Using Structural Characteristics for Autonomous Operation

Carlos Baquero, Francisco Moura
Distributed Systems Group
Universidade do Minho, Portugal
http://gsd.di.uminho.pt/

Keywords: Mobile Computing, Replication, Conflict Resolution

1 Introduction

The majority of current mobile computing systems operate either in conjunction with a central network by some form of weak connectivity or tend to operate in total isolation and perform sporadic synchronization with a backup or a central network. These configurations miss an additional and very useful pattern of operation – mobile to mobile interaction. Recent mobile devices have the capacity for direct communication among them, but this option is essentially neglected by the application software.

In order to address this pattern of operation we believe that there is a need to support re-usable peer-to-peer synchronization mechanisms that both respects data ownership and enables some level of state reconciliation.

Naming this operation pattern as *autonomous operation*, we can observe that this pattern is already found on many legacy applications deployed in distributed systems. For example, personal information managers, Mail/News readers and Web browsers, often store persistent state in local files, but tacitly assume a single copy. Noticing that these separate copies are in fact replicas of a distributed entity, leads to the creation of semantically knowledgeable file synchronizers that strive to restore an unified state from these replicas.

Evolution from static distributed systems to mobile platforms raises a demand for applications that, not only are adapted to user mobility but, take advantage of it. It is clear that despite continuous improvements on connectivity support for mobile environments, the cost and coverage limits still imply a major share of disconnected operation. When connectivity does exist it usually interposes wide area networks between communication peers, when one party is on the road, leading to lower channel quality. On the other hand, user mobility is likely to conduce to, normally unforeseen, physical proximity of the user's mobile computer with other mobile or fixed systems. This occurrence is likely to increase as the installed population of mobile devices increases.

In this work we show that without imposing restrictions on availability, which is a crucial factor for personal applications, it is possible to enable some data sharing among autonomous mobile applications. This sharing would take advantage of any pairwise encounters of replica holders.

To determine the level of sharing that is compatible with permanent availability, we model general purpose data types that provide the necessary reconciliation guarantees. These guarantees are obtained by placing restrictions on the allowed behavior in order to avoid the occurrence of conflicting concurrent operations that would prevent reconciliations. Among other uses, these data types should help to identify sharable segments of data on classes of applications that traditionally support no sharing at all, and identify which parts of the state can be effectively shared.

In the next section we present some examples of sharable data that motivates the modeling of a more generic and higher level description. This description is presented in the third section together with a framework of convergent components. Section four builds on this framework and gives a general presentation of a Java implementation for a component hierarchy. Before presenting the conclusions we show how these tools where used to build a merger for pairs of bookmark files, giving some insight on how to combine the components to create a concrete application.

2 Cases of structurally convergent data

The identification of structurally convergent data, for future use in autonomous mobile applications, can be pursued in classical distributed settings where instances of the same application are used to make independent updates on the same logical data item. A common example is found when working with mail readers on more than a single machine: A given user who receives is mail on a fixed workstation, might also want to keep the mail in is portable machine, and for that, regularly fetches mail by way of an appropriate protocol. Once this user starts sending mail on both machines two replicas of the same logical entity, the sent mail folder, are created. A reconciliation for this replicated data can be obtained by looking at these folders as sets of messages, and observing that each set is in fact a grow-only set. In this case, convergence is directly obtainable by set union, which is the appropriate reconciliation operation for grow-only sets. A further, and more complete, interpretation would pay attention to the order of set elements (messages), and might specify a partially ordered grow-only structure that would track the known dependencies among sent messages.

Other examples drawn from mail management can be found on the tracking of deleted (or read) messages, when mail is handled on multiple locations. Groups of messages that have been read or deleted can also be interpreted under the abstraction of grow-only sets. In the case of deleted messages this would require storing additional information on deletions.

Through these examples we find, legacy data structures that can support reconciliation under appropriate tools, and also cases where adding extra information to these legacy structures would support a richer reconciliation policy.

