DOLCLAN – Middleware Support for Peer-to-Peer Distributed Shared Objects

Jakob E. Bardram and Martin Mogensen

Department of Computer Science, University of Aarhus
Aabogade 34, DK–8200 Aarhus N., Denmark
{bardram,spider}@daimi.au.dk

Abstract. Contemporary object-oriented programming seeks to enable distributed computing by accessing remote objects using blocking remote procedure calls. This technique, however, suffers from several drawbacks because it relies on the assumption of stable network connections and synchronous method invocations. In this paper we present an approach to support distributed programming, which rely on local object replicas keeping themselves synchronized using an underlying peer-to-peer infrastructure. We have termed our approach *Peer-to-peer Distributed Shared Objects* (PDSO). This PDSO approach has been implemented in the DOLCLAN framework. An evaluation demonstrates that DOLCLAN can be utilized to create a real distributed collaborative system for ad-hoc collaboration in hospitals, which demonstrates that the approach can support the creation of non-trivial distributed applications for pervasive computing.

1 Introduction

Support for distributed computing in contemporary production OO languages is based on the remote-procedure call (RPC) paradigm [8] where methods on single-copy objects are accessed remotely from other objects. Both Java RMI and .NET Remoting are examples of this approach. A fundamental challenge to this paradigm is its inherent assumption of a reliable infrastructure. Object registration and lookup is primarily done through initialization, since remote object invocation assumes that objects stay on a stable host machine with reliable networking connections. Remote object invocation is furthermore done synchronously with blocking method calls. When programming applications for pervasive computing environments these assumptions do no longer hold. Such an infrastructure is completely different, consisting of a heterogeneous set of more or less stable host devices with intermitted network connections. Using RPC, RMI or similar under these circumstances leads to highly unstable applications, unless the programmer goes through a lot of work of manually handling all sorts of networking and runtime exceptions.

In order to provide a more resilient programming environment for this unstable runtime infrastructure we propose a new approach for distributed programming, which rely on local object replicas keeping themselves synchronized using an underlying peer-to-peer infrastructure. We have termed our approach *Peer-to-peer Distributed Shared Objects* (PDSO), which has been implemented in the DOLCLAN framework. In this approach, each participating peer maintains a local copy of the object and executes

J. Indulska and K. Raymond (Eds.): DAIS 2007, LNCS 4531, pp. 119–132, 2007.

processes that keep these replicas coordinated in real time. This approach has a range of advantages. First, it keeps applications responsive because the applications are much more robust with respect to network latency. Second, applications can continue to run when disconnected from the network. Third, computational and network load is distributed across the whole network of clients and is no longer tied to the machine hosting the remote object. Fourth, finding and joining a network may be simplified since all participating clients can function as the gateway to the network. There are, however, also a range of drawbacks to this approach, mainly associated with the overhead of distributing and managing the placement, synchronization, and replication of data, as well as handling the underlying communication technology and topology. The purpose of DOLCLAN is to help the programmer handle this real-time object synchronization of distributed objects.

The main contribution of DOLCLAN is a novel peer-to-peer distribution mechanism for object sharing which is especially suited for the creation of collaborative applications in a pervasive computing environment. This object sharing mechanism provides optimistic synchronization strategies, easy deployment of distributed applications, and support for different delivery guarantees – all of which can be accessed by the application developers, if needed.

1.1 Related Work

Different suggestions to improve on the shortcomings of existing RPC-style interaction with remote single-copy objects in RMI, CORBA, .NET Remoting, and DCOM have been suggested. For example, asynchronous RPC [30,18], and CORBA Event and Notification Services [22]. One specific approach to improve Java RMI is to support dynamic caching of shared objects on the accessing nodes, as done in Javanaise [13]. Research has also been done within asynchronous method invocation [18,30], tuple spaces [10,19], or more generally with publish-subscribe interaction styles [21]. All of these approaches mitigate the challenges of intermitted network connections, and lack of scalability and performance in RPC. But they do not support object replication and reconciling, and therefore does not allow the application to continue to access and update the distributed objects while disconnected from the network. In certain tuple spaces, a *global virtual data structure* is achieved by letting each device hold a local copy of a tuple space which is transparently shared with the tuple space of the connected devices [23,9]. By accessing its local tuple space, each component has efficiently access to the global tuple space. Hence, actions that are perceived as local actually has global effects. This approach is similar to distributed objects but does not as such support distributed object-oriented programming, and is not designed to disconnected work since it does not provide support for reconciling work done while disconnected.

