Using Live Distributed Objects for Office Automation

Jong Hoon Ahnn, Ken Birman, Krzysztof Ostrowski, Robbert Van Renesse
Department of Computer Science, Cornell University, Ithaca, NY 14850, USA
{ja275, ken, krzys, rvr}@cs.cornell.edu

ABSTRACT

Web services and platforms such as .NET make it easy to integrate interactive end-user applications with backend services. However, it remains hard to build collaborative applications in which information is shared within teams. This paper introduces a new drag-and-drop technology, in which standard office documents (spreadsheets, databases, etc.) are interconnected with event-driven middleware ("live distributed objects"), to create distributed applications in which changes to underlying data propagate quickly to downstream applications. Information is replicated in a consistent manner, making it easy for team members to share updates and to coordinate their actions. We present our middleware platform, and show that it offers good performance and scalability, with small resource footprint. Moreover, because the approach is highly automated, and the underlying middleware is highly configurable, we're in a position to automatically address security and reliability needs that might otherwise be onerous. In addition to reviewing our existing system, we list open issues, which include integration with external data sources, and updating stored, but inactive objects.

Categories and Subject Descriptors

D.3.3 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems – *Distributed applications.*

General Terms

Performance, Design.

Keywords

Distributed systems, Live distributed objects, Middleware, Office automation, Office information systems, SOA.

1. INTRODUCTION

Since the 1970s, enterprises have experienced a paperless office automation (OA) revolution, a trend now accelerating as web services gain wide acceptance [1]. Yet it remains surprisingly hard to build office applications in which end-users track dynamically changing data, such as databases that reflect inventory or task status or spreadsheets that summarize financials, and even harder to build collaborative applications in which team members cooperate to solve office tasks. The premise of our work is that empowering users to directly create distributed office applications, much as they create office or web documents today, would open the door to productivity advances. Moreover, encouraging end-users to express intent in a high-level form makes it possible to automatically verify that sensitive data is transmitted over encrypted communication channels that critical services run on highly available platforms.

In this work, we report on a distributed office information system (OIS) developed to support office employees and organizations. With our OIS, office workers can design data pipelines, in which workflow events that update databases or spreadsheets can be shared throughout an enterprise in a simple and seamless way. Users interact with the OIS via a drag and drop interface, and although they can also write code, the scheme is powerful enough to allow even non-programmers to build very sophisticated collaborative applications. We are not the first to pursue this direction; prior approaches include goal-based agent systems and intelligent agent-based workflow systems [9]. However, we are not aware of any prior work offering the same benefits. The contributions of this paper are as follows.

- We describe a new "live distributed objects" programming model and show how it can be applied in office automation settings. Although we have published on the basic concept and platform [7, 8], our earlier paper focused on a virtual reality application. This paper is the first to explore integration of this technology with databases and office automation, a scenario posing new questions.
- We present the OIS integration tools in our platform, and discuss the challenges we faced in implementing them. Our prototype system is powerful, but is just a prototype. Some questions remain open, and we also review these.
- Our system incorporates type-checking and reflection mechanisms. Our prototype uses these mostly to prevent users from making mistakes. Down the road, however, reflection-driven coercions could automatically secure sensitive data and ensure that critical components run in a highly-available manner.
- We evaluate performance in free-standing configurations, and look at GUI costs using a methodology recommended by SAP [15].

Researchers both in academia and industry have been interested in using middleware technologies to support office information systems (OIS) [1, 2, 3, 4, 12, 16] since the 1970s. Our system integrates office applications into a componentized event notification framework at the end-user level. For example, if a spreadsheet cell is linked to a live object notification channel, changes in that cell will be propagated to other spreadsheets or databases associated with the live object. The effect is to create a mash-up in which office workers (non-programmers) directly express the manner in which they plan to share information. This paper focuses on issues specific to OIS applications; other aspects of the systems are discussed in [3, 6, 7, 8].

Middleware'08 Companion, December 1-5, 2008 Leuven, Belgium.

