Using Live Distributed Objects for Office Automation

Jong Hoon Ahnn, Ken Birman, Krzysztof Ostrowski, and 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.

Keywords: Live distributed objects, Office automation, Office information systems, SOA, Distributed systems, Middleware

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, etc.

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're not the first to pursue this direction; prior approaches include goal-based agent systems and intelligent agent-based workflow systems [2]. However, we're 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've published on the basic concept and platform [6], 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 [13].

1.1 Related Work

Since the 1970's, researchers both in academia and industry have been interested in using middleware technologies to support office information systems (OIS) [1]. Typically, such work starts with some formal description of office objects in a modeling language [40]. OFFICETALK-ZERO [33], OMEGA [34], OFFIS [35], and OBE [36] are examples of systems that start with information stored in a database and assemble it into various forms suitable for use in office settings. Office tasks are commonly described using process-based models, an approach explored in systems such as OAM [39] and Ticom-II [38]. Structural Model [37] describes office tasks using agents. These approaches can also be combined, as was done in OFS [31], IML [32], and OPAS [22], Semantic Models [23], OFFICETALK-D [24], and SOS [25].

In contrast, 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 [5, 6, 12, 21].

Mash-ups are common in web applications. Such applications commonly use the web services architecture, which our work also supports. However, most existing web services technologies are centralized, with the services running on data centers (for example in Web Office [43], Google Docs [40], and Microsoft Office Groove [41]). In contrast, our platform can support applications that are peer-to-peer in flavor, and where communication occurrs directly between collaborating users. Jini [26] and

JXTA [27] are examples of existing technologies for building peer-to-peer applications. In contrast to our work, both target programmers. Our hope is to largely eliminate programming, allowing complex collaborative systems to be constructed in a drag-and-drop style, with code written only in unusual situations. When coding is needed, any of the 40 or so languages supported by .NET can be used; Jini and JXTA are both oriented towards Java.

Our work has obvious connections to event bus architectures. For example, IBM's Enterprise Service Bus (ESB) [28] and TIBCO's information bus [9] are positioned as enterprise integration solutions. While either could support a vision like the one we express here, neither operates at the end-user level, and neither explores scenarios in which applications are constructed as graphs with multiple such applications connected by event notification channels – a collaboration model more evocative of data replication than of publish-subscribe.

Looking to the future, other kinds of prior work will become relevant. For example, our live objects platform currently lacks components that can support persistence, or manage resource allocation. Yet office documents are commonly stored, and the average application might be in an inactive state, on a disk, much of the time, perhaps with replicas at multiple locations. Dealing with persistence thus stands as a task for future work; when we tackle it, we'll need to understand how transactions (both in the ACID sense, and in the sense of business transactions) can fit into our overall vision, and to track and reconcile distinct versions. Our hope is that by building on the live objects platform, which is itself componentized and easily extensible, it will be possible to incrementally extend the basic OIS over time to address such issues, ideally by encapsulating existing technologies as new kinds of live distributed objects.

2 System Architecture

Figure 1a 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 for Office Information Systems

Automated OIS systems used in distributed environments must satisfy several properties [7, 8, 10, 11]. 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, etc. 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. 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.

For example, suppose that a database application generates events that need to be shared in a highly available manner, and that contain private data. If constraints are made explicit, our platform can detect these requirements by inspection of the components. By expressing goals at a high level, and then using a type system to detect requirements, we can potentially move away from a dependence on programmer skills and towards automated enforcement of critical requirements. This is enabled by a compositional type system with reflection properties: when two components are connected, there are opportunities to inspect their strongly-typed interfaces, to object if they are incompatible, and otherwise to inject additional "connective" components if needed (in effect, to perform automated type coercion).

