An Approach to Middleware for Repeatable Collaborative Processes

Thomas Gregory¹
tom.gregory@GroupSystems.com
¹GroupSystems.com
1430 E. Ft. Lowell Rd.
Tucson, Arizona 85719

Robert O. Briggs¹,²
bbriggs@groupsystems.com
²University of Arizona
Tucson, Arizona 85721

Abstract

Research suggests that in order for GSS to be deployed successfully over the long term, it must be integrated into the daily work practices of an organization, in support of repeatable collaborative tasks conducted by practitioners rather than facilitators. However current GSS technology is not optimized to support repeatable tasks. This paper describes a GSS architecture that integrates five classes of middleware on a universal data model. The purpose of the new architecture is to provide a fast, open, scalable, flexible, extensible platform for rapid development and deployment of customized collaborative applications for mission critical tasks. Early field results suggest the architecture may live up to its promise, but more experience is needed to confirm early indications.

1. Introduction

Research in the lab and in the field suggests that, under certain circumstances, teams can be far more productive when they use a group support system (GSS) than when they do not (See Fjermestad and Hiltz 1999, 2001 for exhaustive compendia of GSS research). Despite its demonstrated usefulness, however, GSS has been slow to gain widespread acceptance in the workplace. Briggs, et al (1999) used the Technology Transition Model to argue that part of the reason the technology has been slow to spread might be that it is often deployed in centers of excellence, in support of teams working on ad hoc, non-routine problems. Such implementations of GSS, they argued, would almost surely be self-extinguishing, and the more successful was the initial rollout, the faster the GSS was likely to die. A solution, they argued might be to shift the focus of GSS away from custom, facilitated, one-off projects toward support for frequently repeated mission-critical tasks that were conducted by the practitioners themselves instead of by outside facilitators.

Existing GSS technologies tend not to be optimized for supporting repeatable methodologies and routine work processes. They offer a myriad of configurable features, which a skilled facilitator can use - in combination with carefully framed instructions - to create a wide variety of useful patterns of thinking among people working together toward a goal (Briggs, et al, 2001). Typically, however, current GSS cannot be easily set up to guide a team of non-facilitations step-by-step through a rigorously designed repeatable process. The GSS must be configured and re-configured on the fly by a human facilitator who guides the group through the nuances of a task.

Current GSS also tend to be closed applications, offering their users little ability to access data from outside applications. Yet, if a collaborative application is to be embedded into the daily work processes of an organization, it must do more than to allow people to interact with one another. It must also allow them not only to bring together their brains, but also the data from outside sources so they can make decisions and take action based on complete information. Collaborative technology must be an integral part of an end-to-end solution for the task at hand.

In order for a GSS to be integrated into many different routine work processes, it would have to be able to accommodate the data structures and types of many third party applications, and yet, it would not be possible to know in advance all the different data structures the GSS might be called upon to support in the future. In this paper we suggest an approach to a universal data model that could accommodate any data types in any kinds of relationships without having to rewrite the application.

In order to support the vast variety of collaborative needs that might emerge in the future, a GSS would not only need to be powerful and flexible, it would need to be extensible. As the need for new software
services emerges, it must be easy and inexpensive to plug in new functionality without disturbing or disrupting the existing functionality. In this paper we describe an integrated middleware architecture for collaborative technology. The architecture is specifically optimized to support repeatable collaborative processes, with a specific focus on open extensibility and the universal data model. We have dubbed this architecture “Apollo” in honor of the men and women of the space program, who worked together against seemingly impossible odds to solve seemingly impossible problems.

1. Middleware and Collaboration

Middleware is an umbrella term for a variety of technologies that links a client application with one or more server applications. For example, in order to bring together information from multiple sources, a web server might use Open Database Connective (ODBC) middleware to access third-party databases.

There are five general categories of middleware (Hailstone, 2001; Heinman & Byron, 2001).