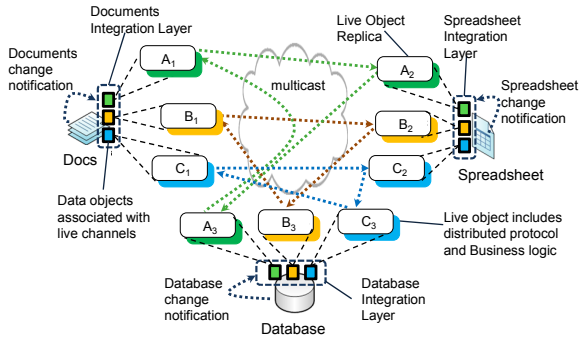

Figure 1. Live object replicas embedded in the documents and the database wrapper.

2. SYSTEM ARCHITECTURE

Figure 1 presents an architectural overview of our system. In this section, we start by summarizing the assumptions underlying the platform, and then review the architecture. The subsections that follow provide details on some of the key functionality, including the live objects platform itself, reliable message delivery and event notification it uses, and the forms of office applications currently supported by our system.

2.1 Requirements

Automated OIS systems used in distributed environments must satisfy several properties [5, 9, 17]. Some of these needs are common to many kinds of systems. For example, OIS systems are highly concurrent and new to be as autonomous as possible, automatically resolving contention when multiple users access documents concurrently, providing interoperability between different office applications and services. However, collaboration applications create unique issues. In such settings, teams of users will often share documents that need some way to reflect changing events within the enterprise. Our approach is to allow office workers to create new kinds of mash-up applications by dragging and dropping dynamically changing data from databases into other sorts of office documents (we'll focus on spreadsheets but can support all sorts of documents), and by sharing information changed within those office documents among the team members. We adopt a fine-grained approach: we replicate and share information at the level of individual office objects, such as individual spreadsheet cells (or rows, or columns), figures in shared documents (which could capture data from sensors or video sources), etc. These "live documents" can then be shared just like any normal document, through file systems or email.

A mash-up is a graph of components, in which simpler applications, data sources, services, and reports are interconnected by event-notification pathways. In such an application, the failure of one component might ripple throughout the system, disrupting all sorts of downstream activities. Moreover, if a component working with sensitive data fails, applications that depend on its output might be tainted. Ideally, one would want to consider fault-tolerance and security issues from the outset. Yet in OIS settings, applications often evolve over time as new needs arise, and these evolutionary events can create new security or availability needs not present in early versions of a system. A strength of our approach is that applications are represented in a high level form.

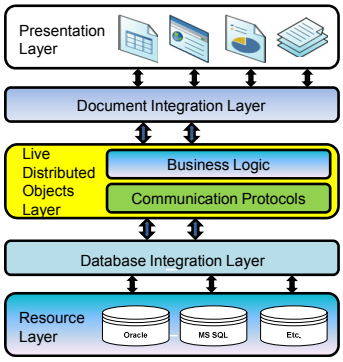

Figure 2. Middleware layers for the office information system.

This makes it possible to automate many of these tasks, in a manner driven by the component-level type system underlying our live objects platform.

2.2 Overview of the Integration Framework

Our overall architecture is depicted on Figure 2. The Resource Layer consists of database systems such as Oracle and MSSQL, residing on servers within the enterprise network. To interface database resources to the live objects layer we use the Database Integration Layer, a wrapper that represents database views and spreadsheet cells as live distributed objects. The underlying data is represented as serialized OLE objects and hence is compatible with a wide range of office applications such as word documents, power-point presentations, spreadsheets. In these sorts of "presentation" documents, the Document Integration Layer allows us to link low-level office objects such as text boxes, rectangles, pictures, or video clips to live object channels. Finally, the Presentation Layer is a set of wrappers that lets us embed these primitive live office objects into documents, web sites, and others.

2.3 Live Distributed Objects

Although brevity prevents a detailed discussion of live objects, we'll give a mile-high summary of the model [7]. Live objects are componentized representations of distributed protocols, such as reliable multicast channels. When activate, these protocols are run by live object *replicas*, which the live objects middleware platform dynamically constructs using recipes expressed in XML. We leverage this option to connect the world of live objects to local instances of databases or spreadsheets.