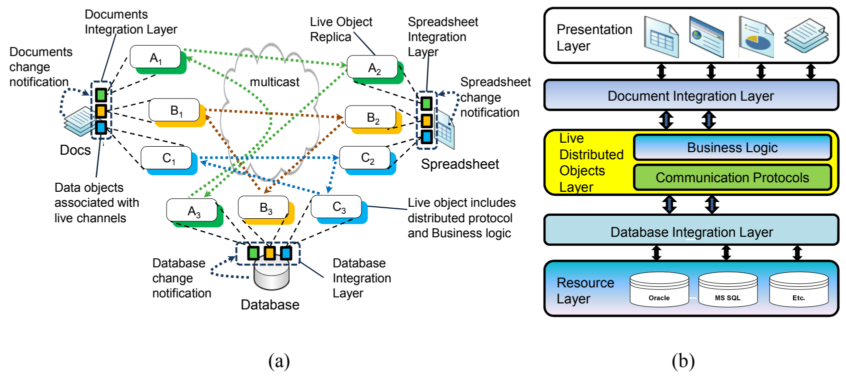

Documents Integration Layer
Documents change notification
Docs
Data objects associated with live channels
A_1
B_1
C_1
multicast
Live Object Replica
Spreadsheet Integration Layer
A_2
B_2
C_2
Spreadsheet change notification
Spreadsheet
A_3
B_3
C_3
Live object includes distributed protocol and Business logic
Database change notification
Database
Database Integration Layer
(a)
Presentation Layer
Document Integration Layer
Live Distributed Objects Layer
Business Logic
Communication Protocols
Database Integration Layer
Resource Layer
Oracle
MS SQL
Etc.
(b)

Fig. 1. (a) Live object replicas embedded in the documents and the database wrapper; (b) Middleware layers for the OIS.

2.2 Overview of the Integration Framework

Our overall architecture is depicted on Figure 1b. The ***Resource Layer*** consists of database systems such as Oracle and MSSQL, residing on servers within the enterprise network. To interface database respources to the live objects layer we use the ***Database Integration Layer*** (DBIL), 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, powerpoint, spreadsheets, etc. In these sorts of "presentation" documents, the ***Document Integration Layer*** (DIL) 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, etc.

2.3 Live Distributed Objects

Although brevity prevents a detailed discussion of live objects, we'll give a mile-high summary of the model [6]. 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. The recipes contain information about the type of a protocol, and are designed to support drag and drop composition, creating graphs: the mashups mentioned earlier. Each object replica consists of some event handling code to run (we can download it from a remote repository, JIT it and link it dynamically). A replica can contain local state, and accepts parameters. Many replicates coordinate their states with other replicas of the same object, but a live object replica can also interface to local resources or applications. We leverage this option to connect the world of live objects to local instances of databases or spreadsheets.

```
<Object xsi:type="ReferenceObject"
   id="{F73B571836E24614A968DE2F15092088}">
 <Parameter id="color">
   <Value xsi:type="xsd:string">Red</Value></Parameter>
 <Parameter id="channel"><Value xsi:type= "Reference-
    Object" id="{10000000000000000000000000000020}">
       // details omitted for brevity
   </Value></Parameter>
</Object>
```

The above example shows the XML recipe for a simple live object (it would be stored in a file with a "**.liveobject**" extension). This particular object is associated with a live spreadsheet; it references another object (the underlying communication channel that will be used) and specifies that spreadsheet cells linked to the object be colored red. The reference provides the live objects platform with enough information to download the needed event handling code (we'll see examples in a moment), check event channel types for compatibility, link them together, and thus activate the mash-up. An application thus consists of a graph of components – with the whole graph replicated at each machine where the application is running. When active, this live distributed object colors the spreadsheet cell to which it is attached red, and replicates the contents relative to other spreadsheets linked to the same cell. We should stress that the end-user never sees this code: it was generated automatically.

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.

The following example illustrates the interface used to bind to a event channel.

```
01 void ICheckpointedChannelClient<IMyType, IMyType>.Receive
02   (IMyType message) {
03   this._data = message._data;                          }
04 void Data_Change_Event(Range target) {
05   _endpoint.Interface.Send(new MType(target._data));     }
```

When a mashup is activated, the communication endpoints associated with the underlying objects are linked together and the corresponding code is downloaded, JITed, and launched [6]. The **Receive** function (line 01) handles incoming events of type ***IMyType*** (line 01-02). The **Data_Change_Event** (line 04) function sends a message when a change occurs in the office document. Data is stored in the user-defined type, ***MyType*** before sending a message (line 05). The *checkpointed channel type* is defined (elsewhere) to have the properties mentioned above: ordered, reliable etc. Any live object implementing the specified behavior can be connected to this endpoint.

Not shown is the code that actually initializes an object upon startup. This is done by requesting that another replica create a checkpoint; the joining process loads state from the checkpoint information to "catch up". If this is the first replica of the object to be launched, it loads its initial state from the on-disk copy of the document, in which it was embedded. Down the road, we plan to extend persistence support to deal with other aspects of the live object lifecycle.

