Departamento de Informática

Faculdade de Ciências e Tecnologia

UNIVERSIDADE NOVA DE LISBOA

2825 Monte Caparica - Portugal

Technical Report

DI-UNL-04-2000

Data Components for Mobile Groupware: The DataBricks Approach*

DataBricks Project Team

José Legatheaux Martins, Luís Caires, Nuno Preguiça, Sérgio Duarte, João Costa Seco, Henrique João Domingos

Departamento de Informática
Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa

CITI
Centro de Informática e Tecnologias da Informação

Quinta da Torre, 2825-114 Monte da Caparica, Portugal
{jalm, lcaires, nmp, smd, jcs, hj}@di.fct.unl.pt

DataBricks PROJECT

http://asc.di.fct.unl.pt/DataBricks

* This work is partially supported by FCT, SAPIENS 33924, 2000.

Data Components for Mobile Groupware*: The DataBricks Approach

DataBricks Project Team

José Legatheaux Martins, Luís Caires, Nuno Preguiça, Sérgio Duarte, João Costa Seco, Henrique João Domingos

Departamento de Informática
Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa
Quinta da Torre, 2825-114 Monte da Caparica, Portugal
{jalm, lcaires, nmp, smd, jcs, hj}@di.fct.unl.pt

ABSTRACT

This paper presents the DataBricks project approach for component-based groupware development. The project is focused on the analysis, design and implementation of system support services suitable for an easier development of groupware applications in particular those aimed at mobile users. The goals of the project are twofold. First, to devise a specialized component model and associated programming language, backed by an appropriate set of development and support tools. Second, to supply a distributed architecture and a set of standard communication and data-management basic components that can be refined and recursively composed. These two closely related goals will speedup the process of building and adapting distributed groupware applications for mobile environments.

Keywords

Data management, component programming, mobile computing, development support.

INTRODUCTION

Advances in hardware and communication technology are leading to a dramatic increase in the use of portable computers. Moreover, a widespread use of applications and services in mobile computers and other embedded devices is also foreseen. This trend will create the opportunity for the development of innovative applications that will let multiple mobile users to cooperate and share information among them in order to achieve common goals - mobile groupware. These applications will require specific data management solutions to allow productive collaboration among users. These solutions must be implemented relying on distributed systems and distributed data management techniques (e.g. multiple object replication strategies, reconciliation of conflicting concurrent updates, fault-tolerance requirements, new group-oriented awareness support, support for variable quality of service adaptation, reconfiguration/adaptation facilities, etc…). To grant programmers the ability to explore sophisticated distributed systems solutions, as well as to enhance the adaptability and tailorability of the systems, it has been long recognized the need to provide simple programming abstractions.

The DataBricks project is focused on the analysis, design and implementation of system support solutions to ease the development of collaborative applications for mobile environments. To this end, we intend to propose a component model, coupled with a programming language; and supply adequate development and support tools. These will allow programmers to meet end-user requirements by adapting a set of application-level components, in an easy and flexible way. This model and support will be used to rebuild and enhance a distributed component framework specially geared towards the support of distributed data-management solutions in mobile environments.

This project extends our previous work [11] in two directions. First, it addresses new data management requirements to cope with resource-poor devices and variable connectivity. It will also provides a model of adaptation to environmental changes and enhanced replication and reconciliation mechanisms. Second, it addresses the definition of a new component model, a related programming language and support tools that extend our previous work in [3], and apply it to the mobile data management scenario. This new framework, tools and set of services will ease the way programmers have to deal with the complexity of the broad requirements imposed by our target environment.

The remainder of this paper is organized as follows. Section 2 presents the DataBricks approach. In section 3 we discuss

* This work is partially supported by FCT, SAPIENS 33924, 2000.

the limitation of current component-based development systems. In section 4, we present some of the basic components of our data-management component framework. Section 5 presents the programming language support at the component model level. Finally, section 6 concludes this paper with some final remarks.