Orca [4,3,2] is an object based programming language and distributed shared memory system (DSM). Orca is based on distributed coherent objects, e.g. Orca does not invalidate objects on write, but propagates the write to all copies of the object. This is done by sending all writes to a primary copy of an object called *the object manager*, which then updates all copies. Coherence is ensured via a two-phase commit protocol and by sending operations using totally ordered group communication, so all updates are executed in the same order at all machines. To a certain respect, our work extends the

principles of Orca, including using a write-update protocol rather than a write-invalidate protocol to address the core consistency challenge in object replication. Our infrastructure also relies on totally ordered group communication. Our work, however, is different in at least two ways. First, we rely on direct object-to-object data synchronization and do not use specialized object managers counting read and write operations. This significantly simplifies program development and deployment. Second, our language support is part of the widely used C# language and does not require a specialized language like Orca.

Globe [1,15] is an object oriented framework for developing wide area distributed applications using distributed shared objects (DSO). On the one hand, the Globe DSO lets the application programmer concentrate on implementing business logic and not worry about distribution and communication. On the other hand, Globe recognizes the need to be able to implement object specific policies on issues such as distribution, replication, and concurrency controls. By implementing a 'replication sub-object', the programmer can create a specific replication policy. Depending on the implementations of the sub-objects, the local object will function as a proxy object, forwarding requests to a real object. Alternatively, the local object can carry out calculations on a local copy of the object state and – depending on the implementation of the replication sub-object – the new state can be propagated to other instances of the distributed shared object. This possibility to override default functionality by implementing specific sub-objects yields a flexible, highly extensible, and scalable framework for creating distributed applications. The approach, however, comes with a huge overhead for the programmer who has to design and implement replication policies in the replication sub-objects.

Our work is situated within this line of research on distributed shared objects and makes contributions primarily in three aspects: (i) we provide language support for a widely used OO language (as compared to special languages like Orca), (ii) we have a simple peer-to-peer distribution and synchronization mechanism for shared objects, and (iii) we support an optimistic synchronization strategy based on user-defined merging methods in write-update protocols.

2 Peer-to-Peer Distributed Object Sharing

The fundamental principles behind the design of our *peer-to-peer distributed shared object* 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 the 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 the development and deployment of a distributed application.

Responsive. Objects are used in highly interactive applications and needs to embody a fast update protocol. This rules out pessimistic concurrency control which typically uses some kind of distributed transactional scheme [27,26] or distributed locks [17].

Distribution-aware. Objects are distribution-aware. This means that a shared object must be declared as distributed, must handle potential conflict resolution, and must consider the kind of delivery guarantees wanted in the network transport layer. These issues are normally shielded from the application programmer but, as explained above, we deliberately want these things to surface in the language support for distributed programming.

The principles involved in peer-to-peer distributed object sharing is illustrated in figure 1, showing a set of distributed objects with replicas 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.

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.

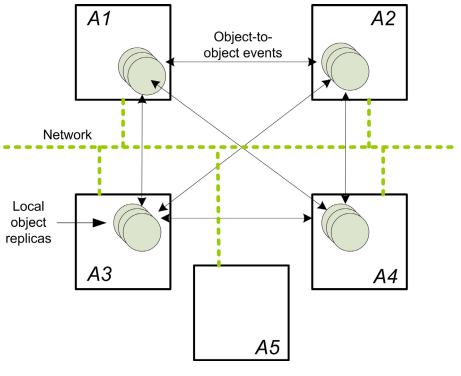

Fig. 1. A set of peer-to-peer distributed shared objects (PDSO) distributed over four address spaces (A1–A4). Each address space holds a local replica of the object which is synchronized by object-to-object eventing. Address space A5 does not currently participate in the object sharing but may join one or more of the objects.