2.5 Office Data Types

We've emphasized that live objects are strongly-typed. This was visible in our channel example, but type checking is actually used throughout the live objects platform [6], 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 (line 01). 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 [6].

```
01 [ValueClass("1'1","IMyType")]
02 interface IMyType : ISerializable
03 {    string _data {       get;       set;       }          }
```

In the example above, we define a new event type IMyType that will carry string values (line 03). The type is annotated with identifier (line 01) that allows it to be referenced from within the XML descriptions of the sort we saw in Section 2.4.

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 (Figure 2). 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.

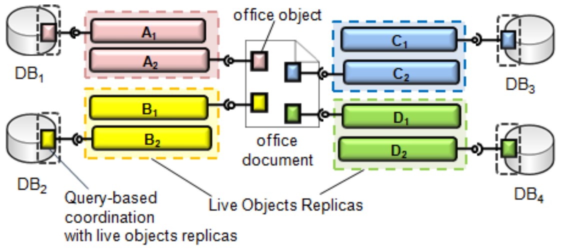


Fig. 2. Live objects can be used to integrate multiple databases, or to connect databases with other kind of office objects such as spreadsheets or documents.

We've 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 5.1.3, 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.

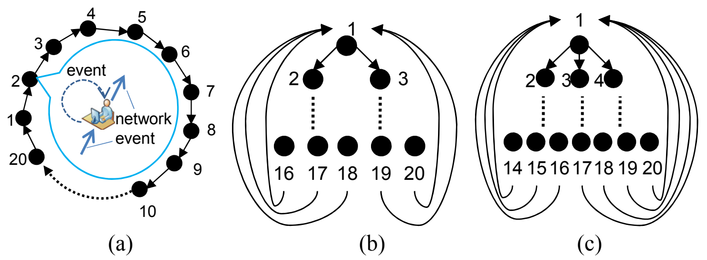


Fig. 3. Various information topologies for office systems such as (a) ring structure, (b) binary tree structure, and (c) ternary tree structure. The "loop back" links are used in our timing experiments but wouldn't be present in "real" configurations (see section 5.1.3).

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).

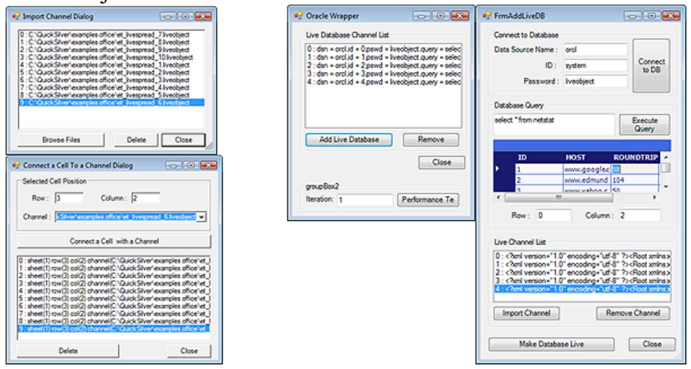
The DIL leverages Microsoft's Object Linking and Embedding (OLE) technology. OLE enables elements of a compound document to communicate with one another via COM interfaces, and also standardized data representations within the Microsoft office product suite [20]. Our DIL leverages these interfaces but 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, etc.

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. Object-based coordination for DIL and query-based coordination for DBIL are covered in Section 4.4.

4.1 Graphical User Interfaces (GUIs)

At design time, developers (who will often lack programming skills) work primarily through the live objects GUI interface.



```
0 : C:\QuickSilver\examples office\et_livespread_7.liveobject
1 : C:\QuickSilver\examples office\et_livespread_8.liveobject
2 : C:\QuickSilver\examples office\et_livespread_9.liveobject
3 : C:\QuickSilver\examples office\et_livespread_10.liveobject
4 : C:\QuickSilver\examples office\et_livespread_1.liveobject
5 : C:\QuickSilver\examples office\et_livespread_2.liveobject
6 : C:\QuickSilver\examples office\et_livespread_3.liveobject
7 : C:\QuickSilver\examples office\et_livespread_4.liveobject
8 : C:\QuickSilver\examples office\et_livespread_5.liveobject
9 : C:\QuickSilver\examples office\et_livespread_6.liveobject
```

Row : 3 Column : 2
Channel : :Silver\examples office\et_livespread_6.liveobject

```
0 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
1 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
2 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
3 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
4 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
5 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
6 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
7 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
8 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et_l
9 : sheet(1) row(3) col(2) channel(C:\QuickSilver\examples office\et
```

```
0 : dsn = orcl.id = 0.pswd = liveobject.query = selec
1 : dsn = orcl.id = 1.pswd = liveobject.query = selec
2 : dsn = orcl.id = 2.pswd = liveobject.query = selec
3 : dsn = orcl.id = 3.pswd = liveobject.query = selec
4 : dsn = orcl.id = 4.pswd = liveobject.query = selec
```

