RC-20869 (92341) 3JUN97
Computer Science/Mathematics 15 pages

Research Report

Embedded Workflow Manager System Design Overview

Guy Hochgesang
Richard Cardone
Houtan Aghili

IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Embedded Workflow Manager
System Design Overview

Guy Hochgesang
Richard Cardone
Houtan Aghili

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{hochges, richcar, aghili}@watson.ibm.com

ABSTRACT

The developing field of workflow management systems has lead to numerous implementations that relate workflow processes, users with defined roles, and activities with prescribed execution characteristics. Many systems provide a graphical user interface for runtime human interaction. This paper presents a design overview of a minimalist, distributed workflow engine that can be embedded in another application or application suite. This workflow subsystem is not visible to the end user at runtime nor does it require a directory of users or activities. It is designed for high performance, scalability and availability. An asynchronous, event-driven process model is used rather than the more typical activity process model. We also discuss our experiences deploying this system, benchmark tests and some areas for future work.

INTRODUCTION

A workflow manager can be thought of as a type of middleware that allows loosely coupled software components or applications to interact. Such a workflow manager can emulate the simple, synchronous connectivity of remote procedure calls or the asynchronous communication of message oriented queuing systems. Workflow middleware, however, is extremely flexible because it is *programmable* via its process definition language. This paper will describe in some detail the system design of an implemented workflow manager that fills this workflow-as-middleware niche.

The ***Embedded Workflow Manager (EWM)*** was designed to be a high performance, highly available, scalable, plug-in subsystem. Incorporating EWM into a larger distributed application does not change the end user's view of the system — EWM has no runtime interface for end users. Inversely, the workflow engine is unaware of end users because users, roles and activities are not required, assigned or otherwise known to the system. Any application that can call the EWM programming interface can be a client of EWM. EWM is a highly distributed implementation of a workflow engine, a runtime class library for client applications, and a set of utilities necessary to define workflow processes and to administer the runtime environment.

EWM provides an event-driven workflow process model which is a subset of the *ObjChart* process model described in [Gan] and [GanWu]. The semantics and notation of this model will be discussed shortly.

The current EWM implementation consists of a number of discrete server or daemon components and the EwmClient C++ class library for clients. The server components run on IBM's AIX operating system, version 3.2.3 or higher. The client class library is available on AIX and OS/2. EWM has been running as part of a clinical health care pilot system for approximately two years. In that time, EWM has performed well over 100 million transactions with its clients. Performance related results are discussed towards the end of this paper.

APPLICATION PROGRAMMING INTERFACE

While EWM provides a rich set of client application programming interfaces, there are only four primary methods for initiating and interacting with workflow processes: the **InitWorkflow**, **OpenCursorOnQueue**, **RecvServReq**, and **SendEvent** methods of the EwmClient class.

InitWorkflow initiates the execution of an instance of a workflow process. The process name and any arguments to the process are specified on the call. The arguments may be basic data types (integers, floating point numbers, or strings) or suitably defined C++ objects. Execution of a workflow instance usually results in the generation of one or more *service requests*. A service request is a message from a workflow instance to an EWM client (i.e., an application program that interacts with EWM). A service request can be used to notify the client application of some occurrence or to request the client application to perform some action. A service request consists of a name and zero or more arguments (as described above). A workflow instance can dynamically specify the target *service request queue* on which a service request should be placed based on its own application logic. Client applications retrieve service requests from one or more queues. All queues are managed by EWM.

OpenCursorOnQueue opens a cursor on a service request queue. Each EwmClient object may open cursors on multiple queues, but no more than one cursor on any given queue. A queue may have multiple cursors open on it at once. Cursors track the current position in the queue for an EwmClient object. Cursors can be opened in one of several modes including exclusive, independent and synchronized. EWM implicitly manages cursor positioning and clean up in case of failure.

RecvServReq reads service requests from one or more queues. Service requests are retrieved in FIFO order from each queue. If more than one cursor is opened by an EwmClient object, the order in which the queues are accessed by this object is indeterminate. EWM will return a copy of the requested number of service requests if they are available. Every time a service request is read, the cursor position is advanced to the next service request in the queue. When the client has received all service requests available through a cursor, the cursor points to the end of the queue. If all of a client object's cursors point to the end of their queues, the RecvServReq method blocks until a service request appears on one of the queues. RecvServReq does not cause service requests to be removed from their queues (reads are not destructive).

SendEvent sends an *event* from a client application program to an executing workflow instance. An event consists of a name and a list of arguments, just like a service request. SendEvent may also send a *done signal* for a service request; this causes the service request to be removed from the queue. SendEvent can also be used to atomically remove a service request from its queue and send an event to a workflow instance in one transaction.

THE EVENT-DRIVEN MODEL

EWM provides an event-driven model of workflow processes. In this model, a workflow process is described by a set of finite state machine transitions. Figure 1 shows the form of a single transition arc between two states of an EWM workflow process. A workflow process can consist of two or more states, and each state may have multiple transition arcs to any other state or states in the process.

$$S \xrightarrow[\textit{Actions}]{\textit{Event [Conditions]}} T$$

Figure 1. Workflow Process Transition Arc

The transition arc in Figure 1 may be read as, "If the workflow process is in state *S*, and *Event* occurs, and *[Conditions]* are true, then perform *Actions* and go into state *T* in one atomic step."

The *Event* part of the arc has the form,

$$EventName(a_1, a_2, \ldots)$$

where *EventName* is the name of the event sent to the workflow process and $(a_1, a_2, \ldots)$ are arguments of the event. Each argument may be a basic datatype (integer, float, string) or a C++ object derived from the (soon to be described) abstract EwmObject class.

The *[Conditions]* are of the form,

$$[c_1, c_2, \ldots]$$

where each c may be a comparison between two basic data items (e.g., "$i = 6$") or an invocation of a method of an object passed as an argument (e.g., "$a_2.method(a_1)$"). This method is expected to be a predicate returning *TRUE* or *FALSE*. The *[Conditions]* are true if all of $[c_1, c_2, \ldots]$ are true.

The *Actions* have the form,

$Q_1 \leftarrow SR_1\ (b_{11}, b_{12}, ...)$
$Q_2 \leftarrow SR_2\ (b_{21}, b_{22}, ...)$
...

where each Q is the name of an application queue, each SR is the name of a service request, and each b is an argument of the service request. Each action places the specified service request with its arguments on a service request queue.