2.4 Reliable Message Delivery and Scalable Event Notification

The OA platform requires a reliable, totally-ordered event notification infrastructure. We discuss the available communication options in Section 5. The particular choice of the transport is not important: live objects decouple the transport from the OA layer via a strongly-typed interface, and we can easily replace the underlying transport infrastructure to use different protocols without any changes to our OA infrastructure.

2.5 Office Data Types

We have emphasized that live objects are strongly-typed. This was visible in our channel example, but type checking is actually

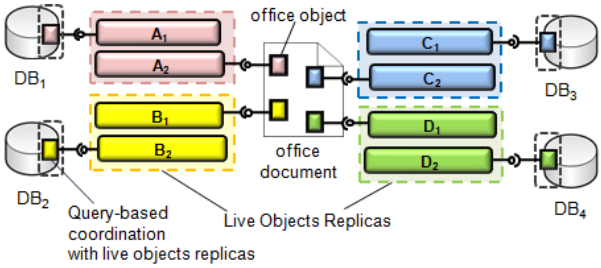

Figure 3. Live objects can be used to integrate multiple databases, or to connect databases with other kind of office objects such as spreadsheets or documents.

used throughout the live objects platform [7], and is used to describe data formats as well as the behavioral properties of the protocols. For instance, any visual element dragged into a chat window has a distributed type that specifies how one can interact with the element, or what services it may provide. The user can define custom event types and live object types as .NET interfaces annotated with descriptive attributes. The live objects runtime dynamically loads new type and component libraries, scans them for annotated interfaces and classes. The .NET code and annotations bootstrap this distributed type system [7].

3. INFORMATION FLOW

Our use of live objects is currently focused on two cases. In one, an information flow originates from a monitored database view. In the second, a live office object (such as a spreadsheet cell or an image) triggers updates. In both cases, other office documents that import the live office object will be updated immediately when a change occurs. Notice that the granularity of replication is rather fine-grained: we're not replicating entire documents or spreadsheets, just individual cells or other office objects such as embedded images or video streams.

For the first case, we leverage a database feature called a *materialized view* to let the application designer select information that will be shared in this manner. Such a view is associated with a query, and automatically recomputed each time the underlying database is updated. With our platform, whenever the materialized view changes, a new event is delivered into the live object replica running on the database server. The update can then be imported into documents such as spreadsheets, other databases, or other kinds of applications shown in Figure 3. The second case is similar: we monitor the contents of a designated live office object, such as a cell of a spreadsheet, watching for changes. We then serialize the contents of the object and generate an event into the associated multicast stream.

We have found it convenient to associate colors with live objects; in the case of a spreadsheet, a live cell acquires the color of the underlying object. This helps the developer, typically a non-programmer, track the different communication options.

Our discussion implicitly reflects rather simple pipelines, where data from databases is pulled into documents, without additional processing. However, more elaborate pipelines can easily arise in OA settings, where a single event may trigger multiple, perhaps independent, chains of reaction. Our evaluation, in section 5.1, focuses on the three scenarios shown below. To facilitate measurement of latencies, each of the cases we consider includes a looped-back link from terminal nodes to the update source, although in practice that last link wouldn't be used.

4. IMPLEMENTATION

In this section, we discuss the architecture of individual layers of our system, focusing on the *Document Integration Layer* (DIL) and *Database Integration Layer* (DBIL).

The DIL leverages Microsoft's Object Linking and Embedding (OLE) technology. OLE enables elements of a compound document to communicate with one another via COM interfaces, and also standardized data representations within the Microsoft office product suite [11]. Our DIL leverages these interfaces but uses only a subset of OLE, concerned with persisting object metadata within a file on the disk. OLE objects thus serve simply as wrappers that provide persistence to the embedded live objects. Application events are relayed by OLE to the live office objects platform. Application events are converted into live object messages, and vice versa. With this approach, spreadsheets, Microsoft Word documents, and other general-purpose office applications and legacy systems can be linked with the live objects framework to replicate events that update the office object, connect it to digital cameras or sensors.