Research on distributed file systems with optimistic replication [5, 10, 7, 11] raised several examples of semantically knowledgeable file mergers, that strive to restore consistency after concurrent changes. Files such as `.newsrc` (see Figure 1) already hold the necessary information for ensuring convergence of segments of read articles for each inspected news group. In this case, we can abstract it as sets of segments (possibly including singular segments). This representation induces straightforward convergence once a suitable merge is defined for overlapping or contiguous segments.

```
alt.elvis.sighting:33,45,60-200,356
alt.emulators.ibmpc.apple2:
comp.object: 1-2628
```

Figure 1: A small `.newsrc` file

Another of these examples shows how the intrinsic order of some data elements together with the selection of a sized subset can be used to apply a convergence procedure. This example deals with the reconciliation of two files with top scores in an arcade game. From the perceived semantics of this top scores file it follows that convergence can be achieved by computing an ordered merge of the two sequences and selecting the largest k elements. The very same procedure can be applied to other cases, that deal with sharable personal information, such as: Keeping information on the 5 credit institutions that provide the best loan taxes; or where can be found the best deals for buying flashram cards.

WWW is also a good source of examples for structurally convergent data formats. For instance, many users cannot avoid adding bookmarks on browsers running on different machines with unshared file systems. If these bookmarks files only allow folders at the root level, they can be treated as growing sets of folders and URLs, in which the folders hold growing sets of URLs[1]. In the case of Netscape bookmark files (for which a merger will be presented in section 4.3), folders can be created at any level, leading to generalized composition with arbitrary levels. A similar model is also found in the structuring of the Usenet News group structure.

3 Convergent components

In this work the identification of structurally convergent data is done together with a formal description

[1]We omit, at this moment, the existence of delete operations that remove URLs. This change of behavior will be properly analyzed on section 4.3.

of elementary data types and composition rules. We will not stress in this article the formal presentation of the model, which is available in another document [1], but we present a description of the environment, enumerate some of the data types, and exemplify the formal description with a sample component.

3.1 Generic components

We now give an informal description of some important components that have been identified thus far.

Constant information is a trivially convergent data type which, nevertheless, plays an important role as it is a common component on composite structures. Its use on the examples on section 4 shows its role on the description of composites.

Segments of immutable information can be treated as single components or combined on special components that rule the creation of correct aggregations of components.

Sets with add-only information are also very common, with the corresponding merge being modeled by set union. Additional properties, like elements with embedded order or other cases, such as elements that denote ranges, imply different merge operations that involve an adequate adaptation of the set union operation.

Dual to grow-only sets, is the notion of sets with deletion-only. The corresponding merge operation is done by keeping common elements. Interestingly, this component can also be modeled by a constant element with the initial set contents and a grow-only set of deletions.

A richer behavior is found on sets that can grow and shrink, such as mailboxes that receive mail and have mail deleted. Those can be modeled by a pair of grow-only sets that describe inserted and deleted messages, and by a proper merge procedure that handles these two sets. This later case, that combines two components, calls for one of the composition rules that are presented in the next paragraph.

There are two special components that combine other components, acting as the building entities for composite structures. The first, composes components in an ordered fixed sequence of components, and defines the merge by executing in sequence the pairwise merge of its elements. The second can be briefly described as a labeled version of the previous. Here components are tagged and merge is an enhanced set union that stores all components that hold distinct labels and computes the merge of component pairs with equal labels. This composition rule enables the description of mergeable tree structures as will be seen in the example in section 4.

The identification of an appropriate component should be driven from its relevance to a particular case and by its potential genericity. Once properties are established the design of the component must apply the restrictions that underly the environment model, in order to ascertain a correct and predictable behavior in all envisioned reconciliation patterns.

3.2 Environment model

The target environment for these components is driven by the capability to combine permanent availability with convergence guarantees. In fact, the model is orthogonal to the pessimistic/optimistic replica management dichotomy, since both these models build on the notion of possibly incompatible operations. Here the aim is to disallow semantically incompatible operations, which, in the process, might imply some adjustments to the overall behavior of the application or component. At lest on what concerns the portion of the state that is to be shared under autonomous operation.