- Object Oriented Middleware (OO)
- Data Access Middleware (DA)
- Transaction Processing Middleware (TP)
- Message-Oriented Middleware (MOM)
- Remote Procedure Call (RPC) Middleware

Object oriented (OO) middleware is designed to link object oriented clients with object oriented business objects in a language and location transparent manner. Some of the competing standards for OO middleware include Common Object Request Broker Architecture (CORBA), Microsoft’s Distributed Component Object Model (DCOM) and Java’s Remote Method Invocation (RMI). Because collaborative technology depends heavily on communication among remote objects, and because those objects are likely to be widely varied over time, OO middleware could help reduce the complexity of extending GSS software. However, OO middleware tends to have limited scalability, so a strictly OO middleware approach to collaborative technology would mean that the number of users who may simultaneously interact with an object would be limited.

Data access middleware is one of the most common types of middleware. It is used to create a common layer of abstraction between applications requiring data from a database and the data source itself. Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) are the most common examples of data access middleware. Other types of data access middleware may be used to bridge an application with an object-oriented database. By using data access middleware, a collaborative system could be deployed on whatever corporate database were available, gaining benefits like automatic back-up, guaranteed transaction integrity, and disaster recovery. A GSS using DA middleware could be ported from data source to data source without affecting the client program. However, such middleware would not be able to provide access to proprietary data sources from third party applications.

Transaction processing (TP) middleware is specifically designed to coordinate complex transaction across multiple data repositories. For example, many TP monitors communicate with multiple data sources through the use of Open Group’s XA interface standard that bridges the TP monitor and the target resource manager (usually a database). By using TP middleware, a GSS developer provides the ability to coordinate complex transactions among its own many collaborative components and among third party applications and data sources. While linking disparate applications through TP typically requires the use of a non-standard programming language, it can minimize the effort required to tie a GSS into mission critical corporate systems.

Messaging Oriented Middleware (MOM) is one of the more popular types of middleware used in the integration of legacy systems. This, combined with data transformation engines, allows applications to communicate freely with push, rather than pull protocols. For collaborative technology, this is an advantage because new contributions to a GSS must be updated in real time on other users machines. However, a pure MOM architecture would have limited scalability because message traffic would increase exponentially with each added user. Teams of 10-20 might experience snappy performance, while teams of 100 or more might bring a network to its knees.

Remote Procedure Call (RPC) middleware allows peer-to-peer function calls to be made from a client application to a distributed host application. RPCs follow a request/response paradigm. RPC technologies are at the heart of most OO middleware implementations. They allow a client application to invoke methods (procedures or functions) in a remote application. While RPC can be difficult to create, it would allow a GSS designer to place processor-intensive services on high-powered remote machines, minimizing the footprint of the client side. However, to some extent those advantages are offset by the network-intensive nature of RPC. A GSS designer must strike a careful balance between functionality and bandwidth.

Each class of middleware offers a distinct solution to a specific type of problem encountered by GSS developers. When making a middleware choice, one must
consider scalability (number of simultaneous users supported), need to access disparate data sources, desire to use object oriented programming techniques, bandwidth constraints, and the need for real-time updates among interfaces. No single class of middleware can be sufficient to address all the challenges associated with GSS applications. An architecture that borrows from all classes is required. For example, MOM might be used to enable communications between group members, allowing contributions to be sent to each participant. Data access middleware might be applied to storing and retrieving contribution data in one or more databases. TP might be used to open communications between a GSS and third party applications. OO middleware might be used to allow collaborative objects of the GSS to interact, and RPC might be used to strike a balance between bandwidth and processor constraints.

There are many commercial middleware products available in each of the five classes, and several which integrate two or more classes of middleware in the same product. However, these products tend not to be optimized to the special demands of GSS implementation. We therefore propose an architecture that draws on all these classes in ways conducive to GSS design. Because GSS must become embedded in day-to-day work processes of an organization, and because the designer of a GSS cannot know in advance what kinds of data in what kinds of relationships a team may need in the future, the architecture must begin with a universal data model.

2. A Universal Data Model

There are many ways on could implement a universal data model – in a relational databases, in flat files, in a persistent object stores, and so on. The principles underlying a universal data model are as follows:

1. Data stores are provided for all computer types of data – string, integer, floating point, blob, etc.
2. Relationships among data are themselves stored as data at runtime, not permanently cast at design and compile time.
3. Relationships are established through indirection rather than directly.