THE DATABRICKS APPROACH AND OVERVIEW

The project's approach to address the complex data-management problems posed to mobile and collaborative computing is based on a distributed object store. The core of the object store is responsible to provide high data availability to applications, thus allowing users to perform their contributions despite voluntary disconnection and possible failures (in the servers or in the communication system). To achieve these goals, it relies on the combination of server replication and client caching using an optimistic approach – users are allowed to execute new updates independently being these updates later merged. Updates will be propagated among servers using an epidemic dissemination protocol.

This distributed platform is complemented with a component framework specially designed for data management in mobile environments. This framework is composed by a set of domain specific components and coordination abstractions that support the implementation of specific global solutions relying on the composition of pre-defined partial solutions. For example, a new application may define a new shared data-type relying on pre-defined components to handle concurrency control and awareness information.

The framework will integrate, in a coherent way, solutions to different problems and requirements: (1) partial replication schemes to support resource-poor mobile devices; (2) a model of adaptation to environmental changes; (3) integrated data-evolution awareness mechanisms; (4) mechanisms that enables the integration of legacy data-systems; (5) concurrency and cache consistency control mechanisms. Solutions for the last three problems have already been pursued in [11] using a similar approach. However, the first two problems demand an enhanced support at the component model level – for example, dynamic reconfiguration seems necessary to handle adaptation to environmental changes without imposing too much work to users. Additionally, the new component model should allow the creation of new base components relying on the composition of more elemental components (i.e., not only the application data types will be composed of pre-defined components, but also these components will be a composition of other basic components).

To this end, a basic component model with a built-in notion of typed component will be designed. The model will support incremental assembly, query, reflection and repository services, storage, migration, instantiation of components, and dynamic reconfiguration of instances, while ensuring type safety. The DataBricks component framework will be designed and implemented using this model. Language level support to express the software architecture and composition operations at a useful level of abstraction, extending de facto standards like DCOM, CORBA or JavaBeans/EJB, will also be provided. The designed language, referred provisionally as ComponentJ, will extend the Java language with general constructs to support the basic typed component model at the programming language level.

WHY DON'T WE USE A "WELL-KNOWN COMPONENT TECHNOLOGY"?

In general, the usual approach of component-based software engineering addresses the fundamental requirements that software systems need to be structured in terms of well-known abstractions and reusable functionality (which implies in architectural models based on the usual properties of object-orientation). The added-value of components, however, is that potentially, each component can evolve and can be adapted as a black box or a closed-product [13]. If some standard or well-known component-oriented development systems are being used today to build new complex applications, we can ask why in the DataBricks approach we don't start by simply adopt these available technology, focusing only on the specific issues of large scale and mobile CSCW/Groupware applications design and implementation.

In fact we find that those technologies have some limitations in terms of new requirements imposed by large scale distribution environments and in the logic of such applications as well as in the way to compose different classes of components as a multi-layered extensible framework . We will summarize some of these limitations in three different items: limitations at component's interaction model level, limitations in terms component's structure related with functionality and usability and, finally, limitations in terms of programming language support and expressiveness criteria for final programmers.

Component's interaction model limitations

In mobile and cooperative applications, the support for shared workspace services and awareness control integrated mechanisms require that, in different circumstances, we must be able to explore different models of group-oriented interactions.

For example, in a client-server model, a single server process manages all the shared cooperative workspace services. Clients, basically, implement the end-user interface, and interact with each other only through server mediation. Clients can use replicate/cache (part of) the shared workspace that is centrally controlled by the server to reduce access latency to shared artifacts, but the logic to control interdependencies and/or conflicts are centrally managed by a well-known entity – the server and its

internal components. This well understood paradigm is simple to program and basically is the model subjacent to well-known component technology like CORBA or COM/DCOM.

In mobile cooperative environments, the major drawback of client-server based models is the lack of fault-tolerance and no scalability properties when a server crashes or is unreachable (but not to client crashes).