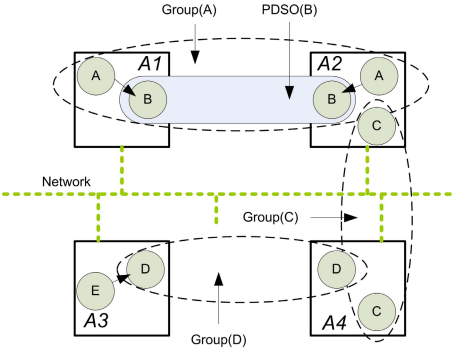

Fig. 2. Five PDSOs (A–E) distributed over four address spaces (A1–A4). Each address space hold a local replica of the PDSOs in the groups the peer is member of.

Each local replica is identified by an Object Identifier (OID). A PDSO consists of the set of local replicas with the same IOD. 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 $PDSO$s, defined by the transitive closure of a specified $PDSO\ x$. I.e. all PDSOs in the object graph that can be reached from x.
$Group(PDSO(x)) = \{PDSO(y) | there\ is\ a\ path\ from\ PDSO(x)\ to\ PDSO(y)\ in\ the\ object\ graph\}$.

Figure 2 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 are directed. Notice also that two peers, address space A2 and A4, are members of more than one group. Groups are used as a scoping mechanism enabling peers to join only a subset of the object graph.

With respect to delivery guarantees from the transport layer we make a key differentiation between what we have termed *accountable* and *ephemeral* events [5]. In replicated collaborative architectures concurrency control between events on distributed clients is absolutely central in order to maintain correct behavior of the distributed system [24]. We use the term 'accountable' for this kind of distributed events, because the system needs to be accountable for the correctness and timing of these events in order to create 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 [26] or GroupKit [25,12]. An IP-based infrastructure would use TCP or reliable multicast to distribute such events. There are, however, a range of other kinds of events which are not subject to the same kind of accountability. Such events are typically absolute values, independent of previous and subsequent events, and may even be missing or dismissed 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 [14]. An IP-based infrastructure would typically use multicast datagrams to distribute such events.

We argue that giving the application developer access to these low-level transport issues in distributed computing is important since he can make appropriate judgments on the choice of delivery guarantees based on application-specific concerns. Such

concerns are not present in contemporary language support for remote objects, like Java RMI, CORBA, .NET Remoting, and DCOM[1].

2.1 An Example

The PDSOs can be used to construct a model for a distributed application, by connecting objects via distributed fields within the objects. Such distributed fields can be declared by using either the *accountable* or *ephemeral* keywords, supported by the language constructs implemented to support the PDSO scheme[2]. A simple example could be a model for a distributed eater or Pacman game. The game consists of a game controller and a game model, which will be used to distribute state between the participating peers. The model is comprised of a game name, a score, a position of the eater and a list of stones visible on the board. Figure 3 shows the model represented as an UML diagram.

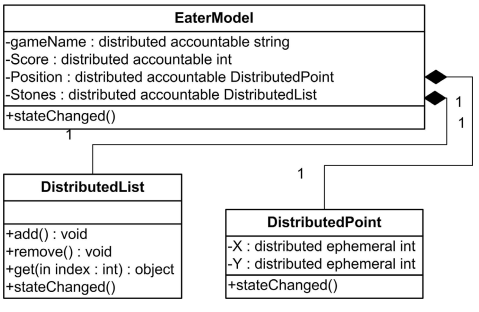

Fig. 3. UML diagram showing the presented part of the EaterModel

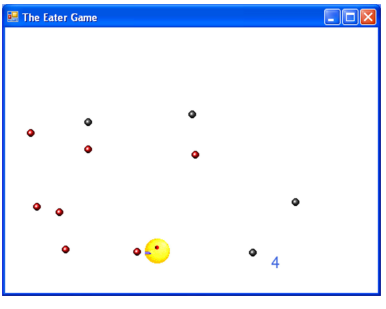

Fig. 4. 'The Eater Game' showing the Pacman, stones, and the score

The game's name and the score is modeled as distributed accountable properties. This enables us to intercept flow control every time the fields are set, and notify the view and the other participating peers of the change. The position of the eater is modeled by the `DistributedPoint` class, which contains two distributed ephemeral properties. Each property corresponds to the X and the Y position of the eater. The choice of using ephemeral variables emphasizes speed of delivery rather than delivery guarantees in changes of the eater position. Finally, the model contains a list of stones, which are visible on the game board. The stones are kept in a `DistributedList`, which is a list created using distributed accountable variables inside PDSOs for holding satellite data.

