Little [1] describe an ad hoc network formed by parking meters and cars wishing to park. Each parking meter writes a tuple to a space to represent an individual parking space that it is responsible for, and this tuple will be propagated through each connected car and parking meter. A car that decides to park in a specific parking space edits the corresponding tuple so as to reserve it.

Conflict can occur if two cars try to reserve the same parking space while not connected to each other, and policies are then used to determine the outcome. The simplest policy would be to give priority to the car that reserved a parking space first. Taking the earliest change is a resolution policy commonly used when there is only limited information about the history of activities. However, extra context data, such as a user's role, allows priority in reservation within a particular set of spaces.

Other applications have different context requirements during synchronization. Examples include factory automation, shared devices in an office, warehouse inventory control systems, a number of which are reviewed in [4].

Most applications can be classified by the number and type of contextual data items involved in synchronization – what matters is the richness of the object model used to support the policy definition.

The survey also revealed a number of applications in which it was hard to find a separate synchronization policy. One example is the SpotMe:People Radar(www.spotme.com) which reminds users at a conference that they want to talk to someone they find themselves close to. Here, the application is itself effectively just a synchronization process.

In the original work on tuple spaces, a space is just an associative store, and there is no independent concept of tuple identity. If a tuple is removed with a take operation, then updated and written to the store, the result is a distinct tuple. To base synchronization on the history of a tuple during a series of changes, we need a stronger concept of identity. This is constructed from a group of fields in the tuple, which together make it unique, and can be used by the middleware to track a tuple as its state evolves. The tuple identity is based on the query processing performed when searching for a tuple. Retrieval by identity is then essentially retrieve by matching selected fields. In this way, the middleware can track each tuple, even though it may have been deleted and then written back into the space, and so can detect conflicts when tuples with the same identity in different spaces differ in their other fields.

The results of this survey have been used to create a library of scenarios, each defining a starting state and an assumed sequence of user actions, network status changes and subsequent synchronization events and stating the outcomes a user expects. The scenarios can be categorized using the supporting information needed.

A proposed policy is then assessed rapidly with a harness in which the policy control components from the real middleware are used to execute each scenario in turn, determining the state changes selected by the policy at each of the synchronization events. The resulting state is compared to the set of outcomes in the scenario to see if the policy is successful.

However, the final test of a system must be based on real experience to ensure that no factors have been overlooked. For this the complete prototype implementation of the mobile middleware is used, controlled by the same components that have already been tested with the scenarios.

The selection of a suitable synchronization policies in this way involves a balance between expressiveness and simplicity; including more factors about the environment, the history of events and the application can lead to better-informed decisions, but at the expense of increased cost. The approach outlined here allows us to to compare the benefits of different control policies.

4. The JavaSpace for Mobile Environment

The aim of JSFM is to provide an application environment that is as close as possible to Sun's traditional JavaSpace, so that existing applications can be transferred directly. It forms a value-added layer, handling problems of distribution and synchronization above a slightly enhanced version of the earlier JavaSpace system. Internally, the JSFM architecture has three main components, shown in figure 1:

- the Port-to-Space (PTS), which links to the application, and maps items in the virtual space to those in the local space.
- the Policy Engine (PE), which is the heart of the middleware, takes all the synchronization decisions, based

on policies it manages. Synchronization is initiated by an event representing a change, either in a local space or in the network connectivity.

- the Multicast Layer (ML), which maintains the peer-to-peer configuration between spaces in the form of a multicast tree, based on AMRoute [2], modified to allow the use of heterogeneous networking environments. This tree is used to distribute events.

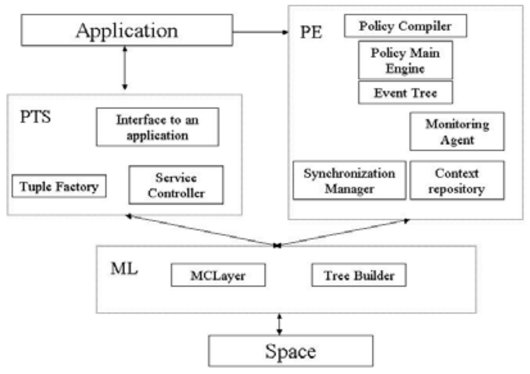

Figure 1. The JSFM architecture.