WORKFLOW DEFINITION

Workflow process development consists of the following steps:

1. Modeling the business process.
2. Defining the business objects used during process execution.
3. Defining and simulating the process flow or logic.
4. Implementing the business objects in C++ as subclasses of EwmObject.
5. Testing the process definition and application defined C++ objects.
6. Installing the process definitions and application defined C++ objects.

Process modeling, definition, simulation and testing, are part of the EWM definition environment which is distinct from the EWM execution environment. A process definition tool is used to perform most of these tasks. The embedding application usually defines its own C++ classes that inherit from the abstract EwmObject class. These classes or class hierarchies can perform anything the application requires. Objects of these classes can be used as parameters to processes, service requests and events. Once process definitions and application objects are complete, they can be installed in the EWM execution environment.

An important feature of EWM is that it does not need to be recompiled or relinked to use application defined objects. EWM can dynamically load objects of unknown type and can indirectly invoke any of their methods. The abstract EwmObject class makes this possible by requiring all process objects to support a simple interface used by EWM servers during process execution.

Process Definition Tool

The current EWM implementation contains a prototype GUI interface that allows one to create the states, actions, conditions and arcs between states that comprise a workflow process definition. Service requests and events (with their parameters) can also be defined graphically. Methods on application defined objects can be referenced. The process definition tool is very simple at this time. The goal is to eventually develop an integrated modeling, definition and simulation environment to handle all the offline tasks associated with process definition. The environment would include an interface repository for process definitions, service requests, events, and application objects.[1]

Process Definition Installation

The initialization of EWM servers includes loading into memory the set of process definitions that may be invoked. Process definitions may also be added or removed from server memory while the server is running. Process definitions and the compiled modules containing application defined classes are stored in specific directories known to the EWM server. Installation of process definitions and application modules is essentially an exercise in UNIX file and directory management.

DISTRIBUTED WORKFLOW EXECUTION

A Hypothetical EWM Domain

EWM was designed with scalability in mind. Figure 2 illustrates a set of hypothetical EWM servers that cooperate to form an *EWM domain*. This stylized EWM domain consists of a global Queue Directory, EWM servers and EWM clients.

Each EWM server is assigned a *logical server id* (S_1, S_2, ... S_n) that uniquely names it within the domain. Each server manages its own set of executing workflow processes, service request queues and service requests. The servers cooperate by sharing these resources among themselves and all their clients. Each client is associated with its default *primary server*. In Figure 2, client C_1 uses server S_1 as its primary server. Clients can be configured to automatically select a primary server from a list. Though usually unnecessary, clients may explicitly change their primary server designation at any time during execution.

Client initiation of a workflow process causes the specified process to execute from beginning to end on the client's primary server. EWM workflow processes do not migrate from one server to another during

[1] The ObjChart modeling tool provides much of this support in a related environment. For more information, refer to http://www.software.ibm.com/clubopendoc/objchart.html.

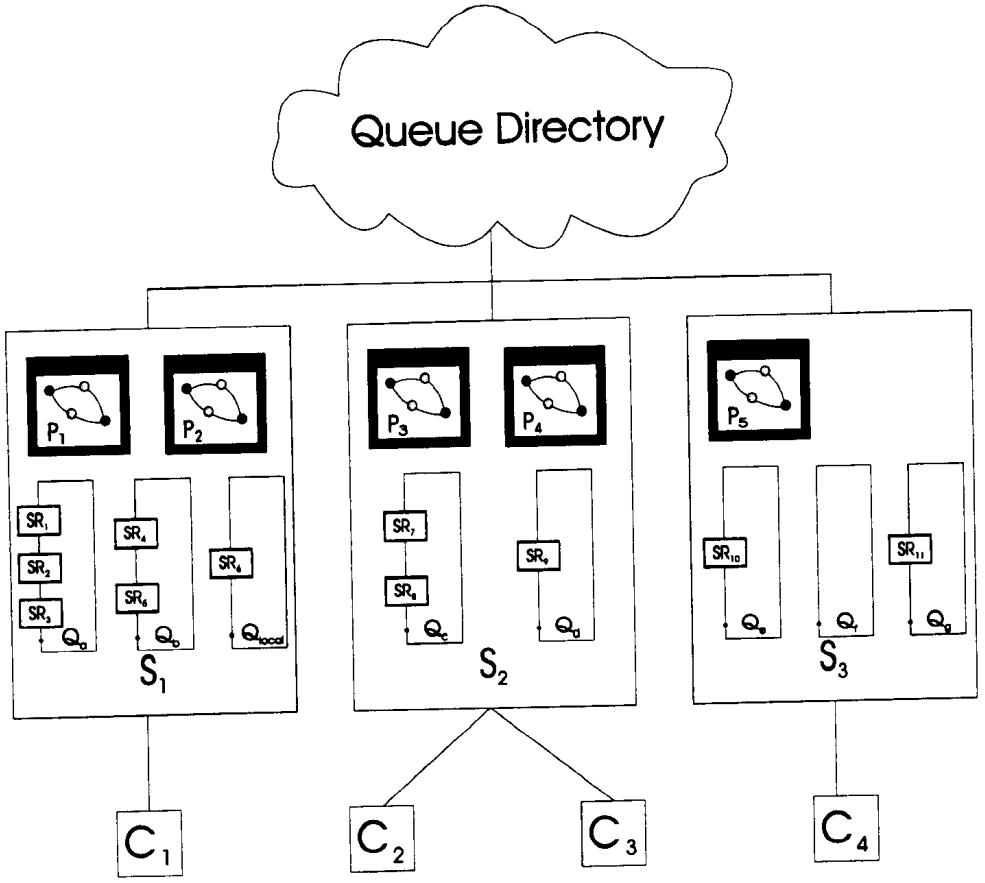

Figure 2. Hypothetical Domain

normal execution. However, all clients and servers in the domain may interact with any executing workflow process through the mechanism of *self-locating ids*. Each workflow process is assigned a unique *workflow id (wfid)* which allows it to be accessed from any client or server node within the domain. Assuming the configuration shown Figure 2 to be static, then only client C_1 could have initiated workflow process P_1, though all clients and servers may send events to P_1. Similarly, service requests are also named using self-locating ids known as *service request ids (srids)*. A service request may be accessed from any client or server in the domain via its srid no matter what queue it resides on.