Iteration: 1

Data Source Name : orcl
ID : system
Password : liveobject

select * from netstat

ID	HOST	ROUNDTRIP
1	www.google.c	58
2	www.edmund	104
3	www.yahoo.c	50

Row : 0 Column : 2

```
0 : <?xml version="1.0" encoding="utf-8" ?><Root xmlns:>
1 : <?xml version="1.0" encoding="utf-8" ?><Root xmlns:>
2 : <?xml version="1.0" encoding="utf-8" ?><Root xmlns:>
3 : <?xml version="1.0" encoding="utf-8" ?><Root xmlns:>
4 : <?xml version="1.0" encoding="utf-8" ?><Root xmlns:>
```

Fig. 4. (a) GUIs for DIL. (b) GUIs for DBIL.

The DIL is accessed through the two dialog windows shown in Figure 4a. 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 [3].

4.2 Startup Costs and Serialization

Startup. 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 live objects 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 etc. The macro code segment is stored within a VB component called *MyModule*, which declares the corresponding event handlers. Keep in mind that different events may occur in document replicas on different nodes within the network. The same module is used, but different methods are invoked by the DIL.

For the first of our two initialization cases, when live objects request a document checkpoint from DIL, the latter issues a call to the **Workbook_BeforeSave** method (line 05). The code serializes the contents of all live spreadsheet cells, creating strings that the DIL compiles into a checkpoint, and that are delivered by live objects to the joining application (line 06). The ability to replicate data isn't limited to spreadsheet cells. Any OLE-conformant objects embedded in compound OLE documents can be replicated using this scheme.

On the node with the newly opened document replica, the serialized strings are received in the **Workbook_Open** function. This loads the checkpoint in line 02, and then deserializes the contents, thus initializing any OLE objects with the state obtained from the document replica that generated the checkpoint.

```
01 Private Sub Workbook_Open()
02    MyModule.Load Workbooks(ActiveWorkbook.Name)
03    MyModule.Deserialize
04 End Sub
05 Sub Workbook_BeforeSave()
06    MyModule.Serialize
07 End Sub
```

As mentioned earlier, DBIL is used only to capture events from a database, not for database replication. DBIL uses a similar serialization/deserialization 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 (e.g. 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 (COM) 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.

During initialization, the procedure **SetCurrentSheet** is called by the DIL, with a reference to the current worksheet of the spreadsheet (line 01, below). We bind a handler to the **Data_Change_Event** function (line 04). Later, the worksheet changes, an update event is raised. We intercept it, as shown below. Now, each time data is updated in cells managed by DIL, our delegate function (*EventDel_CellsChange)* is invoked. The function serializes the worksheet contents, and multicasts the modified values. Upon delivery on spreadsheet replicas, the received values are deserialized by OLE, and applied, in parallel, to the all copies.

```
01 void SetCurrentSheet(Worksheet sheet) {
02    this._sheet = sheet;
03    EventDel_CellsChange =new DocEvents_ChangeEventHandler(
04      Data_Change_Event);
05    this._sheet.Change += EventDel_CellsChange;          }
```

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 aplication 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 [16] and MS SQL Server 2005 [15]). When the DBIL is launched, it registers itself as a database client. In the code fragment below, used with Oracle, this occurs in line 05. To detect changes in the monitored data, we associate a function, **Change_Event** (line 07), with each relation we wish to monitor; this is done using the *OracleDependency* interface (line 06) provided by ODP.NET.

Our code fragment monitors a relation called *mytable*. 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.

```
01 void DB_Connect()  {
02   String query = "select * from mytable";
03   OracleConnection con =new OracleConnection(constr);
04   OracleCommand cmd = new OracleCommand(query, con);
05   con.Open();
06   OracleDependency dep = new OracleDependency(cmd);
07   dep.OnChange+=newOnChangeEventHandler(Change_Event);   }
```

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. In particular, we'll measure CPU utilization and peak memory usage. 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 [5], 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 [5].

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 Measurements

5.1.1 Startup costs

We begin by evaluating the costs associated with opening office documents that use live objects. The first experiment evaluates initialization time and resource consumption as the number of live objects grows in a spreadsheet and an oracle wrapper.

DIL. From the discussion in Section 4, we can see that each live object has two kinds of associated costs: those due to our interception of events, initialization from a checkpoint, etc, and those due to the underlying OLE wrapper used locally within each replica. Our experiments seek to tease out the respective costs.