Apart from permanent availability for performing all operations defined on the component data type (again, no pessimistic locks) each replica should, at any time, be able to derive new copies and each two replicas should be capable of converging into a new replica. These conditions establish the necessary grounds to enable autonomous replication and pairwise merge of replicas that happen to be at hand[2].

Components should also respect some properties that ensure a sound modeling of the underlying environment, and remove indeterminism:

Idempotency The reconciliation of a replica with herself should produce an unchanged replica.

Commutativity The order in which two replicas are supplied to the reconciliation procedure should not be relevant.

[2]As a side note, the interested reader is directed to an analysis of the implications of this operation pattern on current causality models [2].

Type: IncSet **extends** Basic	
$Write : Storage$ $Insert : Elem$ $Find : Elem \to Bool$	
$Init, Fork, Join, Leq$	
$\Sigma = 2^X$	$I = Y$
$Init()$	
$\sigma := \{\}$	$\iota := \perp$
$Insert(e)$	
$\sigma := \sigma \cup e$	
$Find(e) \to b$	
$b := \begin{cases} true & \text{if } e \in \sigma \\ false & \text{if } e \notin \sigma \end{cases}$	
$Join(\sigma\iota', \sigma\iota'') \to \sigma\iota$	
$\sigma := \sigma' \cup \sigma''$	$\iota := \iota' \vee \iota := \iota''$
$Leq(\sigma\iota', \sigma\iota'') \to b$	
$b := \begin{cases} true & \text{if } \sigma' \subseteq \sigma'' \\ false & \text{if } \sigma' \not\subseteq \sigma'' \end{cases}$	

Figure 2: Specification of an IncSet.

Associativity Pairwise reconciliation of three or more replicas should derive the same final result independently of the actual order that was applied in the reconciliation.

Together, these conditions ensure that the components merge policy has a sound behavior under all foreseen traces of the replica pool evolution.

3.3 Description of a component

Once identified, components should be classified and placed in a component hierarchy. Inserted components refer the name of the component who they extend and inherit the operations that they have defined. Here we briefly show the specification (in Figure 2) of a component that models a set with insert operations. This component is coined as IncSet and extends a Basic component that provides default operations for forking and initializing the state (state is represented as σ).

A deeper explanation of the underlying formalism and notation [1] is outside the scope of this article, but the main reasons for its adoption can be briefly presented.

A formal construction of the hierarchy and description of each component helps the classification of the components and guides their implementation, in particular on a object oriented language. The formal description of the component operations, the *Fork/Join* procedures and the documentation of the expected evolution order in the *Leq* function, provides a sound basis for the verification of the component behavior to check if it fits the environment model. In particular, by applying an appropriate formal verification procedure, it is possible to verify if the component is able to compute a merge from any two replica instances, and if this merge is compatible with the state evolution that is conveyed in the component operations that change the state.

4 Toolkit

4.1 Synopsis

This model served as a guide to the implementation of a toolkit that supports the construction of replica mergers while easing the identification of new generic structures. The core of the toolkit is a hierarchy of Java classes (shown in Figure 3) that implement several basic components and the two structuring components. These classes are ready to be integrated on applications that can fit sharable data under this model.

Once data is structured in this way, operations such as autonomous pairwise merge and forking of the structure are promptly available. Being this suitable for new Java applications, it fails to give enough support for the construction of the legacy data mergers that where sketched on section 2. This observation, leads to the definition of a simple intermediate language and mechanisms for writing the state of a given component structure in this language, as well as for reading a description and reconstructing the structure in memory. For this purpose, the top component in the hierarchy, the Joinable abstract class, holds a parser that can instantiate a component structure from a text input written in the intermediate language.