Service request queues, on the other hand, are named by client applications and are located through a global directory facility. When an EwmClient object's **CreateQueue** method is invoked, a queue is created on the client's primary server and its existance is registered in the global Queue Directory. Queues may also be created by an administrator using a utility program.

Service request queues have scope — they can either be local to the server they reside on or global to all servers in the domain. Clients may access any global queue and all queues local to their primary server. In Figure 2, all queues are global except queue Q_{local}. Whereas client C_1 can explicitly access all queues shown in the example domain, client C_2 can explicitly address all queues ***except*** Q_{local} under the depicted configuration. Queue names are unique within their scope. This implies that global queue names are unique throughout the domain and that local queue names may be repeated on multiple servers. Local and global queues behave identically once accessed.

In addition to local and global service request queues, EWM supports *local and global alias names* for queues. Aliases can be created by client programs or by an administrator using a utility program. A queue may be known by any number of alias names. An alias can refer to any queue in the domain, but not to another alias. The same scoping rules that apply to queues apply to aliases.

EWM also supports *automatic queue* definitions. An automatic queue definition predefines, but does not create, a queue. These definitions are always global to all server complexes in the domain. If a predefined queue is referenced and not found within the current scope, it is automatically created.

The definitions of all automatic queues, local and global queues, and local and global aliases are kept in the global Queue Directory. The queues themselves reside on server nodes.

A Simple EWM Domain

The Embedded Workflow Manager is actually implemented using four types of servers, a daemon process that runs on client hosts, and the EWM client interface class library. Figure 3 depicts the minimal EWM installation consisting of one instance of each runtime server on a single host. A client daemon instance runs on each client host. This configuration can be thought of as the centralized, single host case of the fully distributed configuration that will be described shortly.

The **Global Domain Server** manages the master copy of the queue table and the logical server table in permanent store. The queue table contains the name, location, scope, access rights and other descriptive information for every queue in the domain. The logical server table maps unique logical server ids to host network nodes. This mapping allows the resolution of workflow process and service request locations via their self-locating ids. Together these two tables comprise the EWM *domain data*. Authorized clients may dynamically insert, delete or update the domain data at runtime. The Global Domain Server is implemented as single UNIX process.

The **Local Domain Server** caches a copy of the domain data on each server node in the EWM domain. A caching strategy is employed that minimizes network transactions and reduces the occurrence of spikes and overall load on the Global Domain Server. Local Domain Server caching satisfies critical performance and availability requirements in distributed (real life) configurations. The Local Domain Server does not maintain any permanent store and is implemented as single UNIX process. It communicates with the Global Domain Server using dedicated TCP/IP sessions.

The **Execution Server** implements the runtime workflow semantics of the Embedded Workflow Manager. It manages all service requests, service

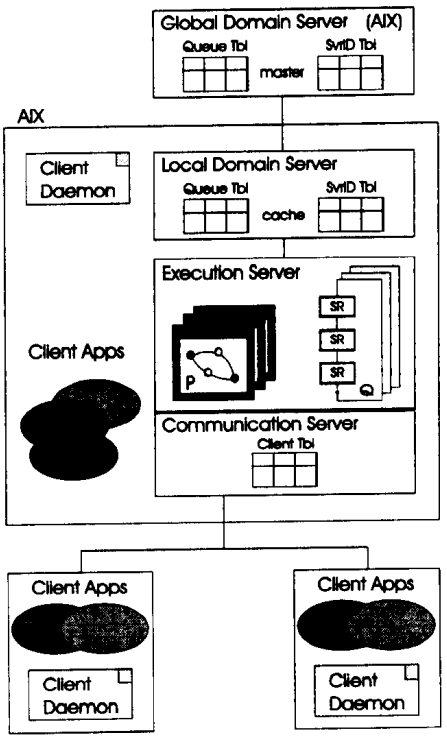

Figure 3. Simple Domain Configuration

request queues, events, executing workflow processes, workflow process definition files and application class module files. Workflow processes are executed as instantiations of process definitions. During process execution, application defined C++ objects will be dynamically loaded and invoked. Service requests generated by the process can be sent to any in-scope queue. Events generated externally by clients may change the state of the process.

The Execution Server maintains the state of the local workflow system in non-volatile memory. The server's permanent store is implemented using a journaling strategy similar to those used in database management systems. This store provides local transaction atomicity and recovery for all workflow process, service request and event state changes.

An Execution Server defines a logical server instance and is assigned a logical server id at configuration time that uniquely distinguishes it in the domain. The Execution Server communicates with its Local Domain Server over dedicated UNIX socket sessions. These connections are used to obtain up-to-date domain data and to identify the Execution Server within the domain. The Execution Server partitions its work across five UNIX processes which interact via shared memory. These processes are distinguished as follows:

Process Name	*Process Function*
Execution	Server initialization and workflow execution.
Communication	Incoming client request handler.
Notification	Asynchronous notification to clients of certain queue state changes.
Delivery	Immediate delivery of service requests and events to other Execution Servers.
Retry	Periodic retries to deliver service requests and events whose previous delivery attempt failed.

The **Communication Server** provides the means by which client applications interact with the EWM subsystem. It provides a class library interface which is dynamically linked into the client application. The EwmClient class provides an object-oriented view of the workflow subsystem to the client application. Each EwmClient object is viewed by the Communication server as a unique client. The Communication Server is responsible for interacting with and managing all EwmClient objects. A single client application may use any number of EwmClient objects concurrently. Each EwmClient object may choose its own quality of service from a variety of network session types. Both TCP and UNIX stream sockets are used.