The limitations of adopting the current well-known component's based technology is caused by the fact that we are convinced that the subjacent client-server philosophy isn't enough and basically is poor and inadequate.

In our approach, the perspective is to provide components that will be used in active group-replication models, where the component's state and data can be actively replicated by a set of servers or by a group of clients to avoid fault-tolerance drawbacks and to improve scalability. Furthermore, there are possible variants based on group-replication models: peer-based component's interactions (at client-level with no server accessibility), peer-group applications that reuse external services (e.g. authentication, security, name services), etc. Active replication groupware models that we are focusing on, imply that application and system support components must cooperate in a shared workspace adopting interaction models essentially different than client-server based paradigms.

Structure of component's functionality and its reusability

So, the meta-infrastructure of well-known today's component technology is client-server based and, in some sense, uses a dualistic approach. In one hand there are components and services encapsulating low-level system support abstractions, which are closed for the final application-level programmers. In other hand, application programmers use those low-level abstractions focusing component's programming techniques on end-user functionality and interface.

The base-system support that is subjacent to well-known component's programming environments implement base abstractions that are oriented for single-user applications. Issues like data-replication, group-awareness control mechanisms, group-resource sharing and floor control, etc., are leaved for groupware programmers.

Furthermore, the adaptation or dynamic reconfiguration of base system support services in terms of specific application needs is completely closed for groupware programmers. Low-level system components and infrastructures are reused by means of semantic interface standards, but without an agreement about what that semantic means when a particular operation is invoked by application level components. Thus, the programming approach remains meaningless, and there is no expressiveness to compose the available functionality in terms of adapting base infra-structural cooperative-oriented services to specific application needs.

For example, the main concern that components entering a system base functionality need to be connected to components that are already fixed and present is addressed by systems like CORBA or DCOM but that are no real composition and reconfiguration facilities leading with internal semantics of remote service components. The JavaBeans approach primarily targets the concern of dynamic loading of distributed components, but uses single point-to-point remote method invocations on volatile server-objects. None of those systems has the approach to an extensible middleware component-oriented architecture (as a component repository providing data management facilities, group-communication services, and application-artifact components that can be reconfigured and recomposed by the programmers, at different levels and in a simple way. We will show in the section 4 how an object-repository can be reused by means of an adaptive component-oriented structure taking in account the above considerations.

Support at programming language level

We also find a lack of suitable programming language support to the construction and composition of components as defined by the before mentioned models. For instance, CORBA and COM/DCOM provide no more than interface definition languages. COM+ has failed to support static type-checking, interface polymorphism and proper language support. SOM is a standard for deployment and composition of object-oriented component binaries, relying heavily on inheritance. Thus, SOM suffers from the semantic "fragile base class" problem [8] which we intend to circumvent in our approach by adhering strictly to black-box composition. JavaBeans allows packaging several classes into so-called beans, which are nothing but regular Java classes adhering to some interface conventions. As a matter of fact, no proper typed language support has been offered specifically for JavaBeans. Although Java is often referred as a component-oriented programming language, the study of programming language support for component based programming still relies on idioms rather than on high-level abstractions.

Our position is that without effective programming language level constructions able to express operations of composition, instantiation, and reconfiguration, the usability of a given component framework will be seriously affected, both at the systems and application level. In particular, we believe that extant component technology is not easily adaptable to some specific requirements of CSCW models (which are in general different from those based on a simpler and once for all fixed architecture, like the client-server approach).

Therefore, we intend to explore a new programming model, which includes specialized constructions to support

component-oriented programming. Other approaches have been suggested in specific contexts: Component Pascal, Units [7], Mixins [1]. In this project, we intend to develop a general notion of typed component following [3], that accounts for new forms of dynamic component adaptation and composition. The resulting programming language, called ComponentJ, extends Java with some basic ingredients essential to black-box object-oriented component programming styles like explicit context interdependences, dynamic binding and subtype polymorphism, multiple views, statically verified (dynamic) late composition, and avoidance of inheritance in favor of object composition. The approach using the ComponentJ programming language the Databricks model will be detailed and illustrated later on, in the section 5.