The intermediate language was targeted at simplicity and holds a syntax similar to Lisp's *S-Expressions* (see Figure 4). This language enables the description of instances of data structures that build around the set of toolkit components.

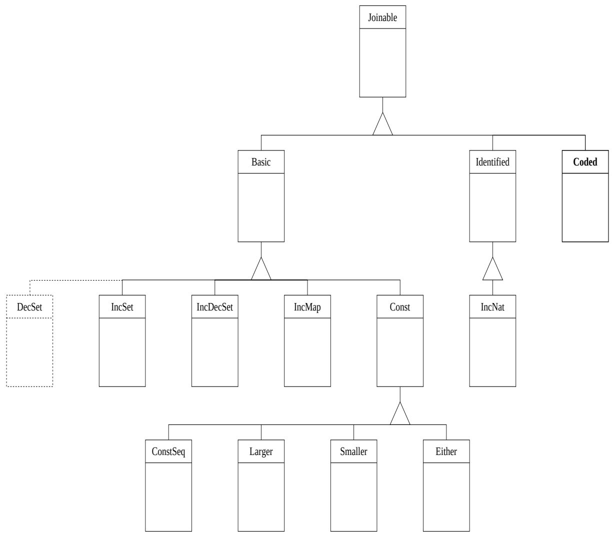

Figure 3: Main component hierarchy

A description in the IL can map tree structures with a variable number of branches [3] in each node.

In this language the CONST element is used to store generic data in the leafs, and is one of the elements that can be used to end recursion. This leaf element stores non structural data in a simple ASCII encoding that uses a pair of characters to represent one byte and keeps the language readable in 7 bit ASCII.

This sequence of bytes is treated as an opaque entity and can store any kind of encoded data. These sequences can be Java objects serializations or other language dependent encodings. Since this CONST data is non structural it does not influence the merge policy and permits interoperability between languages.

The Java framework comes with a small executable `joinilfiles` that given two files with structurally compatible IL representations computes their merge and writes it into a third file. With this executable acting as a back-end, it is possible to use the framework without Java programming. It suffices to produce the IL representation of the replicas that are to be merged and to interpret the generated replica, using any convenient language or tool.

[3]Due to the properties of the two aggregation components, CONSTSEQ and INCMAP.

```
FRASE ::= ‘(’ NAME <ENTITY> (FRASE)* ‘)’
FRASE ::= OPAQUE
NAME ::= ‘SET’ | ‘INCSET’ | ‘INCDECSET’ | ‘INCMAP’
       | ‘CONST’ | ‘CONSTSEQ’ | ‘EITHER’ | ‘DECSET’
       | ‘SMALLER’ | ‘LARGER’ | ‘INCNAT’
ENTITY ::= ‘"’ C* ‘"’
C ::= ‘0’ - ‘9’ | ‘a’ - ‘z’ | ‘A’ - ‘Z’ | ‘_’ | ‘-’
OPAQUE ::= (AA)*
A ::= ‘a’ - ‘p’
```

Figure 4: EBNF grammar of the intermediate language.

4.2 Representation of mail folders

One of the examples sketched in section 2, referred that the set of sentmail can be modeled by a grow-only set (IncSet). In the `pine` mail handling tool it is possible to have separate folders for each month sentmail. Suppose now that using `pine` on two machines, A and B, we have in machine A the folders `sent`, `sent-apr-98` and `sent-aug-98`, and on machine B the folders `sent` and `sent-aug-98`. Since `sent-apr-98` is only present on A, a convenient merge for these data files would merge the two grow-only sets `sent` and `sent-aug-98`, and keep the two resulting folders together with the unchanged unique folder `sent-apr-98`. In fact, this is the merge behavior that is abstracted in the component IncMap. IncMap captures the behavior of a grow-only partial function, which in this case maps labels (represented by Const that identify the folder name) into an IncSet of emails (emails are represented by other Const). When two instances of IncMap are merged, those labels that are unique in each IncMap are kept unchanged, and those who are common to both have their co-domains merged.