For each of the tests reported below, we selected one replica, launched the spreadsheet application, and then recorded processing time and resource consumption while varying the number of live objects embedded in the spreadsheet. We then broke down costs by category: that associated with live objects per-se, and that due to the underlying OLE technology. The results in Figure 6a-c show that OLE is the more expensive of the two: the costs of activating OLE objects are high and rise with the number of embedded objects. The maximum number of OLE objects embedded in the spreadsheet is 200; beyond this, we exhaust memory resources. In contrast, although the very first live objects we activate incur a sharp cost (the blue curve in Figure 6a), subsequent objects add very little additional overhead. In practice, the limiting factor is the cost of using OLE: OLE hits its limits well before we get to 200 objects: in Figure 6c we see that with more than about 60 OLE objects CPU utilization plunges because the application begins to page heavily. Overall, roughly 0.43 live objects can be initialized per second.

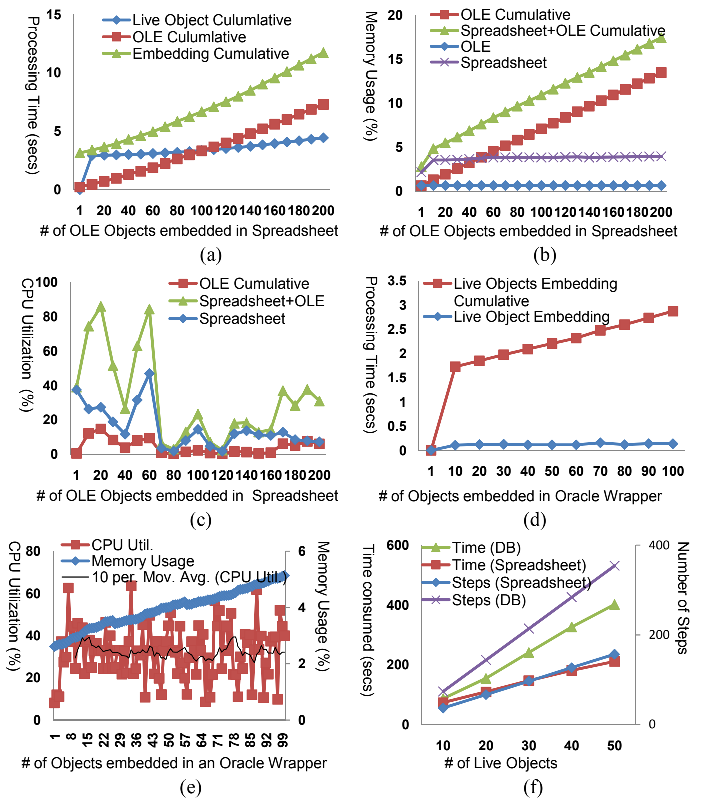


Fig. 6. (a) Processing Time while embedding live objects in a spreadsheet. (b) Memory Usage while embedding live objects in a spreadsheet. (c) CPU Utilization while embedding live objects in a spreadsheet. (d) Process Time while embedding database live objects as the number of live objects increase. (e) Memory Usage and CPU Utilization while embedding live objects in a database wrapper. (f) Time consumed and Number of Steps when the number of live objects grows.

DBIL. The startup process for our Oracle wrapper involves processing live objects descriptions and registering query statements with the database. Figure 6d shows processing time while live objects are embedded. 100 live objects are running in one machine in which each live object creates its own ODBC connection. Processing time increases linearly as the number of live objects grows, but still takes less than 3 seconds with 100 live objects. In Figure 6e, memory usage grows linearly but slowly with 100 live objects, particularly if we compare these figures with the OLE performance described earlier. The average CPU utilization is 32%, but exhibits a substantial fluctuation between 8% and 64%. We suspect that cache effects in the embedding process are associated with low utilization, and latency on ODBC connections with in high utilization, but it is difficult to pin down costs in a system such as ours, where Oracle and Microsoft Excel are being treated as "components". Overall, 0.13 live objects can be embedded per second during the startup process. Note that the resource consumption figures include costs associated with creation of TCP connections to implement WS-notification multicast channels as described in [3].

We conclude that processing time, memory usage, and the CPU utilization is linear in the number of live objects to be bootstrapped in both DIL and DBIL. Since these costs are reasonably low, the system should be stable and predictable.