BASIC DATA COMPONENTS

The DataBricks component framework will be composed by several components that manage different aspects of the object "operation". In this section we will briefly present the most important aspects involved in the component framework and motivate for the need of type-specific solutions (in [11], we present a more detailed discussion of some of these problems).

Concurrency control

In the optimistic replication scheme adopted in the DataBricks project, users may independently perform concurrent updates. Although many algorithms (e.g. [9,12]) have been proposed to manage similar situations, no one seems adequate to all situations. Nevertheless, the use of semantic information has been identified [6,10] as a key element to merge the concurrent streams of activity. To maximize the flexibility in the handling of concurrent updates, we will propagate updates as operations, thus allowing data types to use not only the semantic information associated with the data type but also the semantic information associated with each performed operation.

The *concurrency control* component will be responsible to execute the updates performed by users. Two inter-related problems must be taken into consideration: how to guarantee that all data replicas evolve as expected and how to guarantee that users' intentions are respected.

The first problem can be handled constraining the execution order of updates. For example, executing all (deterministic) updates by the same order in all replicas leads all replicas to the same state. However, as different replicas may have received different subsets of updates, it is usually necessary to postpone the execution of some updates to guarantee that property. In some applications it is not necessary to achieve exactly the same state or due to semantic properties it is possible to rely on weaker orderings to achieve the same state. Several pre-defined components implementing different policies will be defined.

To tackle the second problem, three main approaches have been proposed in literature. First, the use of transactions - an update is committed if data values are equal to those observed by the user, otherwise it is aborted. Second, updates are transformed based on the updates executed after the state observed by the user [12]. Third, semantic information is used in the synchronization process [6,10]. Although we expect that most applications will rely on semantic information, components using other approaches (at least, operational transformations) will be available to programmers.

Besides defining a set of pre-defined implementations to the application programmer, we will decompose this component in more basic sub-components – at least, there will be a sub-component to define the order of updates and another one to control the execution of updates using that order. This way, it will be possible to create new *concurrency control* components composing pre-defined basic solutions.

Awareness

Awareness has been identified as important in the development of collaborative activities because individual contributions may be improved by the understanding of the activities of the whole group [4]. In DataBricks, each processed update may produce a piece of awareness information, thus allowing the system to process the information about the evolution of shared data. The awareness component allows each data-type to use a different mechanism to handle this information – in some applications it is interesting to notify users about the results of their updates (e.g. in a scheduler), while in others a log with the awareness information may be sufficient (e.g. in a document editor).

Our group is also involved in the design of a new event-dissemination architecture to propagate awareness information (see [5] for a preliminary design). This architecture will allow users to specify the way awareness information is propagated to themselves – for example, some user may request to be immediately notified using SMS/pager messages for important information, daily digests for information about activities that he is not directly involved on, and e-mail for other messages.

Clustering

In [11], each object managed by the system may represent a rather complex data object, such as a document or a calendar, and be implemented as an arbitrary composition of common objects. This approach may lead to some problems in resource-constrained devices where it is impossible to manage large amount of data. To overcome this problem, in DataBricks we will define the concept of an object cluster (a similar concept has been used in [2]). A cluster will be a set of objects that share common policies for (most of) the different aspects of object "operation", in

particular for concurrency control – this way, all objects evolve in a coherent way as a single unit. However, each object of the cluster may be manipulated alone. For example, using this approach a document may be defined as a cluster containing the document organization and the document basic elements (chapters, sections). A user may manipulate each section independently without the need to cache the complete document – this can be seen as a form of partial replication.