Synchronization is performed in three steps. First, the synchronization type is determined by matching the properties of the current event with an entry in an Event Type Tree. Then the Policy Main Engine loads policies matching the event type, checks the policy constraints, selects policies that apply and use them to attempt synchronization. First matching by event type simplifies the policy definitions by isolating the individual cases.

5. The Policy Language

The language used in the JSFM Policy Engine needs to allow a policy to reference the current and historical state of a space and its tuples during synchronization. It can be divided into two sub-languages.

The first language defines the tree of event types (based on subtyping) used to determine the policies that apply. Basically, event types are defined by assigning constraints on predefined event attributes to each tree node; examples are the number of times devices have been connected to each other, or the existence of corresponding identities in local and a remote spaces. OCL is used as a type definition language because most users will already be familiar with it.

The second sub-language defines the policies used to invoke synchronization actions when events occur. At present, we have adopted the Ponder language [8] to express the policies for controlling the synchronization process, rather than generate a new language needlessly. The synchronization constraints are expressed in Ponder as obligation policies triggered by events generated by the discovery of inconsistent data. OCL constraints are heavily used in the matching process and to define dynamic source and target domains from the corresponding static types.

Apart from basic facts like the time at which an event takes place, there are many other pieces of information used for determining the synchronization action. For instance, the number of times that a tuple has been accessed while a device was disconnected can be used, because it indicates that the application has used it for taking decisions.

OCL constraints are used to access such contextual information, and also other types, such as type of device (mobile or stationary), or device parameters (bandwidth or remaining battery power). Several kinds of context information can be applied in policies; for example, the time at which a synchronization process takes place can be used to change the priority a policy gives to different spaces. Policy constraints can also refer to information in the tuple itself, and this can be used in prioritizing the changes in different spaces. Actions are currently taken from a pre-defined set allowing, for example, copying or deleting of tuples, but extension to application-specific actions is planned.

In more complex situations, the policy may need to allow a user to assign a priority to each constraint. This could take the form of a score or utility function, so that a score derived from the constraints is used to determine the best course of action, as in the policies in [6]. However, this just transfers the problem of reflecting the designer's intentions correctly to the choice of utility function.

Consider an example where a device from a virtual space becomes disconnected. While disconnected, the device makes a series of five edits to a tuple (take/write sequences), and the other participants in the virtual space perform three editing sequences. Synchronization is clearly needed on reconnection, and the policies in figure 2 might apply. Both policies are triggered by the same type of synchronization event, but their targets are different. This could be because the application uses a number of different tuple-types, and requires a different synchronization behaviour for each.

The first policy is applied to a tuple of type ”TypeA”; it favours a tuple accessed several times, because its value has had more impact on the application's state. Changing a more frequently accessed tuple will potentially invalidate more derived information than changing a tuple with a lower access count.

On the other hand, the policy for a tuple with type ”TypeB” is based on the time since the tuple was last modified. The freshest tuple is used because it is the most up-to-date. If a target satisfies both these types, there is a conflict to be resolved at a higher level.

Sometimes a disconnection may disrupt the synchronization process. If each tuple is processed independently, any

```
inst oblig pol1 {
    on              Normal_Sync_Both_Side;
    subject ⟨space⟩ s = /space;
    target ⟨tuple⟩  t =/tuple→select(t1|t1.tupleType =
                        ”TypeA”);
    do              s.propagate(t);
    when            /space→exists(s1,s2|s1.isLocal = true
                        and s2.isRemote = true
                        and s1.accessCount(t) >
                                    s2.accessCount(t));
}

inst oblig pol2 {
    on              Normal_Sync_Both_Side;
    subject ⟨space⟩ s = /space;
    target ⟨tuple⟩  t = /tuple→select(t1|t1.tupleType =
                        ”TypeB”);
    do              s.propagate(t);
    when            /tuple→exists(t1,t2|t1.localtuple = true
                        and t2.remotetuple = true
                        and t1.identity = t2.identity
                        and t1.time_lastmod >
                                    t2.time_lastmod);
}
```