The Communication Server has no permanent store and is always associated with a single Execution Server. The server is a multi-process UNIX program that uses shared memory for communication between sibling processes and with the Execution Server. The nine process types employed by the Communication Server are:

Process Name	*Process Function*
Parent	Initializes server and monitors execution of children processes.
Transient TCP Session[2]	Handle requests from multiple concurrent clients, where each transaction opens a new TCP socket connection that is closed when the transaction is complete.
Transient UNIX Session[2]	Handle requests from multiple concurrent clients, where each transaction opens a new UNIX socket connection that is closed when the transaction is complete.
Process Spawner[3]	Spawn dedicated processes to handle a single EWM client session.
Dedicated TCP Session[2]	Handle requests from a single client, using a permanent TCP socket connection (child of Process Spawner).
Dedicated UNIX Session[2]	Handle requests from a single client, using a permanent UNIX socket connection (child of Process Spawner).
Multiplexed Session[2]	Handle requests from multiple concurrent clients, using a permanent TCP and/or UNIX socket connection.
Notification[4]	Send interrupts to blocked client objects.
Zombie Client Detector	Detect and clean up abnormally terminated clients.

The Communication, Execution and Local Domain servers use either shared memory or UNIX stream sockets to transact with each other. This necessitates that all three servers run on the same host. Moreover, the Communication Server requires the Execution Server to run and the Execution Server can only run if the Local Domain Server is running. Given these operational constraints, it is convenient to refer to the three servers as a single entity, the *logical server complex* or, simply, the *server complex*. Only one server complex may run per host.

The **Client Daemon** allows the Communication Server to detect abnormally terminated clients and release their EWM resources. A single instance of the Client

---

2 Zero or more processes may be configured.

3 Zero or one process may be configured.

4 One or more processes may be configured.

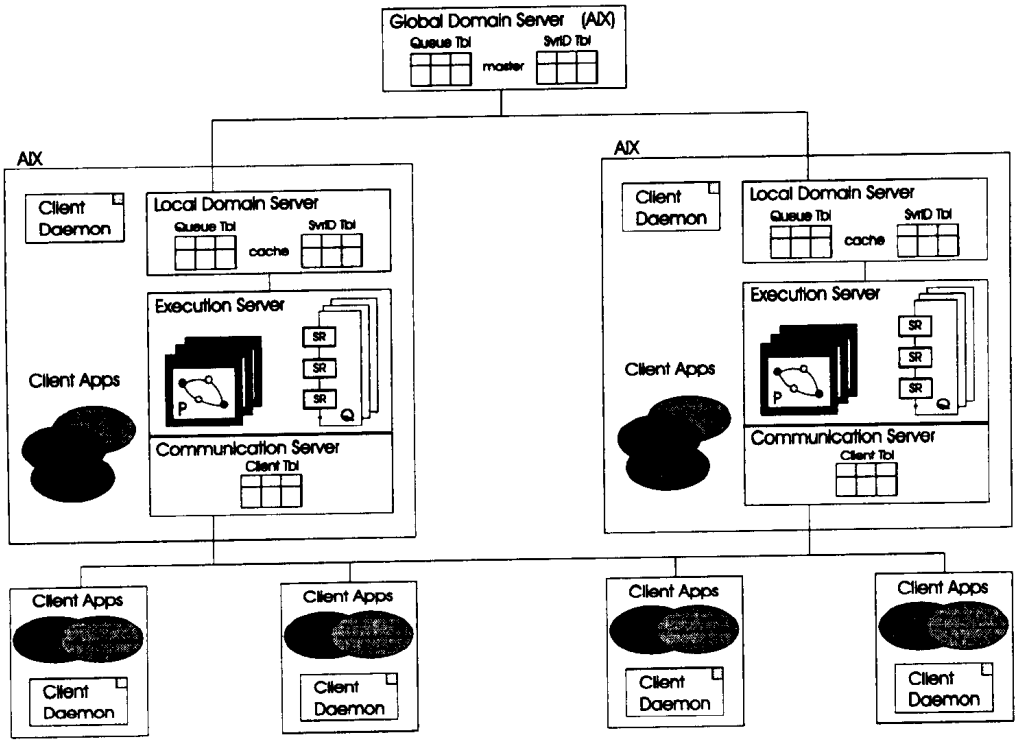

Figure 4. Distributed Domain Architecture with 2 Server Complexes

Daemon should run in the background on each client host. It is a single process program without permanent store and does not interact directly with client applications. The Client Daemon uses process management features of the operating system to dynamically determine the existence of specific client objects in response to Communication Server inquiries.

The Distributed EWM Domain

Figure 4 illustrates the currently implemented distributed EWM architecture. Components described in the simple configuration are now replicated and distributed across a TCP/IP network. Specifically, a distributed EWM domain consists of one Global Domain Server, multiple logical server complexes, as many client daemons as there are client hosts, and the clients themselves. Scalability is achieved by adding new server complexes (on new hosts) to distribute the network and processing load. Domains with eight server complexes have been tested and domains with fifty server complexes have been simulated. Figure 4 depicts a two server complex domain, the simplest and smallest distributed domain.

THE COMPLETE ARCHITECTURE

The Massively Distributed EWM Domain

Figure 5 illustrates the fully distributed EWM architecture which has not yet been implemented. In this domain, a single active Global Domain Server services an N-level tree of Local Domain Servers. Local Domain Servers in the top N-1 levels of this tree act as intermediate nodes for the servers below them in the tree. The global server's load is shared and distributed down the tree by these intermediate nodes which appear as *virtual* global servers to the nodes beneath them. In this way, the number of clients directly connected to the Global Domain Server can be limited even as the number of server complexes grows dramatically. The subtree of just one intermediate node is shown in Figure 5. A Local Domain Server may be part of a server complex and act as an intermediate node simultaneously.

The complete EWM architecture also specifies an optional hot standby configuration for the Global Domain Server. This is a active/backup configuration in which the backup server can immediately detect malfunctions at the active server and force a switchover. The active server can also detect a malfunctioning backup server and run without a backup.

As mentioned, the current implementation does not support the intermediate nodes nor the hot standby server of the complete architecture. The brief description of these facilities here is only meant to illustrate our approach to massive scale up. Design is at a preliminary stage.