When a peer starts an instance of the eater game, it will first obtain a local replica of the `EaterModel` and the PDSOs in the transitive closure of this PDSO. After joining

[1] It is, however, interesting to note that in SUN RPC the implementer has the choice of using either UDP or TCP for transporting remote procedure calls and for broadcasting remote procedure calls [8].

[2] The language constructs is beyond the scope of this paper, but has been presented elsewhere [20].

the game, the state of the model will be replicated between the different local replicas. This is done, by assigning new values to the distributed variables. If a peer for instance moves the eater, new values will be assigned to the distributed properties X and Y in the `DistributedPoint` object. The infrastructure will intercept flow control and propagate the new values around the network. When the new values reaches the designated local replicas in the other address spaces, it will be set on the corresponding objects. This will cause the objects to fire the `stateChanged` event on the objects, which in turn, will fire the `stateChanged` events on the local replicas of the `EaterModel` and the different views can be updated. The same is true for changes in any stone or the score. Notice also, that if any of the distributed variables is assigned the `null` value, this value will also be propagated around the network. When this is done, the object which was previously referenced by the distributed variable might become subject to garbage collection.

If a peer becomes disconnected for a period of time, subsequently reconnecting to the network, the state of the peer and the state of the network might diverge. In such a case domain specific conflict resolution methods, specified by the programmer, will be used to handle conflicts bringing the network back to a consistent state.

3 Infrastructure Support

The proposed concepts presented have been implemented in the DOLCLAN[3] infrastructure [20], which uses a pure peer-to-peer architecture and supports object distribution, state synchronization, object discovery, peer joining, event ordering, and concurrency control. This section describes the system and network architecture (section 3.1) and the architecture supporting this infrastructure on each participating peer holding the object replicas (section 3.2).

3.1 System and Network Architecture

Communication between peers can be carried out in several ways. Events and messages can be either unicasted or multicasted, and both reliable and unreliable communication channels can be utilized. In our current implementation we have chosen to utilize the possibilities of multicasting since many peers will have to receive the same information. Point-to-point connections are possible but would require a quadratic number of unicast connections between peers or the utilization of a sophisticated routing scheme, which would impose an extra performance penalty and delay messages. Even though many peers will have to receive the same information, this is not true for all peers. Therefore the infrastructure has a control channel for reaching all peers and individual channels for smaller groups.

Peer-to-peer Distributed Shared Objects require three things of the underlying system infrastructure: (i) service discovery which enables a peer to find existing PDSOs, (ii) peer joining which enables a peer to join a group and get the state synchronized, and (iii) synchronous object state replication amongst connected peers.

[3] Distributed Objects in Loose Coupled Local Area Networks.

Service Discovery. To find other peers in the network, the joining peer multicasts a `HELLO` message on the control channel. This indicates that the peer is looking for another peer which can help it join a group. If one or more peers exist on the network able to serve the new peer, these peers reply with a `HELLO_ACK` message unicasted to the joining peer. The message contains information about how to reach the sending peer and also information about which channels the events for the shared objects are propagated on. This enables the new peer to start listening for events on the event channels while the state is synchronized via an existing peer. The joining peer now chooses the peer from which it first receives a reply as its *serving peer*. It is possible to pick any peer replying to the `HELLO` message, as all peers replying will have the same state. The picked peer will with high probability be a peer residing close to the joining peer in terms of network latency, thereby optimizing on network latency overhead in the synchronization of the new peer.

Peer joining. After the service discovery phase, the joining peer will need to synchronize the state between itself and one or more groups. The joining peer may or may not contain state of its own state.