To provide a shorter and much more readable description of the IL representation, we omit the actual data encoding of the Const components and differentiate them by a 'label'. This label will take the place of the actual encoded data. The two structures that represent the sent mail in machine A could be expressed as:

```
(INCMAP
(CONST 'sent') (INCSET (CONST 'm346')(CONST 'm739'))
(CONST 'sent-apr-98') (INCSET (CONST 'm873'))
(CONST 'sent-aug-98') (INCSET (CONST 'm973')) )
```

And from machine B as:

```
(INCMAP
(CONST 'sent') (INCSET (CONST 'm576')(CONST 'm039'))
(CONST 'sent-aug-98) (INCSET (CONST 'm163')) )
```

When re-conciliated with the tool `joinilfiles` the generated replica will hold:

```
(INCMAP
(CONST 'sent-aug-98) (INCSET
                         (CONST 'm163')(CONST 'm973'))
(CONST 'sent') (INCSET (CONST 'm576') (CONST 'm739')
                         (CONST 'm346') (CONST 'm039'))
(CONST 'sent-apr-98) (INCSET (CONST 'm873')) )
```

Notice in this example that the order of elements is not relevant within a IncMap, or a IncSet, as both represent sets. Sets, in contrast to sequences, do not convey order to its contents.

The absence of ordered components drives from the impossibility of assigning a total order to events that occurred concurrently and autonomously [12]. On the other hand there are many contents that can order themselves by the inherent properties of their own data, such as the interest taxes in section 2.

4.3 Merge of bookmark lists

Some web browsers are currently available on a multitude of platforms, and for some users it is common to use the nearest browser when web access is needed. A natural consequence of this behavior is the spurious creation of concurrent replicas of bookmarks files.

When evaluating the details of an automatic merger for these files, some semantic changes need to be considered. For instance, most browsers, and Netscape in particular, enable the deletion of bookmarks entries. Concurrent insertions and deletions causes unsolvable problems to an automatic merger when there is no information that enables to decide their relative ordering. However in some cases, such as this, the benefits of making a small change to the existing semantics can overcome its side effects. Concretely, for bookmark files, it is reasonable to assume a grow-only semantics, which means that insertions have priority over deletions (which are not even registered). With the minor problem of having a previously deleted URL or folder reappearing when merging with another file that contains it, the user gains the ability to share and merge its partitioned bookmark index.

The Netscape bookmark file exhibits a reasonably complex data structure. It allows nested folding with URL sets at each node, and each entry is tagged with extra information like 'creation time' and 'last visit'. At a coarse level, the file can be seen as having a fixed header, a variable body and a fixed ending. This structuring is captured by a constant sequence (ConstSeq) with three elements, Const, IncMap and Const. A closer look at the first Const, the header shown in Figure 5, reveals that it contains a mutable portion, where the name of the folder owner is stored. By using the Either component to map this portion the new behavior is to select one of the conflicting owner names, if such conflict arises, which is a reasonable semantics for this case.

```
<!DOCTYPE NETSCAPE-Bookmark-file-1>
<!-- This is an automatically generated file.
It will be read and overwritten.
Do Not Edit !-->
<TITLE>Bookmarks for Remi Bara</TITLE>
<H1>Bookmarks for Remi Bara</H1>
```

Figure 5: Header of Netscape bookmarks file

The inner component in the sequence, the IncMap, gathers most of the complexity. The labels of the mappings are chosen to be the Const instances generated by the folder names or by the URLs. The first, are mapped into a constant sequence that, among other elements, contains a similar IncMap, thus starting recursion. The latter map into a different constant sequence that stores attributes such as the URL title (which was less appropriate than the URLs to use as index in the label) and the 'last visit' and 'creation time'.

These temporal attributes are not treated as Const components[4] but as Larger and Smaller components respectively. For them the appropriate semantic is to keep the latest 'last visit' time and the earliest 'creation time' [5].