Figure 2. Example of a more complex policy.

incomplete attempt can simply be abandoned; synchronization will be retried when the connection is resumed. However, if a more complex policy tries to maintain the consistency of groups of tuples, a transactional structure is needed, and an incomplete synchronization may need to be resumed until the group is complete or is rolled back.

However, as long as the synchronization process is composed of a number of independent events, a sequence of disconnections will still eventually lead to correct synchronization, if some of the periods of connection are long enough to complete the processing of some tuples, and the rate of modification is low enough for there to be a gradual reduction in the work remaining.

6. Conclusions and future work

This paper has shown the use of a policy language for controlling synchronization processes in a mobile environment, both using working mobile middleware and in a test framework that allows a much broader range of policies to be investigated. The policy compiler used in the implementation is based on the compiler provided by the Ponder group, modified to suit our experimental environment.

The work is still ongoing, but experience so far has shown that selection of suitable policies can support a wide range of applications. The library of scenarios is being extended to support a wider range of application types.

In the next phase of this work we are investigating the problems of stability where policies in the various nodes are being updated during use. At present, the project assumes that all devices will use a similar policy, but, in real life, users need to edit their policies at run-time, which leads to the creation of conflicts, since policies in different nodes may cause different actions during synchronization. One way to resolve this is to define a meta-policy that can constrain the local policies, and hence limit the actions that are going to take effect.

References

[1] P. Basu and T. Little. Networked parking spaces: Architecture and applications. In *Proc. 56th IEEE Vehicular Technology Conference*, volume 2, pages 1153–1157, 2002.

[2] E. Bommaiah et al. *AMRoute: Adhoc Multicast Routing Protocol*. Internet Engineering Taskforce, INTERNET-DRAFT, 1998. http://www.ietf.org/proceedings/98dec/I-D/draft-talpade-manet-amroute-00.txt.

[3] M. Boulkenafed et al. AdHocFS: Sharing files in WLANs. In *Proc. 2nd Int. Symposium on Network Computing and Applications*, Apr. 2003.

[4] I. Chlamtac, M. Conti, and J. Liu. Mobile ad hoc networking: Imperatives and challenges. *Ad Hoc Network Journal*, 1(1), Jan.-Feb.-Mar. 2003.

[5] D. Gerlernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1), Jan. 1985.

[6] J. Kephart and W. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proc. 5th International Workshop on Policies for Distributed Systems and Networks*, 2004.

[7] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *13th ACM Symposium on Operating Systems Principles*, 1992.

[8] E. Lupu et al. Ponder: Realising enterprise viewpoint concepts. In *Proc. 4th Int. Enterprise Distributed Object Computing (EDOC)*, Mukahari, Japan, 2000.

[9] M. Mamei et al. Tuples on the air: a middleware for context-aware computing in dynamic networks. In *Proc. 2nd Int. Workshop on Mobile Computing Middleware; at 23rd ICDCS*, pages 342–347, Providence (RI), USA, May 2003. IEEE CS Press.

[10] G. McSorley et al. Tiamat: Generative communication in a changing world. In *1st Int. Workshop on Middleware for Pervasive and Ad Hoc Computing (MPAC), Middleware*, 2003.

[11] P. Picco et al. Lime: Linda meets mobility. In *Proc. 21st Int. Conf. on Software Engineering*, May 1999.

[12] N. Preguica et al. Reservation for conflict avoidance in a mobile database system. In *Proc. 1st Int. Conf. on Mobile Systems, Application, and Services (MobiSys)*, 2003.

[13] D. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th Symposium on Operating Systems Principles (SOSP-15)*, Copper Moutain, Colorado, Dec. 1995.