Consider these principles in order. First, it is axiomatic that if a data model is to be considered universal, it must be able to accept digital data in any form.

Second, in most systems, the relationships among data elements are established at development time. In a relational database system, for example, relationships are...
usually established in the metadata. Address is attribute-of employee. Employee ID is a foreign key in a paycheck relation, and so on. With a universal model, the name of the employee would be stored as data, the address of the employee would be stored as data, and the fact that a particular contribution was in an address-of relationship with some other contribution would also be stored as data.

Third, in order to prevent certain update anomalies and certain referential integrity problems, it is useful to establish relationships among pointers to data elements, rather than directly between the elements themselves. For example, the data for a requirements negotiation might include a win condition with two issues and a single option that resolves both issues (Figure 0a). Depending on their roles, different users might see different views of those data (Figure 0a, 0b). User 1 might see the win condition, the first issue, and the option. User two might see the same win condition, with Issue 2, and the same option. If user 2 were to delete Issue 2 and its child option, the legitimate relationship between Issue 1 and Option 1 would be lost. However, if all relationships are established indirectly through pointers (Figure 0c) then the legitimate relationship between Option 1 and Issue 1 would be preserved, even if Issue 2 and Option 1 were deleted in another view.

Many current-generation GSS store data in a relational model representing lists and trees. Some also accommodate directed graph data structures where relationships may fall outside of sibling or ancestor chains.

Figure 1 is an example of how contributions are related via sibling chains. Figure 2 introduces the ancestor relationship. However, it is not sufficient for a universal data model to be able to represent a particular kind of data structure. It must be possible to create specific instances of such relationships on the fly. For example, an internal audit team might need a GSS to present them with a tree made up of business processes at the top layer and risks on the second layer, with a risk-of relationship between them. A universal data model for GSS must be able to accept such new contribution types (Process, Risk) and new relationships (Risk-of) on the fly without requiring the intervention of a programmer. In GSS, order of data presentation is as important as the content of the contributions. There is meaning in order and relationship. Order may be chronological, alphabetic, priority, importance or some other arbitrary ranking. The challenge then becomes, how to relate contributions when either sibling or ancestor relationships don’t convey an appropriate relationship.

Figure 3 provides an additional method by which to relate contributions. It relates objects not simply by some fixed order as in the previous examples, but rather allows for an unlimited number of relationship types. Finally, by allowing the melding of both of these models, one can start to achieve flexibility required by an extensible GSS application. Figure 4 shows how this might be used. Figure 4 illustrates how contributions can be related using all aforementioned relationship types. Notice that relationships are directional. For example, Contribution 2.1 relates to Contribution 3 via a relationship. Contribution 3 does not have a relationship back to Contribution 2.1.

The next step in providing a flexible GSS data model is providing a method by which contributions can be categorized by type. For example, a contribution could be categorized as a document while another contribution...
may be categorized as a *thought*. The contribution type does not drive the data type of the contribution itself, thereby providing a mechanism by which to group like contributions.

This capability becomes exceedingly important when incorporating data from external systems. If contributions are simply submitted into the GSS without some sort of typing, then segregating the data into meaningful groups becomes nearly impossible. Take the following as an example.

Figure 5 above demonstrates how contribution types might be used to group together Animals and Fish. The contribution type categorizes the being, and the contribution qualifies the being. By using contribution types, we can easily group contributions and if needed, present like contributions together on a screen or a report.

The last element that is important when designing a universal data model is the concept of contribution *attributes*, which are used to further qualify the contribution. Given the fact that the GSS application author cannot anticipate to which degree data must be either qualified or described, then there is a need for the ability to have an unlimited capability to add contribution attributes. Figure 6 shows how this might work.

Figure 6 incorporates attributes with contributions that can be used to further qualify contributions. For example, by inspecting the contribution attributes, we can come to learn whether the contribution is *deadly*, will *scratch*, its *color*, and in the case of the fish, whether they are good to eat.