Our second middleware integration layer, DBIL, is used only with databases. Again, rather than replicating the underlying database (not a useful functionality, since most database products offer vendor-supported replication solutions), our focus is on relaying database events into the live objects framework, which multicasts them to subscribers, such as spreadsheets or other office documents. As summarized earlier, the basic idea is to register a query, which is reevaluated each time the underlying database is updated, computing a new dynamically materialized view, and passing an event to a live object. The technology is very easy to use, and requires no programming skills beyond the ability to compose a query. More details are given below.

Both technologies are accessed through a GUI that we describe in Section 4.1. Startup costs and serialization are covered in Section 4.2. Event notification is discussed in Section 4.3.

4.1 Graphical User Interface (GUI)

At design time, developers (who will often lack programming skills) work primarily through the live objects GUI interface. The DIL is accessed through the two dialog windows shown in Figure 4. The *Import Channel Dialog* (upper) allows the user to import a live object file describing a communication channel. The *Connect a Cell To a Channel Dialog* (lower) can then be used to associate the channel with a specific cell. After selecting the cell, users are presented with a drop-down menu, to select one of available live objects compatible with the office automation logic. When the *Connect* button is pressed, the connection is established, and the values of the cell are synchronized with other connected cells in live documents across the network. The DBIL GUI allows the user to bind a dynamically materialized view to a live object, as shown in Figure 4b. The developer registers a triggered callback, then links the monitored relation to a live object channel. Each time an update occurs, the query is recomputed and if output has changed, a new event will be generated containing the monitored relation. A pre-recorded demo of the whole process can be seen on our web site [10].

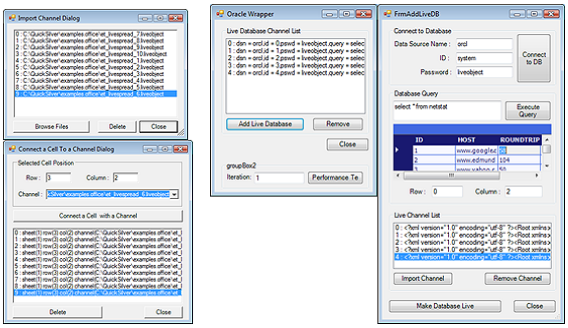

Figure 4. (a) GUIs for the document integration layer (left two). (b) GUIs for the database integration layer (right two).

4.2 Startup Costs and Serialization

Opening a live document entails initializing the communication subsystem and fetching the current version of any replicated data. There are two scenarios: (1) The document is opened, and there are no other replicas already active. In this case, office objects embedded within the document load stored values from the saved document file, (2) Other replicas are active. Here, after loading the document, we fetch the current state of each office object from one of the existing document replicas on other machines.

The live objects platform automatically determines which scenario applies. The platform is itself bootstrapped by using macros in Visual Basic script embedded in a document. These macros intercept events such as opening, closing, and saving the document, changes to spreadsheet cells. The same module is used, but different methods are invoked by the DIL. As mentioned earlier, DBIL is used only to capture events from a database, not for database replication. DBIL uses a similar serialization interface. The mechanism is even simpler, and the code is similar; we omit the details for brevity.

4.3 Data Change Event Notification

Next, we turn to the question of detecting data change events in office objects such as spreadsheets. In the DIL each live object embedded within a document is wrapped in an OLE object. The DIL intercepts events that report changes to any of the underlying objects, for example, when the user types something into a cell, passes them to the embedded live objects to propagate, and then applies the updates to all document replicas. For this purpose, we use a .NET framework feature that allows us to define handlers for Component Object Model events. We filter out OLE events, multicast them, and then deliver them in a distributed manner. For simplicity, in the following discussion we ignore details such as detecting which cell has changed, and we'll just assume that an entire worksheet has been made live.

The database solution is similar. To capture database changes, DBIL uses a feature of ADO.NET 2.0, whereby a *database change notification* event in the data access layer allows a .NET application to be notified whenever the server data it consumes is changed. This feature is supported by most products (our prototype was tested with Oracle Database 10g Release 2 [14] and MS SQL Server 2005 [13]).