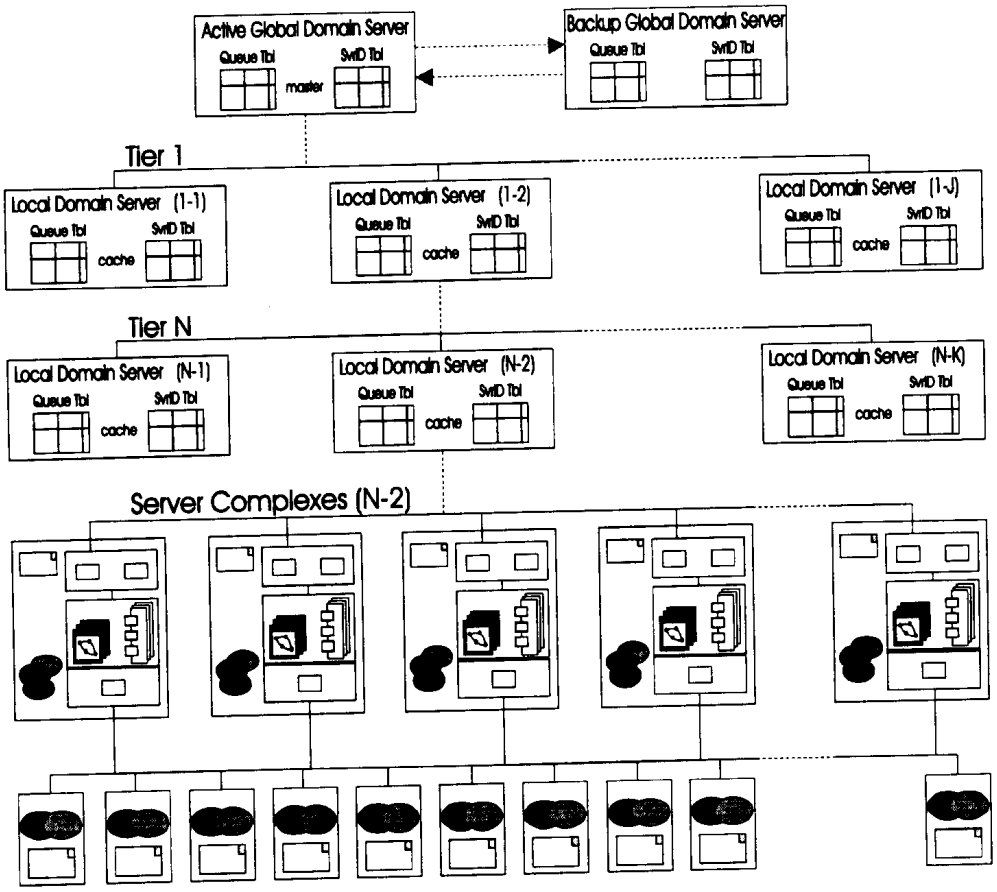

Figure 5. Massively Distributed Domain Architecture

DESIGN OBJECTIVES

Performance

An important design goal for EWM was to achieve a relatively high sustainable transaction rate for a large number of concurrent clients. The Execution and Communication Servers use shared memory and semaphores for interprocess communication to reduce the number of times data must be copied. Execution Server journaling provides full roll-forward recovery while maintaining high throughput. The Communication Server allows clients to choose between a number of persistent and transient session types based on their performance needs. The caching strategy used by the Global and Local Domain Servers is central to the distributed performance and will be discussed shortly under the topic of scalability.

The test results in Table 1 on page 9 describe the performance of a single EWM complex running on an IBM RS/6000 Model 591. The client test programs ran on an RS/6000 Model 590.[5]

In the first test ("Init Workflow"), a producer test program repeatedly initiates a simple workflow process, which queues a single service request and terminates.

In the second test ("Recv ServReq"), a consumer test program repeatedly receives service requests from a queue.

The third test ("Recv ServReq Send done") is the same as the second, except that after receiving each service request, a done signal is sent causing the service request to be deleted.

All results are in transactions per second (rounded to the nearest integer), where a transaction is defined as one iteration of the producer or consumer test program. In the first two tests, each transaction involves a single interaction with EWM. In the third

[5] Models 590/591 are deskside, departmental servers with SPECint95 rating 3.33/3.84 and SPECfp95 rating 10.4/12.4.

test, each transaction involves two interactions with EWM: a RecvServReq call and a SendEvent call.

In Table 1, the row labeled "T" shows the transaction rates using transient TCP sessions between the client test program and the EWM Communication Server. A new TCP session is initiated and terminated for each transaction. The row labeled "D" shows the transaction rates using dedicated (permanent) TCP sessions. The results in Table 1 show that dedicated sessions perform substantially better than transient sessions. Dedicated sessions should be used by clients with high transaction rates. Transient sessions may be used by low volume clients to reduce the demand on TCP/IP resources.[6]

Table 1. Sustained Transactions Per Second of a Single Server Complex

Client Session Type	Init Workflow	Recv ServReq	Recv ServReq Send done
T	23	24	12
D	57	112	56

Scalability

Scalability of distributed applications like EWM ultimately depends on client and server processing speeds, network speed and bandwidth, the number of server hosts, and the ability to effectively utilize multiple server hosts. Though faster servers and networks will improve EWM's total throughput, there is a practical limit to this approach. The ability to massively scale up EWM comes from incorporating new server hosts (i.e., server complexes) into a domain. Adding more server complexes will only increase total domain throughput and concurrency if the following hold true:

1. The overhead for using many servers is small.
2. The overhead for allowing many clients is small.
3. The network load can be distributed.
4. The processing load can be distributed.

DOMAINS WITH MANY SERVERS

EWM servers interact with each other over the network when redirecting events and service requests to remote workflow processes and queues, respectively. The Execution Server performs this remote redirection asynchronous to its other processing. As long as the processor and network are not saturated, the effect of this redirection is marginal.[7] Efficient domain configuration minimizes the number of redirections that take place by clustering clients that usually interact on the same server complex. Redirection may take place in any domain with more than one server complex. Adding more server complexes to a domain may either increase, decrease or not effect the number of redirections depending on their configuration.

On the other hand, adding new server complexes to a domain does increase the size of the logical server table managed by the Global Domain Server and cached by Local Domain Servers. Ultimately, this server table information is transmitted to and cached on each client application. It is used by EWM client interface code to route events and service requests directly from the client node to the target server using self-locating ids. The architected maximum number of server complexes allowed in a domain is 65,534.

