The ABC Adaptive Fusion Architecture

Jonathan Bunde-Pedersen, Martin Mogensen, and Jakob E. Bardram
Department of Computer Science, University of Aarhus
Aabogade 34, DK-8200 Aarhus N., Denmark
{jbp,spider,bardram}@daimi.au.dk

ABSTRACT
Contemporary distributed collaborative systems tend to utilize either a client-server or a pure peer-to-peer paradigm. A client-server solution may potentially spawn direct connections between the clients to offload the server thereby creating a hybrid architecture. A pure peer-to-peer paradigm may on the other hand fully eliminate the need for a server. However, some situations call for the strengths of both approaches without relying on either of them. A system might both be used in environments where an infrastructure is present and in environments where it is not. In this paper we present an architecture and early implementation of a system capable of adapting to its operating environment, choosing the best fit combination of the client-server and peer-to-peer architectures. The architecture creates a seamless integration between a centralized hybrid architecture and a decentralized architecture, relying on what we have termed Peer-to-peer Distributed Shared Objects (PDSO). The proposed solution has been implemented and early evaluation has begun. Furthermore, the approach has been utilized to create a real distributed collaborative system for collaboration in hospitals.

Categories and Subject Descriptors
C.2.4 [Computer Systems Organization]: Computer Communication Networks—Distributed Systems; D.2.11 [Software]: Software Engineering—Software Architectures

Keywords
centralized architecture, decentralized architecture, hybrid architecture, Activity-Based Computing, ABC, Peer-to-peer Distributed Shared objects, PDSO, accountable, ephemeral

1. INTRODUCTION
Our research is based on long term empirical studies of work in the healthcare domain [1], and we use this as our application area. The Activity-Based Computing (ABC) middleware [2, 6, 4] is a generic middleware component which allows users and developers to organize their applications into activities, which form the basis for both synchronous and asynchronous collaboration. This article focuses on the distributed architecture of this middleware, which provides a novel adaptation mechanism between pure decentralized operation and a centralized hybrid approach.

Different suggestions to improve on the shortcomings of existing RPC-style interaction with remote single-object in RMI, CORBA, .NET Remoting, and DCOM have been suggested. For example, asynchronous RPC [22, 13], CORBA Event and Notification Services [16], and publish-subscribe [7, 8]. One specific approach to improve Java RMI is to support dynamic caching of shared objects on the accessing nodes, as done in Javanaise [10]. All of these approaches mitigate the challenges of unstable intermittent network connections and lack of scalability and performance but does, as such, not directly support object replication. The main contribution of our work, is the ability to enable every unit in the system to continue asynchronous collaboration even in situations without network connectivity, due to the full replication of state on all peers. Combined with the ability to adapt the architecture to the execution environment, this gives a very robust and scalable communication platform for building distributed collaborative systems.

The rest of this paper is organized as follows: First we will give a brief introduction to the concepts of ABC. Section 2 will present a scenario, which is used to motivate our fusion architecture. Section 3 presents the architecture itself, while section 4 briefly gives some implementation details and the evaluation performed so far. Lastly, section 5 gives a conclusion.

1.1 On Activity-Based Computing
Activity-Based Computing is a research area devoted to helping users cope with the increased complexity of daily computer use as well as providing a more suited interaction metaphor (than the desktop) for pervasive environments. The main concept driving ABC research is that of activities [6, 4].

Computational activities as a concept are the grouping of applications and files into logical bundles (or activities). This allows a user to easily switch between different work-tasks by resuming and suspending a bundle as a whole. A human activity is for instance the task of “writing a paper”, and the corresponding computational activity is then a description of the applications involved e.g. a text-editor, a bibliographical tool, and a browser along with their state. The state of an application is the information which is deemed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MPAC’06 Melbourne, Australia
Copyright 2006 ACM 1-59593-421-9/06/11 ...$5.00.
relevant to save across instantiations or state which should be synchronized in collaborative sessions. For example a text-editor may need the filename and position of the cursor as its state.

We deal with computational activities which can be created, suspended, and resumed on any activity-enabled computing device in the environment at any point in time. They can be shared asynchronously, or shared among several persons working simultaneously.