In general, we would register the user's query, and monitor the result – the dynamically materialized view mentioned earlier. When a database table is updated, our event handler is invoked. We can now capture any updates and publish them to components embedded in live documents. OLE serialization is a broadly supported office standard, hence databases and spreadsheets can be connected without additional wrapper code.

5. PERFORMANCE EVALUATION

Our performance evaluation focuses on two metrics: response time and throughput. In this section, we'll measure the overhead associated with each of the three layers in our architecture: presentation, integration, and resource/DB. We'll also look at the network load.

The live objects platform currently provides two types of communication channels: one uses a simple TCP-based application level multicast (ALM) substrate, and the other using Quicksilver Scalable Multicast [6], and based on IP multicast (with more options expected later in 2008). For the purposes of this evaluation, we use TCP-based channels to mimic SOA eventing architectures, in which senders maintain TCP connections to receivers (WS-eventing standards assume this sort of architecture, hence even though our simple ALM isn't fully WS-* compliant, the performance seen here is similar to what we would expect when working with a commercial WS-* product that runs over TCP). Communication overhead in this model increases linearly as a function of the number of document replicas. Were we to use QSM, which is highly scalable, overheads would rise much more slowly. In fact, we doubt that extreme scalability will be in an issue in OA settings. In most anticipated use scenarios, individual office documents would be accessed by just a few users at a time. A comprehensive discussion of throughput and scalability of the multicast substrates provided by the live objects platform can be found in [6].

The evaluation presented below focuses on resource footprint and latency, the two metrics most relevant to performance in our layered OIS architecture. The experiments reported here use a cluster of 22 nodes, with Pentium III 1.3 GHz CPUs, 512 MB memory, on a 100 Mbps LAN, running Microsoft Windows Server 2003 Enterprise Edition SP2. One machine is dedicated to running the Oracle database server. Another machine serves as a controller for the TCP-based multicast substrate. The remaining 20 nodes act as clients accessing our replicated office documents. We show overheads involved in opening live documents, data change event notification, and evaluate various topologies in terms of efficiency, concurrency, and scalability in Section 5.1. We summarize the results in Section 5.2.

5.1 Performance Measurement

5.1.1 Information Flow (Event Pipelining)

Live office applications will often be structured into pipelined processing configurations, with each application passing data to another office application that does some processing and then generates its own change events for propagation further downstream. To understand performance in such cases, we evaluated three topologies: a ring, binary tree and ternary tree in 20 nodes of our cluster. All of these "loop" the events back to the source, as a simple way to measure end-to-end latency without

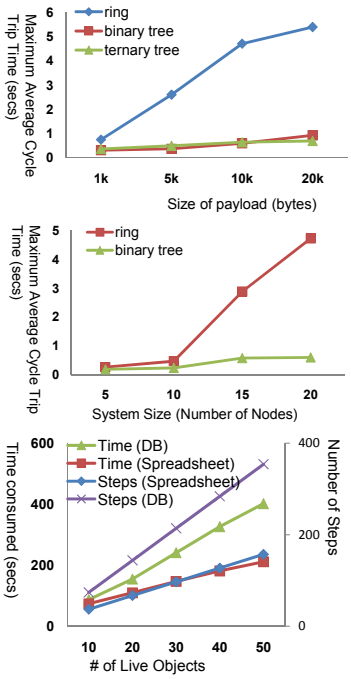

Figure 5. (a) Maximum average cycle trip time when size of payload varies under various topologies. (b) Maximum average cycle trip. (c) Efficiency : time consumed and Number of Steps when the number of live objects grows.

worrying about clock synchronization. As will be seen below, costs are sufficiently high for these kinds of office application pipelines that the extra load imposed on the root node has no significant impact on the overall picture.

Cycle trip time (CTT) grows as a function both of payload size and path length, as seen in Figures 5a-b. The payload costs are dominated by OLE's XML serialization and deserialization overheads; as the paths grow longer and these are incurred again and again, they become significant. The overall picture suggests that the system reasonably stable and offers reasonable performance, although the costs associated with the Oracle configuration are fairly high. We attribute this to costs of the Oracle dynamic view materialization and to the overheads associated with the trigger mechanism.