By combining the flexibility of establishing the relationships among the contributions along with assigning contribution types and adding the ability to qualify a contribution, the GSS developer will be able to represent virtually any piece of data and relate it in such a way that it has meaning to the users of the GSS.

3. Putting it all Together

After creating a universal data model, the next challenge is to provide a mechanism by which contributions can be sent into the data model, and users of the GSS can be presented with just the right contributions, in just the right format at just the right time.

A collaborative process will typically be comprised of multiple steps. For example, Step 1 may be to brainstorm and to collect ideas. Step 2 might be to converge on a subset of the brainstorming ideas. Step 3 might be to rank the contributions against some criteria.

![Figure 3. Directed Graph Relationship Model](image)

![Figure 4. Sibling and Ancestor chains with directional ad-hoc relationships](image)
In order that the team may accomplish its task, the GSS should provide just the right user interfaces, configured in just the right way to create the group dynamics and patterns of thinking required for each step. The contributions collected in Step 1 must move seamlessly to Step 2. The results of Step 2 must feed the polling in Step 3. It must be possible for an individual to modify a contribution in Step 2, and for the users who might still be working in Step 1 and Step 3 to see those modifications in real time.

In order to accomplish this, the data model of the GSS must be completely independent of the software that create and access it. The collaborative interfaces must serve as filtered views of the larger data set, and the data must not “know” anything about the interface from which it initially came. It would be possible for a GSS designer to implement this through a key feature of MOM middleware, a publish-and-subscribe (P/S) capability. With a P/S implementation, each collaborative interface would register with the server, and subscribe to certain data types (Ex. Give me all the business processes, and give me all the risks that are child-of business processes). The server would then monitor all incoming contributions, watching for incoming processes and risks (child-of processes). Any that arrived would be immediately forwarded to all who had subscribed for those types. For example a user may subscribe to contributions of type, Animal. If a contribution of that type with the content, Cat, is modified, then any user subscribed to Animal will receive an alert that Cat has changed. Additionally, if a user were to add Carp, then all subscribers to Fish would receive the notification of the addition. This ability to proactively alert a user of to the presence of a new contribution is called push technology. By using a P/S strategy, data communication is kept to a minimum, because update messages only go to those who have subscribed for the type involved in the transaction.

Data access middleware may be included in the GSS to persist its data. If designed appropriately, a GSS can take advantage of a simple ODBC or JDBC connector to a relational database for the storage of process meta data and/or contributions.

Given the highly varied possibilities of a universal data model, an implementation on a relational store might suffer slow performance. By integrating with an object store, a GSS could take advantage of maintaining the integrity of the object model without having to translate the object model to the relational model and back. However, a commercial relational database can provide automated features like guaranteed transaction integrity, automated backup, and disaster recovery. It is possible to combine the best of both approaches by operating in memory against an object model, but having a separate service that converts the object model into a relational model and persists it to a database in near real-time. Thus, both performance and integrity can be obtained.

Transaction integrity is a very important aspect of GSS and must not be ignored. For example, it must be possible to prevent multiple users from accidentally overwriting one another’s work when all choose to edit the
same contribution at the same
time. Therefore, a GSS design
must consider how it will ensure
data integrity through the use of
some transaction processing
mechanism.

If transaction integrity is
going to be maintained, it must be
performed at the contribution
level. This means that no two
users can access the same
contribution simultaneously for
editing. This means that data
must be centralized so that an edit
lock can be requested and granted
directly on the item that is going to
be modified. Until the
modification is complete, all edit
lock requests must be rejected.
Once the edit is complete and the
lock released, the object once
again becomes available for
locking and modifying.

The final middleware
cOMPONENT that needs to support a
flexible GSS is the RPC
mechanism. Not only do GSS
end-user clients need access to
different business rule objects, but
external systems also need access.
Therefore, a flexible GSS needs to
accommodate a communication
vehicle that is both robust and
extensible.

XML provides this
required robustness and
extensibility. By utilizing XML as
the remote procedure call protocol,
one standardizes the way by which methods or procedures
are accessed within the GSS. The GSS end-user client
would communicate with GSS methods by using the same
set of XML-based RPC messages, as would an external
system.