If the joining peer contains no state information, then it needs to obtain the shared state from the serving peer. This is done by a process called 'Just-in-time-eventing' (JITE) where the joining peer first receives a snapshot of the replicated state while collecting events from the other peers during the process. After setting the state of the new peer to the snapshot, the peer also commits the collected events in the correct order [11,29]. If the events were not collected, then the snapshot approach needs to stop any work on the shared object until they were synchronized. This would greatly reduce the responsiveness of the collaborative applications using the infrastructure.

If the joining peer contains state information then the states must somehow be merged. Such a joining peer with state information may be a peer which has been disconnected for a period of time while the user has continued working. The merging or conflict resolution of state based on the local state and the state from the network is highly domain specific. In some cases it makes sense to use the most recent state, in other cases it makes sense to merge the two states, and sometimes the merge is based on the semantics of the application. The joining peer obtains the network state (using the JITE approach) and this state is then given to a conflict resolution method implemented by the application programmer. This enables the programmer to create application specific conflict resolution algorithms.

Synchronous Object State Replication. Synchronous object state replication keeps the replicated objects synchronized, while peers are modifying them. The design should consider basic state change situations, but also be able to handle situations, where two or more peers modify the same component concurrently.

In order to reduce implementation complexity, maximize end-user responsiveness, and minimize communication overhead, the infrastructure utilizes an optimistic concurrency control mechanism based on absolute state events. Event ordering and concurrency control is managed by an extended version of the Lamport clock algorithms [16]. The algorithm uses a logical clock and adds the identity of the sending peer process into the event. Each event is stamped with a timestamp consisting of

`(time,peer,process)` which eliminates the possibility that two events should be stamped with the same logical timestamp. When using this timestamp on each state change event, consistency on fields can be ensured by applying all events with a higher timestamp than the latest committed. If an event is received out of order, the event is simply dismissed. Note that dismissing of events, that is received out of order, will have no influence on the state of the object because only absolute (and not delta) values are sent.

The biggest problem with this design is the case where an event message disappears in the network because of unreliable communication channels. This problem could be eliminated, by using a reliable protocol, but this might imply a huge performance penalty due to the increased communication, as for instance the case with reliable multicast. Sometimes an application may need delivery guarantees and hence pay this penalty, and in other cases the application might not care about reliable delivery but is more focused on speedy delivery. This is precisely the difference between accountable and ephemeral events as introduced earlier and in Bardram et al. [5].

3.2 Peer Architecture

Figure 5 illustrates the peer architecture which consists of three layers. The *application layer* contains the application which is typically programmed according to the model-view-controller pattern. Part of the model uses distributed shared objects, which are located in the *distributed model layer*. This layer contains the distributed part of the model, which consists of distributed objects and nothing else.

The *communication layer* implements the network architecture described in section 3.1 and is responsible for the distribution of state changes to other peers and for managing incoming state changes. This layer is also responsible for the communication between peers holding replicas of distributed objects. This layer keeps track of communication (I/O), event ordering, naming services, and the state of the distributed objects.

Closest to the physical network there are three *I/O Controllers* controlling one form of communication each: TCP unicast, ordinary IP multicast, and reliable multicast. A controller is capable of sending a message to a specified connection point and listening for incoming messages from other devices. The *Communication Controller* manages the I/O controllers and new I/O controllers can be added to support other network protocols.

Management of state change is done by three processes. The *JITE Controller* controls the Just-In-Time-Eventing mechanism explained above. The JITE controller handles a state change event if such an event arrives and no object that corresponds to the event is bound in the naming service. When a new object is created from a remote location, the object is handed to the JITE controller which checks if it contains any events that should be applied to the object. If such events exist they will be applied and the state of the object is up to date. The *Naming Service* is responsible for mapping distributed objects to names and names to distributed objects and it contains methods for looking up an object by name and vice versa. The naming service is used by the distributed object controller. The *Logical Time Tracker* is responsible for keeping track of the logical time by updating the time on both incoming and on outgoing messages.