5.1.2 Efficiency

Performance evaluations of commercial OA products, such as SAP's product line, typically evaluate the productivity of application developers by measuring the time needed to build a new application. To evaluate this in our OIS, we designed an experiment patterned on SAP's standard benchmarks. Table I lists the steps needed to construct a completely new collaborative application, using either the DIL or the DBIL. We asked how much time a trained end-user would expend in performing these steps. We assume all necessary applications are pre-installed in the system. Since user input times vary widely, we performed the sequence of operations 20 times and averaged the result.

For example, we created a sales report which needs 40 data

Table 1. User Interaction Steps for doc. integration layer

Step	Instruction (repeat steps 6 through 9)
1	Launch a spreadsheet.
2	Start an Import Channel dialog.
3	Import all live objects into the dialog by a drag/drop interface or a file browsing dialog.
4	Close the dialog.
5	Start a Channel Coordination dialog.
6	Select a cell to be connected to live object in the spreadsheet.
7	Choose a live object file already imported in a drop-down box.
8	Press a Connection button.
9	Close the dialog.
10	Save the spreadsheet. Close the spreadsheet.

fields connected to live objects, associated with 40 live channels that we connected to database fields based on queries. Setting up the 40 live spreadsheet objects took 120.7 seconds and 127 steps; 40 live databases objects took 326 seconds and 284 steps shown in Figure 5c. Overall, our sales report was constructed in 447 seconds and required a total of 411 user-input steps. This seems quite reasonable to us. Once built, such an application would be shared through the network file system, or by email. The eventual users pay no additional costs at all: they simply open the application and use it.

5.1.3 Scalability

Scalability of the OIS is important criteria when considering business applications in distributed domains. Users will need to know that our system is capable of handling existing transactional workloads while continuing to maintain performance even as the workloads increase significantly. Our experiments shed light on this question. The first experiment shown in Figure 6a forms a ring topology and measures cycle time as we vary the number of participating nodes, forcing the nodes to share updates using a single multicast channel. This results in an extreme case because as many as 1530 live objects ultimately communicate through one channel. The CTT rises starting when more than 200 live objects run (20 nodes with 10 live objects running on each machine) because of channel contention.

In figure 6b varying numbers of live objects communicate side by side with different multicast channels. Recall that these experiments run over a WS-notification layer that uses point-to-point TCP connections, hence each of these channels requires its own set of TCP channels. With 20 nodes running 10 live objects each, one would need approximately 4000 TCP connections. In fact, performance degrades sharply even before we reach that scale. In practice, we believe that only smaller applications would be able to operate over TCP; scaled up applications such as these larger configurations would need to migrate to Quicksilver, to exploit its much better scalability properties. Earlier, we commented that our TCP-based ALM is intended to conform with WS-* eventing standards, whereas Quicksilver, which uses IP multicast to disseminate data quickly, deviates from those standards. Our tests make it clear that live distributed objects can be used in OA settings, but that rigid adherence to the WS-* standards results in scalability limits that could be avoided if the WS-* standards were more flexible. In a different setting, we dis-

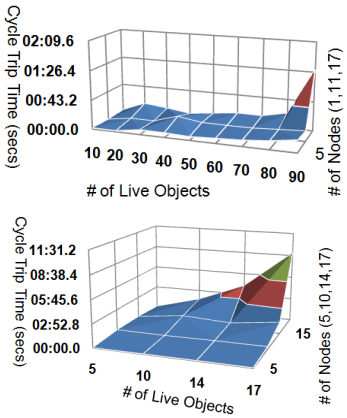

Figure 6. (a) CTT in a channel. (b) CTT in various channels.

cuss possible extensions to the standards aimed at this issue.

5.2 Discussion

In summary, we have seen that resource consumption is roughly linear in the number of live objects used within a live office document while activating a live document, and that subsequent event processing time of our system is easily predictable as the number of live objects grows. We were somewhat disappointed that Oracle performance was so low, but this reflects performance issues within Oracle itself, presumably related to the way it implements materialized views and triggers. In the information flow section, we explored a variety of flow topologies and showed that live objects can be used in pipelines; the pipeline length was the main performance-limiting factor. Finally, we evaluated performance in situations that stress the TCP-based multicast channels and showed that they work well for smaller configurations, but degrade sharply with scale; in production settings that employ large numbers of inter-connected live documents, users would need to employ a scalable substrate, such as Quicksilver.