The test results in Table 2 on page 10 indicate that increasing the size of the logical server table has little effect on client application performance. Clients in a domain with 63,000 assigned server ids took .68 seconds longer to load the server table than in a single server domain. This near-worst case performance penalty occurs only once during each client application's initialization with EWM. The results also show that searching and inserting into a large server table also scales well with the number of logical servers. All times are in seconds.

---

[6] All test results reported in this paper were obtained using RS/6000 uniprocessor clients and servers each with one token ring network adapter running at 16Mb. The small variations in performance should not be considered significant because neither the network nor the machines could be completely isolated for testing. Unless otherwise noted, all tests were run using production configurations (full authentication, journaling, etc.).

[7] Results from load distribution tests reported in Table 4 on page 11 support this assertion.

Table 2. Effect of Increasing Number of Server Complexes

Number of Server Complexes	Server Table Load Time Increase	Server Insert Time
1	-	-
100	.02	.027
1,000	.03	.028
10,000	.16	.034
20,000	.20	.035
40,000	.46	.037
63,000	.68	.041

DOMAINS WITH MANY CLIENTS

Other than the additional runtime load caused by increasing the number of concurrent clients, the domain's queue table usually increases in size as the number of clients increases. Embedding applications usually associate one queue with each of their clients or end users. We can expect caching between the Global and Local Domain servers to be slower as the amount of queue table data increases. Queue name resolution (table searches) should also be more expensive as the queue table grows in size.

The results in Table 3 reflect the effects of increasing the number of defined queues. Figures in each column represents the average of multiple runs of 1,000 transactions. Local Domain Server caching contributed to a 5% increase in the time needed to perform the first run at the 100,000 entry size; caching effects were not noticeable for smaller table sizes. After the first run at each table size no further caching activity occurred. The results also indicate that queue table searches and insert/delete operations scale relatively well. "NR/s" means the number of name resolutions per second; "Queue Create-Delete/s" means the number of queue creation/deletions (both operations) per second.

Table 3. Queue Table Performance

Queue Table Size	NR/s (misses)	NR/s (hits)	Queue Create-Delete/s
0	1148	-	12.3
1,000	1186	926	12.2
10,000	1176	833	11.9
100,000	1197	438	4.1

DISTRIBUTING NETWORK LOAD

EWM clients and servers can be distributed across a wide area network as topology dictates. Clients and servers can be configured to interact most efficiently with their usual partners by optimizing their network proximity. Further configuration includes choosing appropriate quality of service options for each Communication Server client. All clients and servers in the EWM domain must be able to communicate with each other. EWM uses TCP/IP and does not broadcast data. EWM works best when the underlying wide area network can provide LAN-like performance. If this is the case, EWM should scale with the network as routers and bridges are added.

DISTRIBUTING PROCESSING LOAD - PART 1

The ability of EWM to scale up rests mostly in its ability to efficiently distribute load over multiple hosts. EWM can be scaled upward by adding new host platforms to a domain. Each host would run a new server complex to distribute the processing (and network) load. For scalability, the addition of new server complexes should increase the total throughput of the EWM domain in near-linear fashion. Also, the addition of new server complexes should not cause the Global Domain Server to become a bottleneck.

Table 4 on page 11 illustrates the effect on overall domain throughput as hosts are added. The total load on the domain was kept constant and the network had sufficient bandwidth. For practical reasons, the largest domain size we could test consisted of eight server complexes.

Table 4. Domain Scale-Up			
Server Com-plexes	**Client Trans/s**	**Meas-ured Scale-Up**	**Ideal Scale-Up**
1	84	-	-
2	96	1.0	1.0
4	171	1.8	2.0
6	273	2.8	3.0
8	351	3.7	4.0

The above results were obtained by running seven client test programs on each of four client hosts concurrently. These twenty-eight client programs provided a constant load as server complexes were added to the domain. This was an extreme load for a single LAN segment. Each program instance opened two multiplexed sessions with the servers. One dedicated session was used to submit workflow process initiation requests; the other was used to receive resulting service requests and to send done signals. This init-recv-send sequence was repeated without delay for a period of time so that reproducible transaction rates could be measured. The transaction rates reported in Table 4 indicate the number of client-initiated EWM transactions per second, occurring over the whole domain.

For each run, fifty-six multiplexed client sessions were distributed between the servers evenly and in a manner that minimized the affinity between any client/server host pair. Except in the single server case, each invocation of a workflow process resulted in the generation of a service request on a remote queue. This is a worst case scenario in which server-to-server transactions were maximized. Even though only three EWM transactions were issued on each iteration of a test program's init-recv-send loop, four network transactions resulted in multi-server domains. Since single server domains cannot experience this implicit server-to-server traffic, and since our tests maximized this traffic, we did not consider the single server domain when calculating scale-up factors. The two server domain was used as the basis from which scale-up factors were calculated.

Table 4 shows the scale-up to be nearly linear. Each additional pair of server complexes increased the domain's throughput by approximately the capacity of two hosts.[8] As a rule of thumb, using transient rather than dedicated client sessions can cause total throughput to decrease by thirty to eighty percent. The actual decrease depends on the speed of the authentication subsystem (which must authenticate each new session) and the transaction mix. Running the same tests reported in Table 4 with unauthenticated transient sessions typically resulted in a thirty percent decrease in total domain throughput.

DISTRIBUTING PROCESSING LOAD - PART 2

To address the issue of the Global Domain Server becoming a bottleneck, tests were run to simulate a domain in which the single global server managed fifty directly connected, concurrent clients (i.e., server complexes). The results in Table 5 have been normalized to account for the effects of running multiple Local Domain Servers on a single host which is not normally possible.[8]

Table 5. Global Domain Server Fan-Out				
Server Com-plexes	**Total NR**	**NR/s**	**Meas-ured Scale-Up**	**Ideal Scale-Up**
1	10,000	781	-	-
50	500,000	31847	41	50

Changes to the queue table were performed once every second in the fifty host simulation. This was done to force cache refreshes on each Local Domain Server. In stable production systems, queue table changes might occur once every fifteen minutes.

The number of queue name resolutions per second scaled well when the number of server complexes increased to fifty. Linear scale-up would have required a fifty fold increase in throughput — we experienced a forty-one fold increase (82% of ideal) using conservative measurements. This simulation indicates that a Global Domain Server can directly manage at least fifty Local Domain Server clients without becoming a bottleneck even under extreme loads.