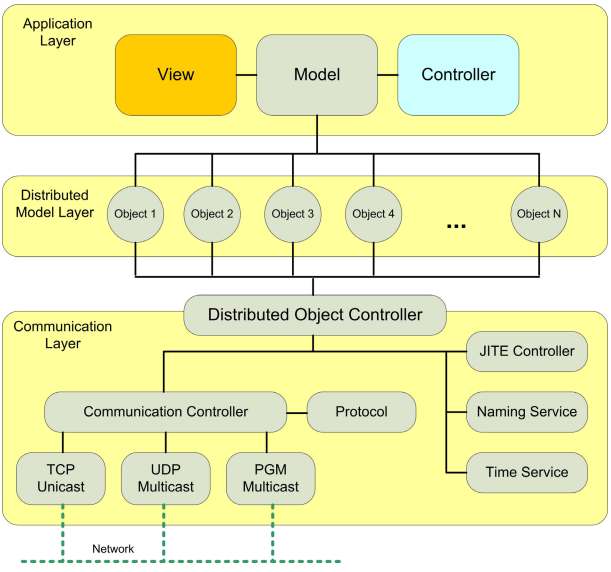

Fig. 5. The architecture of each peer (host) participating in peer-to-peer distributed object sharing. The architecture is divided into three layers: application layer, distributed object model layer, and the communication layer.

The *Distributed Object Controller* works as a facade between distributed objects and the communication layer. When a state change occurs in the distributed objects, the controller will propagate this change to the other participating peers. When a state change arrives from a remote peer, the controller updates the distributed object.

4 Implementation and Evaluation

The infrastructure supporting the proposed PDSOs and a pre-compiler enabling the language support has been implemented. The implementation has been subject to extensive evaluation including completeness of expressiveness, complexity of use, run-time performance, and concept utility. Due to the focus of this paper, presenting the concepts of PDSOs, we shall only present a part of the evaluation focusing on performance and concept utility.

4.1 Performance

Performance evaluation of the DOLCLAN infrastructure has been reported elsewhere [20]. This shows that the infrastructure performs well – both with regard to response time and memory footprint. In this context, we would however like to highlight one particular performance measurement, namely the performance penalty introduced by initiating the propagation of a variable value change.

Figure 6 shows the performance penalty introduced by initiating the propagation of a variable value change. The test measures the time it takes before variable changes have

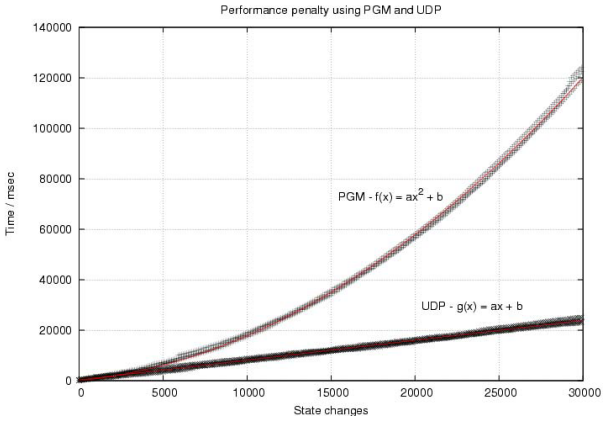

Fig. 6. Performance penalty as a function of number of variable state changes. The graph shows the time it takes to push the state change event into the network asynchronously.

been sent asynchronously into the network. Note that this test does not say anything about the time it takes before the remote peers have received the changes. As reliable multicast protocol we used the Pragmatic General Multicast Protocol (PGM) specified in RFC 3208 [28]. It is clear from the diagram, that there is a significant difference between using reliable and unreliable multicast, even if all communication is done asynchronously. The performance penalty using unreliable multicast has been matched as a linear relationship, while the performance penalty using reliable multicast has been matched with a polynomial relationship. One of the arguments in this paper, is that the application developer should be aware of such differences and have the possibility to make the decisions. This test supports our idea of the need to distinguish between ephemeral and accountable field types.

4.2 Utility

The PDSO concept and the infrastructure described in this paper, has been used to create support for ad-hoc collaboration in the activity-based computing (ABC) framework [5,6]. Previously, the ABC framework was designed according to a client-server architecture and collaboration took place via the activity server. Now, peer-to-peer collaboration can be initiated between two peers with no access to an activity server and activities are replicated on the local peers. This has yielded a higher responsiveness in real-time collaboration and has created support for disconnected work.