Other aspects

As in [11], it will be necessary to define other base components, such as components to store updates, to store meta-information about the objects, etc. We think that it is important to emphasize three aspects that will be addressed in the DataBricks project. First, we will provide base components that allow different data organizations. For example, we will provide a base data component that automatically manages multiple versions. Second, we will allow the integration of legacy/external systems. To this end, base data components that act as surrogates of legacy systems, in particular RDBMS, will be developed. We expect to further develop a preliminary approach that we have presented in [11] with the integration of partial replication mechanisms. Third, we expect to define mechanisms that allow the system and applications to adapt to modifications in the computing environment. In particular, we expect that variations in connectivity could be explored in order to provide the best possible quality of service for users - for example, depending on connectivity some operations may be executed immediately in servers or locally in the client cache.

PROGRAMMING LANGUAGE SUPPORT

Component-based programming can be seen as a refinement of object-oriented programming in which black-box composition of components that present explicit interfaces substitute inheritance as the basic reuse mechanism. A component programming language must include a notion of "component" as a first class entity and provide specific operations on components. By internalizing such operations at the programming language level one gets more opportunities for performing static analysis for the verification of interesting runtime properties of component instances. For instance, it should be possible to represent the dynamic composition of components (their dynamic reconfiguration) and their instantiation in a safe way - for instance, ensuring that all required resources are available. Following these principles, team members have designed a simple component model that identifies a few basic operations that bring component-oriented programming concepts explicit in a program. Aspects like name space management (some COM component can be referenced in a context that it does not exist, an execution error will occur) or obstacles to reuse resulting from undisciplined implementation inheritance are banned out from our model in order to eliminate implicit context dependencies of components. Moreover, we have defined for our model a type system ensuring that certain run-time errors do not occur due to ill-defined composition operations.

A concrete implementation of this model is the programming language ComponentJ, which is a mild extension of the Java Language we have been working on. To build applications, the programmer defines and (re)uses components instead of classes to build the intended software architecture. Component construction operations bring the expressive power of an architecture definition language and enable the application programmer to define

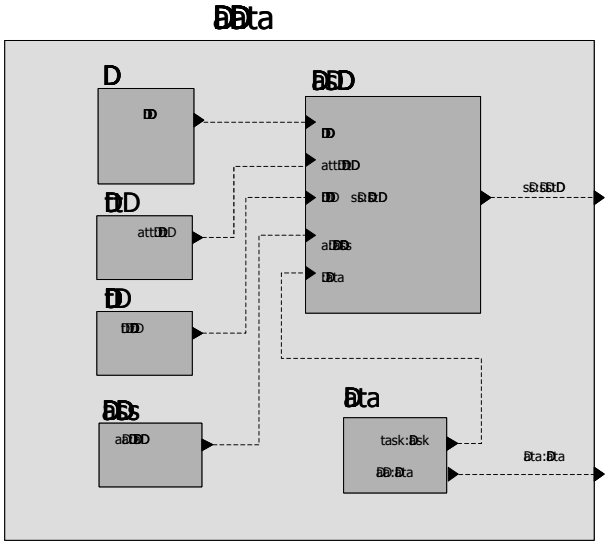

Fig 1. The `CSharedData` component graphical representation

components from scratch, or to compose existing components into a more complex architecture. Such components are class-like entities that can be instantiated, and present an explicit description of their interface both in terms of the services they provide, but also in terms of the services they require. Components can only produce instances when those required services are properly available in the environment (e.g. are plugged in). The basic syntactical elements in a component declaration are: explicit requirements (input ports), provided services (output ports), inner components and blocks of methods implementing the needed functionality. These elements can then be interconnected through a "plug" operation. Note that these composition operations can be used at runtime and components are first-class values that can be freely manipulated, that is, the definition of a component is not some form of (static) link-time specification, moreover the safety of such operations is ensured by the type system.