[8] See notes near the end of this paper for details on the normalization of servers.

In systems where the ultimate source of all work initiation is human users, each server complex can be configured to handle hundreds of concurrent users. For installations with users numbering in the tens of thousands, the intermediate node Local Domain Server support proposed in the complete EWM architecture would probably be necessary. We estimate that a fan-out factor of forty-one from the global server and each intermediate node would allow for reasonable performance at all nodes. The architected maximum number of 65,534 logical servers in a domain would be achieved with two tiers of intermediate nodes. The current implementation does not support intermediate nodes, so empirical data is not available to verify this design.

Without further testing and simulation, we cannot prove that EWM scales up in a linear fashion in domains with a large number of server complexes. We have shown that in domains with two, four, six, and eight server complexes, the increase in throughput when servers are added is nearly linear. We have also shown that the Global Domain Server does not become a bottleneck when fifty server complexes are simulated and frequent cache refreshes are forced. Given the system design and the empirical data, we believe EWM can achieve near linear scale-up as domains grow in size. Scalability should only become non-linear when external constraints such as network bandwidth come into play.

Availability

The complete EWM architecture specifies a hot standby Global Domain Server with its own copy of the domain data. This would eliminate the the Global Domain Server as a single point of failure in a domain. The hot standby capability is not currently implemented. The effects of losing the Global Domain Server can be mitigated by employing a recovery strategy and by supporting *limited function mode*. Both of these techniques have been implemented in the current version of EWM.

The termination of a Global Domain Server can be readily and immediately detected by its Local Domain Server clients through facilities provided by TCP/IP. The global server can be restarted on the same or alternate host machine as long as a copy of the domain data is available. For this reason, the global server's permanent store should be mirrored (using external facilities) on multiple hard drives preferably on multiple nodes. A properly configured domain allows all Global Domain Server clients to automatically locate the restarted server.

Each server complex will quickly transition into limited function mode whenever the global server becomes inaccessible. While in limited function mode, the complex can support all client requests that do not modify domain data (i.e., the queue configuration or logical server table). The Local Domain Server allows read-only access to its domain data cache while periodically attempting to reconnect to the global server. When the global server reappears on any configured node, each server complex will automatically reattach, refresh its cache, and resume full function mode execution. The behavior of the Local Domain Server's cache can be adjusted for a balance between performance and consistency.

The failure of a server complex most directly effects clients using that complex as their primary server. Some of these clients will have to reinitialize with the server when it becomes available again, others will be able to pick up where they left off. Transient errors will be experienced by all clients attempting to interact with the server complex while it is down. The persistent data maintained by the Execution Server must be available after a failure for recovery (data mirroring is recommended here also). A server complex may be recovered on any host connected to the network which does not already have a server complex executing. The ability of a domain to automatically readjust its server configuration allows clients to transparently locate the recovered logical server complex wherever it starts up.

Reliability

The Execution and Global Domain servers journal all significant state changes as atomic transactions. Changes either take effect completely or do not take effect at all. Recovery involves starting with an archive file and rolling forward the journals which, as mentioned above, should be mirrored. Server complexes can always be recovered independently and in parallel as long as the transactions being rolled forward do not modify domain data. If the domain data is updated by recovering transactions, roll forward events across multiple servers may need to occur in a specific order. Currently, this synchronization is a manual process.

On the client side, the Communication Server can detect abnormal client program terminations using the Client Daemon. EWM resources reserved, possibly exclusively, by these defunct clients are freed for use by other clients. Network failures that occur after a transaction has been committed, but before the client

receives the results, may leave a client unable to determine if its last request took effect or not. Incorporating more robust transaction processing capabilities is under consideration to deal with these rare events.

Serviceability

In addition to the recovery mechanisms already mentioned, EWM program failures can be isolated through a tracing and error logging facility. This is available on all client and server components. Server components also have a well-defined set of formatted operator messages that are written to file when important events occur. These files can be monitored by system management software to perform any needed integration into a larger distributed application environment.

Configurability

All EWM servers are initialized using configuration files when invoked. Some runtime characteristics may be changed while the servers are executing. New or modified workflow process definitions can be added to executing server complexes without effecting processes already executing. Approximately fifty utilities are provided to manage the execution environment of a domain.

Security

EWM uses the Generic Security Service (GSS) API to enable authentication via trusted third party authentication systems. Currently, IBM's Network Security Program is being used. Future migration to DCE's Kerboros implementation is being planned. All transactions between clients and servers or between servers are authenticated except in the following cases:

- Shared memory operations between the Execution and Communication server processes.
- New service request notifications sent by the Communication Server to client objects.
- Client verification requests sent by the Communication Server to the Client Daemon.

Servers hosts are assumed to be secure. The usual UNIX interprocess communication security is used to restrict access to server shared memory segments. The other transactions listed do not include any user data and would, at worst, cause operational anomalies (but no data loss) if intercepted. Encryption is supported and can be used on all authenticated network transactions.

The EWM servers restrict client actions by classifying all clients as either regular or administrative. Only administrative users are authorized to modify the server runtime environment or the domain data definitions of other users.

CONCLUSIONS AND FUTURE DIRECTION

As mentioned in the introduction, the incorporation of EWM into a large, distributed medical application demonstrates that the workflow manager is in fact embeddable. The notion of a workflow management system that does not determine an application's user interface has proved to be very pliable. It can be incorporated as middleware into existing applications in a manner transparent to the end user. Also, EWM does not require extensive registration, knowledge or administration of users or activities. EWM can integrate into any existing user directory or authorization scheme. EWM encourages a modular approach to a large, distributed application by limiting its own scope.

The current implementation has also proven to be scalable and able to sustain relatively high transaction rates. As a matter of fact, the clinical information system using EWM found it fast enough to be used in the bulk loading of historical medical data. Millions of records were streamed through EWM to populate the new system's databases. Though not intended or optimized for such tasks, using EWM saved the cost of developing numerous extraction and conversion utilities.

We believe that the event-driven model of workflow processes has several strengths. The paradigm of concurrently executing workflow instances and client application programs communicating via asynchronous events is natural for modeling real world concurrency. The event-driven model is also convenient for expressing exceptional conditions; an exceptional condition can often be handled by adding a new arc with an event representing the exception. A discussion of the advantages of the event-driven model in representing medical processes is presented in [GanWu].