In the ABC-framework collaboration is modeled as a number of activities referencing a number of services. The activities represent work tasks and the services represents applications used in the work tasks. To enable ad-hoc collaboration, we used the existing model, but turned the local representation of an activity into a distributed object containing several distributed slots. The same was done with the local representation of a service. This instantly gave us the communication and synchronization between the different participating peers for free.

The effort of extending the ABC framework to support ad-hoc collaboration was limited, counting days rather than weeks or months. Moreover, it showed that the PDSO concepts and the supporting infrastructure are well suited to support the creation of more complex distributed application tasks that just a simple game. The technology is now part of the ABC framework and we are currently creating support for activity-based computing in a hospital setting, by integrating to a Picture, Archiving, and Communication System (PACS) and an Electronic Health Record (EHR). The plan is to deploy the ABC Framework including the distributed shared objects in a hospital. The support for ad-hoc collaboration implemented using the distributed shared objects will enable clinicians to initiate a real-time collaborative session between a surgeon in the operating room and an expert located elsewhere in the hospital.

5 Conclusions

One of the key features of the peer-to-peer distributed shared objects presented in this paper is their support for ad hoc object sharing in loosely coupled networks. The peer-to-peer – or object-to-object – discovery and synchronization makes it simple to create, lookup, and join the distributed objects with their shared data. You can simply look up the object, join it, get a replica, and start to use it as another local object. This indeed makes distributed programming simple while maintaining awareness about the distributed nature of the application.

Furthermore, to support distribution in a pervasive computing environment, the PDSO infrastructure supports intermitted network connections. A peer continues to work while disconnecting and may re-join the network and the PDSOs set of objects later. This applies equally well for smaller network interruptions and for disconnected use. In the former case the user would most likely not even notice the small glitch since all distributed objects are available locally. In the latter case, the user is able to continue working on his local object model and upon reconnect he can re-join the shared network model potentially being involved in some conflict resolution.

The notion of distributed shared objects have been receiving increasing attention because this approach addresses some of the core challenges in existing RPC-based remote method invocation schemes, and it holds the potential to ensure large-scale distribution while ensuring local responsiveness in applications. This paper have suggested one approach to create infrastructure support for such distributed shared objects and should hence be seen as one contribution in this line of research. In our future work we plan to improve on the infrastructure, especially focusing on making support beyond a local area network, and to continue making pervasive computing applications using these distributed shared objects in C#. The latter would also include creating support for e.g. the Pocket PC platform in the .NET compact framework.

Acknowledgments

Jonathan Bunde-Pedersen provided valuable feedback on the ideas and language support presented in this paper. This work is partly funded by the Competence Centre ISIS Katrinebjerg. The ABC project is funded by the Danish Research Council under the NABIIT program.