To be more concrete, let's illustrate the basic flavor of our approach with a specific example, an implementation of a shared data type to which we have called co-object [11]. In

Fig 1, we depict a set of base components (*CLog*, *CAttr*, *CConcCtl*, *CAwareness*, *CData* and *CCapsule*) used to compose a bigger component called *CSharedData*. Some services like logging, concurrency control and awareness are provided by the components *CLog*, *CAttr*, *CconcCtl* and *CAwareness*, which were chosen from the component repository. The component *CCapsule* coordinates this set of services, and provides a unique interface to the shared data type, by relying on the *CData* component, which bridges to the application data. Roughly, this architectural description will be expressed in ComponentJ as shown in Fig 2.

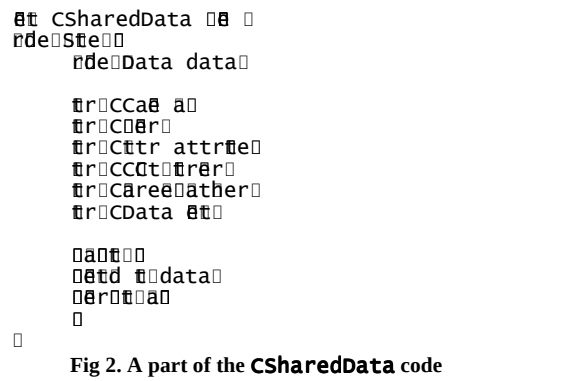

Fig 2. A part of the `CSharedData` code

CONCLUDING REMARKS

In this paper we have presented the DataBricks project. This on-going research effort is designing support for the development of collaborative applications for mobile environments. To this end, it will provide a data-management component framework that decomposes data-management in a set of complementary services (e.g., concurrency control). Different implementations will be provided to programmers allowing them to create the "right" solution composing several pre-defined basic solutions. Problems specifically related with mobile environments will be taken into consideration (e.g. support for partial replication and adaptation to variable connectivity). This component framework will be designed using a new component model that overcomes some limitations existent in widely used component models. The associated linguistic support allows programmers to easily compose their solutions at an adequate level of abstraction.

REFERENCES

[1] Bracha, G., Cook, W., Mixin-Based Inheritance. In *Proceedings of the Conference on Object Oriented Programming: Systems, Languages and Applications*, 1990

[2] Bakker, A., et al. From Remote Objects to Physically Distributed Objects. In Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems, Dec. 1999.

[3] Costa Seco, J., Caires, L. A basic model of typed components. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, LNCS 1850, Springer, 2000

[4] Dourish, P., Bellori, V. Awareness and Coordination in Shared Workspaces. In *Proceedings of CSCW'92*, 1992.

[5] Duarte, S., Legatheaux Martins, J., Domingos, H., Preguiça, N. DEEDS - An Event Dissemination Service for Mobile and Stationary Systems. In *Actas do 1º Encontro Português de Computação Móvel*, 1999.

[6] Edwards, W., Mynatt, E., Petersen, K., Spreitzer, M., Terry, D., Theimer, M. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proceedings of UIST'97*, Oct. 1997.

[7] Flatt, M., Felleisen, M., Units: Cool Modules for HOT languages. In *Proceedings of the ACM SIGOPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*. 1998

[8] Forman I. R., Conner M. H., Danforth S. H., Raper L. K. Release-to-Release Binary Compatibility in SOM. *ACM SIGPLAN Notices OOPSLA'95*, 1995

[9] Karsenty, A., Beaudouin-Lafon, M. An Algorithm for Distributed Groupware Applications. In *Proceedings of 13th ICDCS*, May 1993.

[10] Munson, J., Dewan, P. Sync: A Java Framework for Mobile Collaborative Applications. *IEEE Computer*, June 1997.

[11] Preguiça, N., Legatheaux Martins, J., Domingos, H., Duarte, S. Data Management Support for Asynchronous Groupware. To appear in *Proceedings of CSCW'00*, December 2000.

[12] Sun, C., Ellis, C. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of CSCW'98*, 1998.

[13] Szypersky, Clement. Emerging component software technologies – a strategic comparison, In *Software Components and Tools*, Springer 19(1), pp 2-10, 1998