The structured interpretation of the bookmark was materialized in a parser that interprets the HTML subset used on the files. This parser was responsible for the creation of the intermediate representation and acted as part of the front-end, of the generic merger, for the Netscape bookmark file format.

[4]They would possibly never match.

[5]These two components can only be used with elements that can be compared by their intrinsic data such as these. Unfortunately doing this comparisons with time stamps that where recorded on machines with unknown clock differences is only reasonable when timer or order reliability is not an issue.

5 Conclusions

Most approaches, to the design of merge policies for the re-integration of data that is independently changed by two or more parties, adopt an asymmetric vision of the replica holders by assigning a special primary role to one holder [3, 8]. Under this vision, reconciliation is often seen as re-integration of disconnected updates into a common central replica [4] or as the commitment of tentative updates [9]. In contrast, in our replication topology, we explored a symmetric vision in which replicas are equipotent. This positioning as benefits when the number of replicas is not known or when the replicated data is clearly symmetric, such as when supporting cooperation among groups of mobile systems.

The presented framework targets two distinct application areas that nevertheless fit the same replication model; the creation of semantically knowledgeable mergers for data files that are independently updated, and the support for data models that allow sharing among ad-hoc instances of personal information management (PIM) applications on mobile devices.

Having tested the toolkit with file mergers, we expect that it's use for cooperation among PIM applications will raise new perspectives in an area where applications have very conservative data sharing polices. In fact, most of the PIM applications for hand held devices ignore the possibility of user to user cooperation and only address synchronizations and backups with a desktop host. This, neglects vast possibilities of data sharing among users that can connect their devices but have no application support for synchronizing their data.

Recent industry efforts point to the incorporation of Java virtual machines on small portable computing devices and in specialized systems such as mobile phones. Other efforts, such as *Bluetooth*[6], lead into the creation of new forms of proximity networking that will complement current infrared solutions. Both these trends will certainly foster new research efforts on mobile to mobile cooperation with a Java emphasis.

Further information on this project as well as access to the package is provided in the Web at `http://gsd.di.uminho.pt/People/cbm/public/cdtao.html`.

6 Acknowledgments

To Remi Bara for his contributions to this project.

References

[1] Carlos Baquero and Francisco Moura. Specification of convergent abstract data types for autonomous mobile computing. Distributed Systems Group, Minho University, May 1997.

[2] Carlos Baquero and Francisco Moura. Causality in autonomous mobile systems. In *Third European Research Seminar on Advances in Distributed Systems*. Broadcast, EPFL-LSE, April 1999.

[3] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. The bayou architecture: Support for data sharing among mobile users. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, December 1994. http://www.parc.xerox.com/csl/projects/bayou/.

[4] Marc E. Fiuczynski and David Grove. A programming methodology for disconnected operation. Technical report, University of Washington, March 1994.

[5] Richard D. Guy, John Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeiner. Implementation of the ficus replicated file system. Technical report, University of California, Los Angeles, 1990.

[6] Jaap Haartsen, Mahmoud Naghshineh, Jon Inouye, Olaf Joeressen, and Warren Allen. Bluetooth: Vision, goals, and architecture. *ACM Mobile Computing and Communications Review*, 2(4):38–45, October 1998.

[7] P. Kumar. Flexible and safe resolution of file conflicts. In *USENIX Winter 1995 Technical Conference*, New Orleans, LA, January 1995.

[8] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Sixteen ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[9] Evaggelia Pitoura and Bharat Bhargava. Building information systems for mobile environnements. In *Third International Conference on Information and Knowledge Management*, pages 371–378, November 1994. http://www.cs.uoi.gr/ pitoura/.

[10] Peter Reiher, John Heidemann, David Ratner, Gregory Ski nner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the Summer Usenix Conference*, 1994.

[11] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experiences with disconnected operation in a mobile environment. In *USENIX Symposium on Mobile and Location Independant Computing*, pages 11–28, Cambridge, Massachusetts, US, August 1993.

[12] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 3(7):149–174, 1994.