The transport layer over which the protocol
would run is wholly independent. Regardless of whether
the XML runs over TCP/IP sockets or named pipes, the
RPC mechanism remains in tact. Additionally, external
MOM can be used to integrate with third party systems.
Any system capable of connecting to MQSeries or
MSMQ, and capable of either generating XML or having
its messages transformed into XML could communicate
with the GSS.

4. A Practical Implementation

Figure 7. illustrates a practical approach to a GSS
architecture that takes of an object oriented universal data
model, a relational persistent store, and five kinds of
middleware in a flexible and extensible GSS solution. We
have created a prototype implementation of this
architecture, and it is currently in benchmark testing. At
the top most layer, the GSS is accessible through plug-in
communication modules. The current implementation
includes IP/Sockets, Named Pipes, DCOM/CORBA, and
message queues. All these different communication types
connect to the GSS through a common communications
layer.

At the heart of the GSS platform, is the Core
Dispatch Services module. It is responsible for
coordinating the efforts among the XML Transformation Services and the P/S Controller. Additionally, it performs all routing to the RPC Services layer.

Below the RPC Services are plug-in modules of additional functionality required by a flexible and extensible GSS. These include the GSS business rules, back channel communications, security service and any additional data services required by the system. The Universal Data Model resides under the GSS business rules given that the GSS business rules control the behavior of the Universal Data Model.

Benchmarks of the early prototypes are promising. Early test suggest that scalability speed will exceed earlier, more limited applications of middleware to GSS. By implementing a caching mechanism for the universal data model, we have been able to achieve a very high transaction rate. Additionally, when storing to a persistent store such as a relational database, we have achieved transactional integrity by incorporating a transaction-processing object within the Universal Data Store, without suffering an unacceptable degree of performance degradation.

Early concerns about the performance of the universal data model have proven unfounded. By implementing the object model in memory and the relational model in the background on disk, we have achieved access speeds comparable to hard-coded data models. The flexibility of the model has enabled the development of 14 core plug-in collaborative tools over the past 6 months, and hundreds of special purpose tools snapped-together from the core tools. In some cases we have been able to cut the development cycle for new collaborative tools from months down to hours, creating several new tools per day. To date, we have not found any data structure we cannot accommodate with our implementation of a universal data model. By supporting lists, trees and complex graphs, with user-defined types and user-defined relationships, we have been able to accommodate every need to date.

Lastly, by using a publish-and-subscribe mechanism for pushing data to clients, we can minimize the numbers of messages that must be created and sent. Depending on the type of client and the type of communication protocol, the client may have to poll for a message, as in the case of a client using the HTTP protocol, or may be asynchronously sent to the client as in the case of IP/Sockets. Regardless, early research shows that the communications, while adding some latency in sending information back to the client, performs in a reasonable manner.

While early research is promising, additional research will be required to prove the architecture and the theories on which it is founded. For example, the universal data model has not yet been stressed by simultaneous access by hundreds or thousands of users. Nor has it been exhaustively demonstrated that it can, in fact, accept any data from any third-party application and return it intact.

The feature-and-function set of the prototype is far from complete. It is not clear at this juncture what features and functions might be missing, and what unexpected consequences might derive when new functionalities are added. The plug-in aspect of collaborative services modules have been tested, but not stressed. Although we implemented messaging in XML, XML tends to be bulky. It is not yet clear whether XML can support the volume of transactions that a large-scale GSS will require. It may be necessary to tokenize XML commands in order to accommodate bandwidth constraints.

The new architecture was deliberately implemented to be open at every level to encourage third-party development of new services, data interchange modules collaborative tools, and collaborative applications. It is our hope that it will give rise to a robust aftermarket where facilitators and collaborative technology developers exchange freeware and shareware, deploy, and in some cases, commercialize their own additions to the architecture.

References


Briggs, R.O.; de Vreede, Gert-Jan; Nunamaker, J.F., and Tobey, David H. ThinkLets:
Achieving Predictable, Repeatable Patterns of Group Interaction with Group Support Systems (GSS)  