5.1.2 Data Change Event Notification

As we saw in section 3, event notification occurs when data changes in a live spreadsheet or an Oracle database linked to a live object channel. We quantify performance with regard to three metrics: processing time, memory usage and CPU utilization, again showing the overall cost and the live object component of the cost in each case. Figure 7a-b show the average processing time for the spreadsheet and the Oracle wrapper, which increases linearly in the number of data change events. The peak at the 138th iteration in Figure 7a is due to the relatively high CPU utilization shown in Figure 7d. Figure 7d also shows that the average CPU utilization for the spreadsheet decreases slightly after the peak because of cache effects. With a steady rate of events, CPU loads actually drop after an initial startup period of higher costs. Memory use is stable at roughly 2%. Figure 7d presents the average CPU utilization, showing fluctuation between 30% and 80% average utilization at 60%. Memory usage increases nonlinearly; we attribute this to the CPU-intense nature of the ODBC interface code.

5.1.3 Information Flow (Event Pipeline) and Efficiency

Information Flow. 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 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 7a-7f. 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 is reasonably stable and offers reasonable performance, although the costs associated with the Oracle configuration are fairly high (stemming mostly from Oracle itself, as seen in Figure 7b and 7c). We attribute this to costs of the Oracle dynamic view materialization and to the overheads associated with the trigger mechanism.

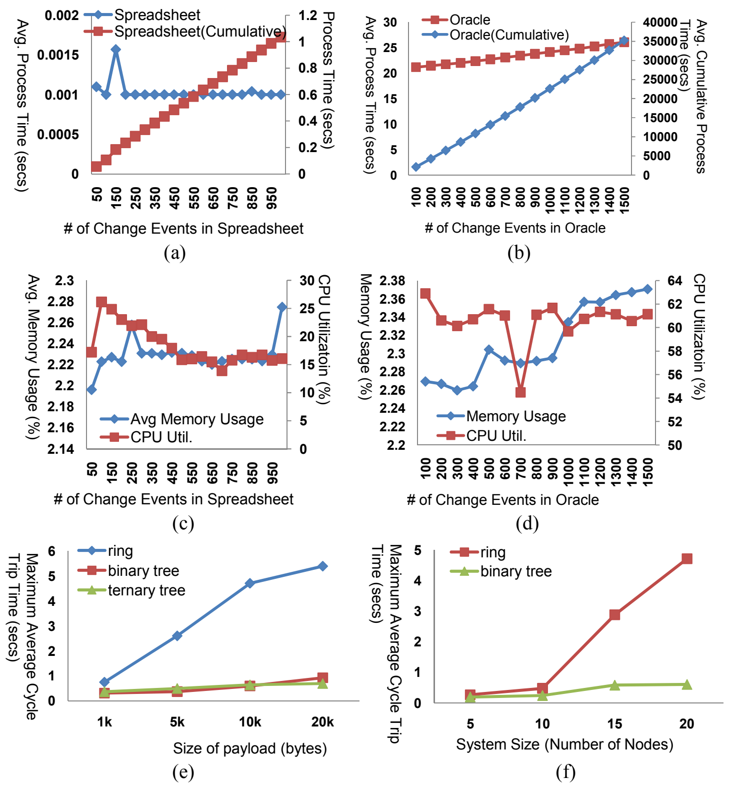


Fig. 7. (a) Processing Time while propagating data change events in a spreadsheet. (b) Processing Time with database events. (c) Memory Usage and CPU Utilization for the spreadsheet. (d) Memory Usage and CPU Utilization for the Oracle wrapper. (e) Maximum average cycle trip time when size of payload varies under various topologies. (f) Maximum average cycle trip.

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 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 as shown in Figure 6f. 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.

Table 1. User Interaction Steps for efficiency testing of the system

Document Integration Layer		**Database Integration Layer**	
Step	Instruction	Step	Instruction
1	Launch a spreadsheet.	1	Launch an Oracle wrapper.
2	Start an Import Channel dialog.	2	Press an Add Live Database button.
3	Import all live objects into the dialog by a drag/drop interface or a file browsing dialog.	3	Type information for database connection in text boxes.
4	Close the dialog.	4	Test database connection pressing a Connect DB button.
5	Start a Channel Coordination dialog.	5	Type a query statement in a text box.
6	Select a cell to be connected to live object in the spreadsheet.	6	Execute the query to see the result.
7	Choose a live object file already imported in a drop-down box.	7	Import all live objects into the dialog by a drag/drop interface or a file browsing dialog.
8	Press a Connection button.	8	Choose a live object file already imported in a list box.
9	Close the dialog.	9	Press a Connect button.
10	Save the spreadsheet.	10	Close the Oracle wrapper
11	Close the spreadsheet.	Repeated steps 3 through 9	
Repeated steps 6 through 8			

5.1.5 Concurrency and Scalability

Concurrency. When a single machine must support large numbers of live objects, or form large numbers of database bindings, scalability of our platform will determine the associated costs. Our experiments show that as many as 136 live objects can be used on a single machine, mostly because of the memory limitations discussed earlier (arising from the underlying OLE object). Communication costs are also an issue.