Our previous work has resulted in a middleware which supports this interaction with computational activities, providing mechanisms for organising ones work and applications into activities. Activities are stateful and when we talk about the state of an activity we refer to the already mentioned state of all the individual applications. Tracking this state allows us to suspend an activity by persisting the entire state and exiting all applications. Later, the persisted state can be retrieved and the activity resumed by instantiating the contained applications and restoring their state. Furthermore, we allow multiple users to participate in the same activity, synchronising the state of all applications in the activity across two or more machines.

2. SCENARIO

The following scenario presents a sequence of events which illustrate the levels of connectivity and access to resources which may occur in an emergency scenario, and which we seek to support with the architecture described in this paper.

Hansen, a paramedic, is on duty when two cars collide on the highway. He and his co-driver are based at a hospital 10 minutes away and they speed towards the crash site. Underway Hansen prepares himself by loading a basic emergency activity onto his PDA. This activity is prepared with applications into which Hansen can plot various information about injuries and otherwise relevant medical information. Once at the site Hansen delivers first aid and records information about the patients on his PDA. All emergency personnel have access to editing and adding information to each patient. When he gets in range of the ambulance the information is synchronized with a client inside the vehicle which immediately forwards the data through a GPRS or similar connection to the hospital. Sensors attached to the patients are included as part of the activity, and data is streamed over this connection as well as they return to the hospital. Meanwhile, doctors have been monitoring incoming changes to the activity and have begun to populate their local copies with information about the medical history of the patients. The doctors and other medical staff are thus prepared to take over the activity which is completely transferred and merged with the hospital infrastructure. The activity is now stored on a server in the hospital and further treatment information is recorded directly into it.

The scenario illustrates the use of activities in three different localities, each having a characteristic level of infrastructure access. The first level is found in the field where access to basic resources is, at best, scarce. Here, the roaming devices have intermittent infrastructure connectivity as they move in and out of range of the ambulance. It is not possible to rely on network connections and those connections which may be established have very low bandwidth. We find the second level of access inside or close to the ambulance, where the paramedic equipment may be hooked into the LAN of the vehicle which again may be connected with a remote infrastructure (the hospital) through a GPRS or similar service. The last level of access is found when the device gets in connectivity range of a stable infrastructure. Here, full access to the remote resources can be established and we may rely on all connections. In all three cases the roaming device may have transient connections to peers close by and may at these sessions exchange information in a peer-to-peer fashion.

In the following section we will present our architecture for the ABC middleware and how we support the above scenario.

3. ARCHITECTURE

An important issue in realising the aforementioned scenario, is the choice of architecture to base the system upon. We will begin by describing the architecture of the core ABC middleware. Then we will give two sections describing the architecture of the communication modules; the hybrid architecture for accessing an infrastructure tier and the distributed object scheme which is used for peer-to-peer communication. Finally, we will present how these two modules are fused together and how the adaptation to a given environment proceeds.

3.1 Overall architecture

The core of the ABC middleware has a basic two tiered architecture. An infrastructure layer provides support for persistence, event propagation, and methods for synchronizing activities. This layer may reside on one or more servers and is independent of the client layer. The infrastructure layer is responsible for persisting activities through the Activity Store component, for managing runtime state and events through the Activity Manager and for distributing state-changes and other collaboration-related data through the Collaboration Manager. The Context Service component keeps track of users and machines as well as their context. The layer is commonly deployed on one or more servers – physically separated from the client layer.

The client layer runs on all active devices and interfaces with the local operating system, applications and data. The Activity Controller provides handles for resuming and suspending an activity, and manages connections to the infrastructure as well as to other clients. It is also within this layer we have implemented the peer-to-peer communication mechanisms and the distributed object scheme.

The connection to the infrastructure uses a stateless HTTP-like protocol called ABCP. The ABCP protocol is an XML-based protocol with storage methods GET and POST as well event-based communication through PUBLISH and SUBSCRIBE methods. The storage methods act as their HTTP counterparts and allows for retrieval and posting of activity-data. Activities are described using an XML format so they are highly portable. The client itself also maintains a cache which permits users to keep working in an activity in situations no external connections are possible. Once a connec-
tion has been re-established, the cache will synchronize with the infrastructure once again.

The Collaboration Controller allows an activity to be shared. When sharing an activity the state of all services in the activity is synchronized on all participating devices.