Conceptually, the process definition language can be enhanced to extend the power of workflow managers in new directions. For example, a transaction processing model with user controlled commit/rollback and compensational, flexible or pivotal transactions could be supported [Alo] [Moh95]. Language enhancements could also include the ability for workflow processes to communicate with each other and execute other workflow processes (both synchronously and asynchronously). Additionally, work could be done in the area of timing constraints, deadlines, time-out periods, time-of-day events, etc..

Some runtime exceptions can best be resolved by dynamic human interaction. For example, service requests directed to non-existent queues end up in a *dead-letter* queue. The ability to interactively inspect, modify, and then re-inject these service requests into the system is desirable. The same capabilities are needed when a workflow process detects a runtime error or inconsistency. In both cases, it is necessary to interactively modify the state of a running process. Since processes and service requests can themselves contain any object defined by the application, these tools need to be very general and dynamic in how they treat the data. The challenge is to relate service requests to their sources (which may no longer exist) and provide enough semantic information for users to make an informed judgement. Some of these capabilities already exist in EWM, others require enhancements.

A number of architectural changes are also under consideration. Expanding the use of store and forward protocols on server complexes can provide more fault tolerance and better support for mobile clients. The incorporation of commercial queuing, database management, transaction processing, distributed object requestor/broker or global directory systems may allow us to concentrate on the workflow aspects of the system while building on a very robust foundation. Here, trade-offs with performance must be carefully considered.

Finally, the issue of portability and the tight coupling with any language (C++ in our case) can also be revisited. The current implementation allows client applications the full flexibility and power of an object oriented programming language, but it also restricts them to that language. Along the same lines, issues of interoperation with other workflow management systems and compliance with the Workflow Management Coalition standards are also areas for future consideration [WFMC95] [WFMC96].

NOTES

Table 4 Normalization

Unfortunately, the server host machines 7 and 8 were not as powerful as hosts 1-6 in Table 4 on page 11. Thus, the transaction rate for the 8-server complex case had to be normalized to account for the fact that we were not increasing the computing power by two more equivalent host machines. In order to perform the normalization, we used a single client program running the init-recv-send loop against each host machine running a single server complex. This provided us with a benchmark transaction rate for each host machine. Host machines 1-6 had almost identical transaction rates and were assigned normalization factors of 1.0. Host machines 7 and 8 were assigned normalization factors of 0.55 and 0.38, respectively, which were the ratios of their transaction rates to those of the more powerful hosts. For 8 server complexes, the actual transaction rates measured were multiplied by 1.15 to obtain the transaction rates shown in Table 4 on page 11. This is the ratio of the sum of the normalization factors for 8 equally powerful servers to the sum of the normalization factors of the 8 actual servers (1.15 = 8.0 + 6.93).

Table 5 Normalization

Architecturally, only one Local Domain Server may run per host. Fifty host machines were not available for use, so a simulation using five machines was run to test the scalability of the Global Domain Server. Our results were correlated using the following rationale:

1. We recorded the time for a single Global Domain Server client to perform 10,000 name resolutions. Each resolution was a "hit" on a queue table previously filled with 10,000 queue definitions. A throughput of 781 NR/s was achieved in 12.8 seconds using our fastest processor.
2. We then ran 10 clients on the same machine and achieved a throughput of 877 NR/s in an average running time of 114 seconds per client program. The 10 clients took on average 8.9 times longer to run than a single client took. If 10 clients took 10 times as long to run as a single client, the clients could be said to run strictly serially. This near-serial behavior was also observed when two to ten clients were run.
3. We assert that running 10 clients on a host is essentially equivalent to running the 10 clients on 10 different machines, where each machine is approximately 9 times slower than our actual machine. From the perspective of the Global Domain Server, each client is requesting service about 9 times less often than in the single client case. This factor of 9 became our normalization factor.
4. The 50 client case was run by executing 10 client programs on each of 5 hosts. An average throughput of 70.8 NR/s in 141.2 seconds resulted. Applying the normalization factor of 9 leads to an adjusted throughput of 637 NR/s in 15.7 seconds for each client program.
5. We chose not to perform any normalization to adjust for the different processor speeds of the five client machines. The machines are nearly

homogenous and our baseline single client test ran on the fastest machine. This results in the most conservative, least favorable, throughput figures.

6. By forcing each Local Domain Server to refresh its cache once a second, we created conditions where the global server might become a bottleneck. In the baseline single server case, 12 cache refreshes occurred. In the 50 server case, over 7,000 cache refreshes were forced which resulted in a refresh/transaction ratio over 11 times higher than for the baseline case ((141 * 50)/500,000 > 11(12/10000)). Again, this depresses the throughput and results in reporting a more conservative scale-up factor.

REFERENCES

[Alo] G. Alonso, M. Kamath, D. Agrawal, R. Gunthor, A. El Abbadi, C. Mohan. 'Advanced Transaction Models in Workflow Contexts', *IBM Research Report*.

[Gan] D. Gangopadhyay, S. Mitra, S. Dhaliwal. 'ObjChart-Builder: An Environment for Executing Visual Object Models', In *Proceedings of TOOLS-12 Conference*, Prentice-Hall (August 1993).

[GanWu] D. Gangopadyhay, P. Wu. 'An Object-Based Approach to Clinical Procedure Automation', In *Proceedings of 5th Symposium on Computer Applications in Medicine*, American Association of Medical Informatics (October 1993).

[Moh95] C. Mohan, A. El Abbadi, D. Agrawal, R. Gunthor, G. Alonso, M. Kamath. 'Exotica: A Project on Advanced Transaction Management and Workflow Systems', *ACM SIGOIS Bulletin* (August 1995).

[WFMC95] Workflow Management Coalition. 'Workflow Management Application Programming Interface (Interface 2)', document number WFMC-TC-1009, *http://www.aiai.ed.ac.uk/wfmc* (November 20, 1995)

[WFMC96] M. Anderson. 'Workflow Management Coalition Draft Workflow Standard - Interoperability Abstract Specification', document number WFMC-TC-1012, *http://www.aiai.ed.ac.uk/wfmc* (June 3, 1996).