We sent 1kbytes payload for each trial, and measured delays, averaging over 30 trials. Figure 8a shows that the DIL is faster than the DBIL (we traced this to the cost of event notifications in ODBC which, as was seen earlier, is quite expensive). Figure 8b shows that the CPU utilization for the DIL in a spreadsheet is close to that of the DBIL in Oracle, although DBIL memory use grows with the number of live objects. This memory is associated with the connections to the database. In the case of the spreadsheet, the packet dissemination latency increases linearly up to the 70th live object embedded, but then soars as paging kicks in. A similar phenomenon is seen with Oracle as the system reaches 60th live objects. Memory growth for this test is graphed in Figure 8b.

The results show that substantial numbers of live objects can run concurrently in spreadsheets and databases. Memory sizes appear to be the primary limiting factor, and from these experiments, one can even predict the limits at design time.

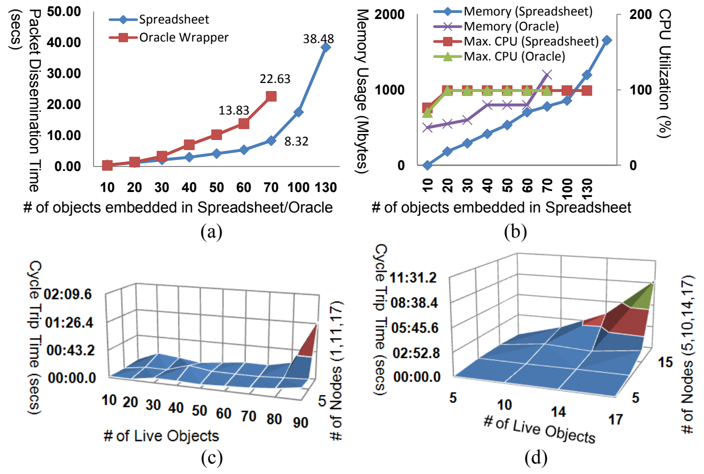


Fig. 8. (a) Packet Dissemination Time using the same channel. (b) Packet Dissemination Time using different channels. (c) CTT in one channel scenario. (d) CTT in various channels scenario.

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 8c 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 8d 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 discuss possible extensions to the WS-* 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 interconnected 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're also working on adapting our platform for use in WAN settings. Beyond these near term issues lie hard questions associated with supporting transactions and dealing with enterprise life-cycle management.

7 Conclusion

This paper described the design and implementation of a 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've 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.

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 they provided.

References

1. Dirk E. Mahling, Noel 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. Kenha Park, Jintae Kim, Sooyong Park. Goal based agent-oriented software modeling. Proceedings of the Seventh Asia-Pacific Software Engineering Conference, pp.320, December 05-08, 2000
3. Live Distributed Objects at Cornell. http://liveobjects.cs.cornell.edu
4. K. Ostrowski, K. Birman, and D. Dolev. Declarative Reliable Multi-Party Protocols. Cornell University Technical Report, http://hdl.handle.net/1813/8221
5. K. Ostrowski, K. Birman, D. Dolev. Quicksilver Scalable Multicast. In submission
6. K. Ostrowski, K. Birman, D. Dolev, J. Ahnn. Programming with Live Distributed Objects. In Proceedings of 22nd European Conference on Object-Oriented Programming (ECOOP'08)
7. Giampio Bracchi, Barbara Pernici. The design requirements of office systems. ACM Transactions on Information Systems (TOIS), vol. 2 no. 2, pp. 151-170, April 1984
8. Umeshwar Dayal , Meichun Hsu , Rivka 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
9. Brian Oki , Manfred Pfluegl , Alex Siegel , Dale Skeen. The Information Bus: an architecture for extensible distributed systems. Proceedings of the fourteenth ACM symposium on Operating systems principles, pp. 58-68, December 05-08, 1993, Asheville, North Carolina, United States
10. Giampio Bracchi , Barbara Pernici. The design requirements of office systems. ACM Transactions on Information Systems (TOIS), vol. 2 no. 2, pp. 151-170, April 1984
11. Jammes, F.; Smit, H.; Service-Oriented Paradigms in Industrial Automation. IEEE Trans. on Industrial Informatics, 2005, vol. 1, no. 1, pp. 62-70
12. 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
13. SAP. SAP Standard Benchmark https://www.sdn.sap.com
14. Traudt, Erin, Amy Konary (June 2005). 2005 Software as a Service Taxonomy and Research Guide 7. IDC. Retrieved on 2006-08-25
15. Microsoft. Microsoft SQL Database. http://www.microsoft.com/sql
16. Oracle. Oracle Database. http://www.oracle.com/database
17. Business Scenarios. http://www.microsoft.com/office/solutions/default.mspx
18. K. Birman. The process group approach to reliable distributed computing. Communications of the ACM (CACM) 16:12 (Dec. 1993)
19. F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. ACM Computng Surveys. 22, 4 (Dec. 1990), pp. 299-319
20. Microsoft Corporation (December 1993). OLE 2 Programmer's Reference: Creating Programmable Applications with OLE Automation. vol. 2, Programmer's Reference Library, Microsoft Press
21. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Proceedings of the Second IEEE/Create-Net/ICST International Confe-