3.2 Centralized hybrid

If an infrastructure layer is available, then the middleware functions as a centralized hybrid architecture. The infrastructure layer is used for storage of vital information while non-vital is distributed in a peer-to-peer fashion. This approach ensures delivery guarantees from the transport layer by making a key differentiation between what we have termed as accountable and ephemeral events [3]. In replicated collaborative architectures concurrency control between events on distributed clients is absolutely central in order to maintain correct behavior of the distributed system [17].

We use the term ‘accountable’ for events relying on concurrency control, because the system needs to be accountable for the correctness and timing of these events in order to be considered as a well-behaved collaborative system. Examples of accountable events are the classical text insert, move, and delete commands in collaborative editors or the state changes in general purpose frameworks like Corona [19] or GroupKit [18, 9]. An IP-based infrastructure would use TCP or reliable multicast to distribute such events. Accountable events are thus routed through the infrastructure and made subject to concurrency control before being distributed further.

There are, however, a range of other kinds of events which are not subject to the kind of accountability. Such events are typically absolute values, independent of previous and subsequent events, and may even be missing or disregarded if needed. We call these events ‘ephemeral’ because they are short-lived and transient. Examples of such events are telepointer events, voice events, and other collaborative awareness events like the ones in the MAUI Toolkit [11]. An IP-based infrastructure would typically use unicast datagrams to distribute such events. Ephemeral events are thus distributed optimistically in a peer-to-peer fashion.

There are several advantages in using a hybrid architectural model and then differentiating between accountable and ephemeral events. Since state management and concurrency control is done on the server, simple – often optimistic – concurrency algorithms can be employed. The task of accommodating late joiners and asynchronous collaboration is simple also and there is no peer discovery required [21]. On the other hand, because a large portion of the collaborative events and awareness support is done peer-to-peer, responsiveness to user input and robustness to network failures are high. In addition, the system scales much better both with regard to network bandwidth and central processing power [3].

Still a hybrid architecture possesses some of the drawbacks from both the client-server and the peer-to-peer architectures. However, these can be reduced considerably in a number of ways.

Firstly, by using stateless communication protocols, as we do between client and infrastructure layers, the problem with network and server failures can be mitigated. If a client fails, the server is not left with any state to maintain and the client can reconnect seamlessly once it recovers.

Secondly, by using client side caching and algorithms for re-joining a server after a disconnection, we have nearly eliminated the problem of temporary server crashes. If the infrastructure layer fails, clients may be unable to contact the server for a while, but communication can be resumed and clients re-synchronized once the infrastructure has recovered.

3.3 Decentralized ad-hoc

The centralized hybrid approach still relies on a server to synchronize and to enable live sharing of data. In order to enable total infrastructure independent operation we need an additional pure peer-to-peer scheme. We have created a fully decentralized architecture, relying on what we have termed Peer-to-peer Distributed Shared Objects (PDSO), since we rely on local object replicas keeping themselves synchronized using an underlying peer-to-peer infrastructure. This approach is an extension of existing research on distributed shared objects. The fundamental principles behind the design of our approach are:

Physical distribution Instead of viewing a distributed object as an entity running on a single host with others accessing it remotely, we physically distribute a copy of the object to all hosts using this object in an application. Hence, applications access and use objects as local objects which ensures fast responsiveness. Objects are distributed on creation (remote instantiation) and removed from the local address space on deletion (distributed garbage collection).

Synchronized objects The state of any distributed shared object is kept synchronized in real time, if possible. Hence, state changes are propagated to all object replicas. State synchronization is handled by the underlying infrastructure, but the objects themselves are involved in potential conflict resolution, using domain specific conflict resolution algorithms.

Peer-to-peer update Physically distributed objects rely on a peer-to-peer – or object-to-object – synchronization strategy. Hence, no central entities like an object broker or an object registry are involved in object registration or lookup. Each object is responsible for looking up and synchronizing with its replicas. This principle makes distributed programming simple from
the developers point of view since there are no configuration overhead associated with development and deployment.

**Responsive** Objects are used in highly interactive applications and need to embody a fast update protocol. This rules out pessimistic concurrency control which typically uses some kind of distributed transactional scheme [20, 14, 19] or distributed locks [12].