A number of issues lie beyond the scope of this paper. Future challenges include implementing security and privacy, and in particular, integration of the OIS and the live objects platform with existing security infrastructure, such as Active Directory, X.509 certification and other such services. We are also working on adapting our plat-form for use in WAN settings. Beyond these near term issues lie hard questions associated with supporting transactions and dealing with enterprise life-cycle management.

6. CONCLUSION

This paper described the design and implementation of middleware architecture for office information systems. The design builds upon a concept we call live distributed objects, adapting them to office automation settings. This yields a new style of live office documents, in which office applications and replication technologies are cleanly integrated. We have created a visual drag and drop environment, in which end-users with little or no programming ability can create distributed applications by leveraging existing documents and databases. Our evaluation shows that the system is very easy to use, performs well, and scales well in realistic LAN settings.

7. ACKNOWLEDGEMENT

Our work was funded by AFRL/IF, AFOSR, NSF, I3P and Intel. We'd like to thank Mahesh Balakrishnan, Lakshmi Ganesh, Chi Ho, Maya Haridasan, Tudor Marian, Yee Jiun Song, Einar Vollset, Hakim Weatherspoon, and Eric Suss for the feedback.

8. REFERENCES

[1] D. E. Mahling, N. Craven, W. Bruce Croft. From Office Automation to Intelligent Workflow Systems. IEEE Expert: Intelligent Systems and Their Applications, vol. 10, no. 3, pp. 41-47, Jun., 1995.

[2] ELLIS, C., NUTT, G. Office information systems and computer science. ACM Comput. Surv. 12, 1 (Mar. 1980), 27-60.

[3] G. Bracchi, B. Pernici. The design requirements of office systems. ACM Transactions on Information Systems (TOIS), vol. 2 no. 2, pp. 151-170, April 1984.

[4] Google. Google Docs. http://documents.google.com

[5] Jammes, F.; Smit, H.; Service-Oriented Paradigms in Industrial Automation. IEEE Trans. on Industrial Informatics, 2005, vol. 1, no. 1, pp. 62-70.

[6] K. Birman, M. Balakrishnan, D. Dolev, T. Marian, K. Ostrowski, A. Phanishayee. Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India, January 7-12, 2007.

[7] K. Ostrowski, K. Birman, D. Dolev, J. Ahnn. Programming with Live Distributed Objects. In Proceedings of 22nd ECOOP, Jul., 2008.

[8] K. Ostrowski, K. Birman, and D. Dolev. Live Distributed Objects: Enabling the Active Web. IEEE Internet Computing, vol. 11, no. 6, pp. 72-78, Nov/Dec, 2007.

[9] K. Park, J. Kim, S. Park. Goal based agent-oriented software modeling. Proceedings of the Seventh Asia-Pacific Software Engineering Conference, pp.320, December 05-08, 2000.

[10] Live Objects at Cornell. http://liveobjects.cs.cornell.edu.

[11] Microsoft Corporation (December 1993). OLE 2 Programmer's Reference: Creating Programmable Applications with OLE Automation. vol. 2, Programmer's Reference Library, Microsoft Press.

[12] Microsoft Office Groove. http://office.microsoft.com.

[13] Microsoft SQL Database. http://www.microsoft.com/sql.

[14] Oracle Database. http://www.oracle.com/database.

[15] SAP. SAP Standard Benchmark https://www.sdn.sap.com.

[16] SIRBU, M., SCHOICHET, S., KUNIN, J., HAMMER, M. OAM: An office analysis methodology. MIT, Office Automation Group Memo OAM-016, 1981.

[17] U. Dayal , M. Hsu , R. Ladin. Business Process Coordination: State of the Art, Trends, and Open Issues. Proceedings of the 27th International Conference on Very Large Data Bases, pp. 3-13, September 11-14, 2001.