rence on Communication System software and Middleware (COMSWARE). Bangalore, India, January 7-12, 2007
22. LUM, V., CHOY, D., AND SHU, N. OPAS: An office procedure automation system. IBM Syst. J. 21, 3 (1982), 327-350
23. GIBBS, S. Office information models and the representation of 'office objects'. In Proceedings ACM SIGOA Conference on Office Systems (Philadelphia, June 1982), ACM, New York, 21-26
24. ELLIS, C., AND BERNAL, M. OFFICETALK-D: An experimental office information system. In Proceedings ACM SIGOA Conference on Office Systems (Philadelphia, June 1982), ACM, New York, 131-140
25. BRACCHI, G., AND PERNICI, B. SOS: A conceptual model for office information systems. In Proceedings of ACM SIGMOD Database Week Conference (San Jose, Calif., May 1983), ACM, New York, 108-116
26. Sun Microsystems, Inc. Jini http://www.jini.org
27. Sun Microsystems, Inc. JXTA v2.0 Protocols Specification. http://www.jxta.org
28. Dave Chappell, Enterprise Service Bus, O'Reilly: June 2004
29. IBM. WebSphere Enterprise Service Bus http://www-306.ibm.com/software/integration/wsesb
30. Zloof, M. M. and deJong, S. P. (1977). The System for Business Automation (SBA): Programming Language, Communications of the ACM. vol. 20, no. 6, pp. 385–396
31. TSICHRITZIS, D. Form management. Commun. ACM 25, 7 (July 1982), 453-478
32. RICHTER, G. IML-inscribed nets for modeling text processing and database management systems. In Proceedings Very Large Data Bases Conference (Cannes, Sept. 1981), 363-375
33. ELLIS, C., AND NUTT, G. Office information systems and computer science. ACM Comput. Surv. 12, 1 (Mar. 1980), 27-60
34. BARBER, G. Supporting organizational problem solving with a work station. ACM Trans. Office Inf. Syst. 1, 1 (Jan. 1983), 45-67
35. KONSYNSKI, B., BRACKER, L., AND BRACKER, W. A model for specification of office communications. IEEE Trans. Commun. COM-30, 1 (Jan. 1982), 27-36
36. ZLOOF, M. QBE/OBE: A language for office and business automation. IEEE Computer 14, 5 (May 1981), 13-22
37. AIELLO, L., NARDI, D., AND PANTI, M. Structural office modeling: A first step toward the office expert system. To appear in Proceedings 2nd ACM Conference on Office Information Systems (Toronto, June 1984), ACM, New York
38. BAILEY, A., GERLACH, J., MCAFEE, P., AND WHINSTON, A. An OIS model for internal accounting control evaluation. ACM Trans. Office Inf. Syst. 1, 1 (Jan. 1983), 25-44.
39. SIRBU, M., SCHOICHET, S., KUNIN, J., AND HAMMER, M. OAM: An office analysis methodology. MIT, Office Automation Group Memo OAM-016, 1981
40. Giampio Bracchi, Barbara Pernici. The Design Requirements of Office Systems. ACM Trans. Inf. Syst. 2(2): 151-170 (1984)
41. Microsoft. Microsoft Office Groove. http://office.microsoft.com/en-us/groove
42. Google. Google Docs. http://documents.google.com
43. WebEx. Web Office. http://www.weboffice.com