**Distribution-aware** Objects are distribution-aware. This means that a shared object must be declared to be distributed, must handle potential conflict resolution, and must consider the kind of delivery guarantees wanted in the network transport layer.

The principles involved in peer-to-peer distributed object sharing is illustrated in figure 2, showing a distributed object with a replica in four different address spaces (A1–A4), using object-to-object communication pathways to keep the replicas synchronized and sending remote instantiation and garbage collection events.

![Figure 2: A set of peer-to-peer distributed shared objects (PDSO) distributed over four address spaces (A1–A4). Each address space holds a local replica of the object which is synchronized by object-to-object eventing. Address space A5 does not currently participate in the object sharing but may join one or more of the objects.](image)

The main idea is that a distributed object, called a Peer-to-peer Distributed Shared Object (PDSO) consists of several local replicas that keep their state synchronized. Each local replica is identified by an Object Identifier (OID), a PDSO consists of the set of local replicas with the same OID. A set of PDSOs can be tied together by use of distributed variables; we call such a set a group.

To be more precise, we are using the following terms:

**OID** Object Identifier. The OID is used to name a single instance of a local object replica. Several local object replicas can have the same OID, but not within the same namespace.

**PDSO** Peer-to-peer Distributed Shared Object. A set of local object replicas, that keep their state synchronized. A PDSO is defined as the set of local object replicas named by the same OID. I.e. PDSO(\(s\)) = \{local replicas x|OID(x) = s\}

**Group** A set of PDSOs, defined by the transitive closure of a specified PDSO \(x\). I.e. all PDSOs in the object graph that can be reached from \(x\).

\[\text{Group}(\text{PDSO}(x)) = \{\text{PDSO}(y)|\text{there is a path from PDSO}(x) \text{ to PDSO}(y) \text{ in the object graph}\}.\]

Figure 3 shows five PDSOs distributed over four address spaces. The PDSOs are named A, B, C, D, and E respectively. Each distributed object is comprised of several local replicas, all named with the same object identifier (OID). The local replicas comprising the PDSO named B have been highlighted. Also shown in the figure are three groups, namely Group(A), Group(C), and Group(D). The groups are the transitive closure of the named PDSO. Group(A) is therefore comprised of PDSO(A) and PDSO(B), whereas Group(D) equals PDSO(D) because the edges in the object graph is directed. Notice also that two peers, address space A2 and A4, are members of more than one group.

![Figure 3: Five PDSOs (A–E) distributed over four address spaces (A1–A4). Each address space hold a local replica of the PDSOs in the groups the peer is member of.](image)

Using the concept of peer-to-peer distributed shared objects, we are able to create a fully decentralized communication architecture, capable of supporting the same types of collaboration as the centralized hybrid architecture. Delivery guarantees are handled in the same manner as the centralized hybrid architecture - by distinguishing between accountable and ephemeral communication. The decentralized communication architecture enables us to support ad-hoc collaboration in environments without existing infrastructure.

### 3.4 Adaptive fusion

The infrastructure needed to support the scenario of section 2, has to incorporate properties from both the centralized and the decentralized architectures. Therefore, we have created a fusion architecture, capable of adapting to the environment and delivering the properties needed in the given setting. The architecture adapts its communication strategy based on information obtained from the incorporated service discovery mechanisms. It will try to provide as good a platform for collaboration as possible.

In the scenario the paramedics move around in the real world, thereby changing the environment in which their personal digital devices reside. At the hospital, the supporting
infrastructure is fully evolved, including high-speed, reliable network connections. This permits the use of centralized architectures, possible utilizing direct communication between the clients, thereby leveraging the server load. Moving away from the hospital, the environment changes. Network connections decrease in both bandwidth and stability. This means that collaborative widgets, such as tele-pointers and voice-links, has to give way to ensure bandwidth for essential data communication.

Upon arrival at the accident scene, infrastructure may be non-existing or unreliable. The only infrastructure at the accident scene, is the infrastructure that the paramedics bring themselves. This means that collaborative middleware has to utilize pure decentralized architectures in order to guarantee operation. The following sections will give an in depth description of the different modes the architecture will operate in at the different environments.