References

1. Bakker, A., van Steen, M., Tanenbaum, A.S.: From remote objects to physically distributed objects. In: FTDCS '99: Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems, p. 47. IEEE Computer Society, Washington, DC (1999)
2. Bal, H.E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Ruhl, T., Kaashoek, M.F.: Performance evaluation of the orca shared-object system. ACM Trans. Comput. Syst. 16(1), 1–40 (1998)
3. Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S.: Orca: A language for parallel programming of distributed systems. IEEE Trans. Softw. Eng. 18(3), 190–205 (1992)
4. Bal, H.E., Tanenbaum, A.S.: Distributed programming with shared data. In: IEEE CS 1988 International Conference on Computer Languages, pp. 82–91. IEEE Press, Piscataway (1988)
5. Bardram, J.E., Bunde-Pedersen, J., Mogensen, M.: Differentiating between Accountable and Ephemeral Events in the ABC Hybrid Architecture for Activity-Based Collaboration. In: Proceedings of the IEEE International Conference on Collaborative Computing (CollaborateCom 2005), pp. 168–176. IEEE Press, Orlando, Florida (2005)
6. Bardram, J.E., Bunde-Pedersen, J., Soegaard, M.: Support for activity-based computing in a personal computing operating system. In: CHI '06: Proceedings of the SIGCHI conference on Human factors in computing systems (To appear), ACM Press, New York (2006)
7. Beaudouin-Lafon, M., (ed.): Computer Supported Cooperative Work. John Wiley and Sons, New York (1999)
8. Birrell, A.D., Nelson, B.J.: Implementing remote procedure calls. ACM Trans. Comput. Syst. 2(1), 39–59 (1984)
9. Cugola, G., Picco, G.: Peerware: Core middleware support for peer-to-peer and mobile systems (2001)
10. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
11. Geyer, W., Vogel, J., Cheng, L.-T., Muller, M.: Supporting activity-centric collaboration through peer-to-peer shared objects. In: GROUP '03: Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work, pp. 115–124. ACM Press, New York (2003)
12. Greenberg, S., Roseman, M.: Groupware toolkits for synchronous work. In: Beaudouin-Lafon [7], pp. 135–168
13. Hagimont, D., Boyer, F.: A configurable rmi mechanism for sharing distributed java objects. IEEE Internet Computing 5(1), 36–43 (2001)
14. Hill, J., Gutwin, C.: The MAUI Toolkit: Groupware Widgets for Group Awareness. Computer Supported Cooperative Work 13(2), 539–571 (2004)
15. Homburg, P., van Steen, M., Tanenbaum, A.S.: An architecture for a wide area distributed system. In: EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop, pp. 75–82. ACM Press, New York (1996)
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
17. Lipkind, I., Pechtchanski, I., Karamcheti, V.: Object views: language support for intelligent object caching in parallel and distributed computations. In: OOPSLA '99. Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 447–460. ACM Press, New York (1999)
18. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: PLDI '88. Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, pp. 260–267. ACM Press, New York (1988)

19. Matsuoka, S., Kawai, S.: Using tuple space communication in distributed object-oriented languages. In: OOPSLA '88. Conference proceedings on Object-oriented programming systems, languages and applications, pp. 276–284. ACM Press, New York (1988)
20. Mogensen, M.: Distributed objects in loose coupled local area networks. Technical Report, Computer Science Department, University of Aarhus (2005)
21. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The information bus: an architecture for extensible distributed systems. In: SOSP '93. Proceedings of the fourteenth ACM symposium on Operating systems principles, pp. 58–68. ACM Press, New York (1993)
22. OMG. Corba services: Common object services specification, chapter 4: Event service (March 2001)
23. Picco, G.P., Murphy, A.L., Roman, G.-C.: LIME: Linda meets mobility. In: International Conference on Software Engineering, pp. 368–377 (1999)
24. Prakash, A.: Group editors. In: Beaudouin-Lafon [7], pp. 103–134
25. Roseman, M., Greenberg, S.: Building real-time groupware with groupkit, a groupware toolkit. ACM Trans. Comput.-Hum. Interact. 3(1), 66–106 (1996)
26. Shim, H.S., Hall, R.W., Prakash, A., Jahanian, F.: Providing Flexible Services for Managing Shared State in Collaborative Systems. In: Rodden, T., Hughes, J., Schmidtk, K. (eds.) Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, Lancaster, UK, pp. 237–252. Kluwer Academic Publishers, Boston (1997)
27. Smith, D.A., Kay, A., Raab, A., Reed, D.P.: Croquet - a collaboration system architecture. In: C5 2003. Proceedings. First Conference on Creating, Connecting and Collaborating Through Computing, pp. 2–9. IEEE Press, New York (2003)
28. Speakman, T., Crowcroft, J., Gemmell, J., Farinacci, D., Lin, S., Leshchiner, D., Luby, M., Montgomery, T., Rizzo, L., Tweedly, A., Bhaskar, N., Edmonstone, R., Sumanasekera, R., Vicisano, L.: PGM Reliable Transport Protocol Specification. RFC 3208 (Experimental) (December 2001)
29. Vogel, J., Geyer, W., Cheng, L.-T., Muller, M.J.: Consistency control for synchronous and asynchronous collaboration based on shared objects and activities. Computer Supported Cooperative Work 13(5-6), 573–602 (2004)
30. Yonezawa, A., Briot, J.-P., Shibayama, E.: Object-oriented concurrent programming abcl/1. In: OOPLSA '86. Conference proceedings on Object-oriented programming systems, languages and applications, pp. 258–268. ACM Press, New York (1986)