Figure 4: The Adaptive Fusion Architecture is capable of adapting to the environment. Based on service discovery mechanisms and resource awareness, the infrastructure will morph from centralized to decentralized over various fusion architectures.

The implemented middleware uses a service discovery mechanism, which will tell the system which resources are available in the environment. Based on these information, the system will choose the best mode of communication. In the first phase of the scenario, the paramedics are located at the hospital, which means, that they can access the stable infrastructure of the hospital. The service discovery mechanism will obtain a reference to the server and the different clients will be set up according to the earlier presented hybrid architecture. The collaboration enabled in this setting is fully evolved. State changes will flow through the server and be forwarded to other clients participating in the collaboration. Tele-pointers, voice-links, and other collaboration widgets will flow directly peer-to-peer between the clients. This setup is very stable and easy to deploy and maintain. System administrators can guarantee quality of service and data backup mechanisms, thereby easily accommodating the legal and political requirements for data storage on a hospital.

As the paramedics enter the ambulance, they leave the stable environment of the hospital, and enter another environment where network connections are more transient and bandwidths are low. In this setting the paramedics may use the client residing in the ambulance. This client will be a member of the server based network, but due to the lack of bandwidth, only a degraded form of collaboration is supported. As in the hospital state changes will be sent through the server, but the client will not be able to establish tele-pointers, voice-links, and other collaborative widgets, which uses ephemeral events as their distribution mechanism. If the client should loose the network connection, and thus the connection to the server, the paramedics are able to continue their work on a local copy of the activity provided by the cache component. Upon reconnection, the client will automatically merge the work done into the activity residing on the server.

At the accident scene, the middleware cannot rely on any existing infrastructure. The infrastructure may have been destroyed in the accident or there may not be any infrastructure there at all. The paramedics bring their own infrastructure based in the ambulance, but perhaps the paramedics have to move away from the ambulance and thereby also away from the infrastructure. In such a setting, the personal digital devices of the paramedics will engage in an ad-hoc network. The service discovery mechanisms will be unable to detect a server, but be able to detect the other personal digital devices in the area. Using the earlier described peer-to-peer distributed shared objects, the middleware will utilize a completely decentralized architecture, enabling collaboration between the paramedics at the accident scene.

The collaboration supported will be exactly the same as at the hospital, including state changes, voice-links, tele-pointers, etc. The difference between this scene and the hospital scene, is that both ephemeral and accountable events will be distributed in a peer-to-peer manner. Even if the different clients in the ad-hoc network should loose connectivity for a period of time, the individual clients will offer the ability to continue work while disconnected, and upon reconnection the incorporated synchronization mechanisms will resynchronize the network. If one of the clients in the ad-hoc network should come within range of the ambulance, the client in the ambulance will join the ad-hoc network, while still being a client in the server based network. This enables the ambulance client to act as a bridge, relaying state changes in the decentralized network to the centralized network and the other way around. This means that it is possible for a client in the decentralized network to collaborate with a client in the centralized network, creating a network architecture which is a fusion of a centralized and a decentralized architecture.

4. FUTURE WORK

Evaluation of the fusion of these schemes as well as the adaptivity mechanism is still future work. Both schemes have been evaluated separately and these results show that: The centralized hybrid scheme scales reasonable well (to at least 50 simultaneous users) on a standard 10Mbps LAN - and the differentiation between accountable and ephemeral events leads to a significant reduction in server-based traffic. Rejoining a client is accomplished almost instantly because only accumulated changes are used to resynchronize the state of the client, so the amount of data needed to be transferred remains small. A detailed technical evaluation is available in [3]. The Peer-to-peer Distributed Shared Object scheme has been evaluated along four dimensions: Completeness, Complexity, Performance, and Utility. The results have been reported elsewhere [15], and show satisfying results in every aspect.

Ongoing evaluation mainly focuses on the utility aspects of utilizing the peer-to-peer distributed objects as platform for various pervasive applications.
5. CONCLUSION

This paper presented a fusion architecture for the ABC middleware, which enables activity-based clients to collaborate in both environments with an infrastructure as well as in those without. The adaptation mechanism ensure that work can continue in both environments and that synchronous collaborative features degrade gracefully. The architecture furthermore demonstrates a novel way of incorporating two disparate communication modes without relying on a hybrid infrastructure.

6. REFERENCES


