

Token Dispensers for GSS Race Conditions: Locking Things That Don't Exist

Robert O. Briggs
University of Arizona
Delft University of Technology
GroupSystems.com
bbriggs@cmi.arizona.edu

Tomas P. Gregory
GroupSystems.com
1430 E. Ft. Lowell Rd.
Tucson, AZ 85719
tgregory@groupsystems.com

Abstract

A technological challenge that distinguishes group support systems (GSS) from many other technologies is the need to accommodate a variety of race conditions. A race condition occurs when two or more users of the same shared data object execute processes on that object that could produce conflicting results. Some particularly challenging race conditions obtain when users contend for the right to create resources that do not yet exist. Resources that do not yet exist are not available to be locked. This paper offers token dispenser technology as a general-purpose solution for addressing a variety of race conditions, including those where end user clients contend for resources that do not yet exist. The paper describes the details of an implementation of token dispensers as a plug-in service under the Apollo architecture (an open architecture for collaborative clients and servers). The paper summarizes the results of field trials of the token dispenser implementation by developers and end users. It concludes with discussion of future directions for token dispenser research.

1. Introduction

A technological challenge that distinguishes Group Support Systems (GSS) from many other technologies is the need to accommodate a variety of race conditions. With GSS, each user has a different computer, but all users may contribute to and modify the same data set at the same time. Each user has an independent cursor in the same shared data. *Race conditions* obtain when two or more users simultaneously execute processes that may produce conflicting results. For example, one user of a shared outline tool might move an item to the top of the outline at the same moment another user moves

the same item to the bottom of the outline. Or one user might choose to add sub-headings to an item just as another user chooses to delete it.

Race conditions are exacerbated by network latency. Changes made by one user may not instantaneously update on the screens of other users. Working over the Internet, for instance, if a user were to move an item within a shared outline, it might be several seconds before the change was reflected on the screens of all other users. During that time any number of users might have initiated any number of transactions on that same contribution.

The technical challenge of GSS is further heightened because some race conditions obtain for the opportunity to create resources that do not yet exist. For example, a shared form tool might present the contents of a record that all users must share. Before shared work could begin, however, some user or some automated process would have to create the shared record. If two users enter the system simultaneously, both might send the command to create the shared record, giving rise to two records where only one is wanted. Because the record does not yet exist, it is not possible for either user to lock it, preventing the unwanted duplication.

The sophistication and usability of a GSS is, to a large extent, bounded by the way it handles race conditions. This paper describes a server-side technology called a *Token Dispenser* that can be used to address a variety of GSS race conditions.

It is important to advance the state of the art of GSS because research shows that, under certain circumstances, people using Group Support Systems (GSS) can be vastly more productive than people who do not use GSS (See Fjermestad and Hiltz, 1998; 2000 for) for a comprehensive list of GSS

research findings). Under other circumstances, teams using GSS are not as productive as those who use other means (For example, Pinnsennault, et al, 1999). As with any kind of tool, the effects produced by GSS depend on the characteristics of the technology and the way it is wielded (Briggs, Vreede, Nunamaker, & Tobey, 2001). Advancing the state of GSS technology may broaden the variety of circumstances under which users may derive the benefits of GSS.

Advancing the state of GSS may also increase the value it can provide in the field. Despite field results that demonstrate significant benefit when GSS is applied to certain mission critical tasks (Adkins, et al., 1998; Quaddus, Atkinson, and Levy, 1992; Dennis et al., 1997; Davison, 2000; Vreede, & Bruijn, 1999; Vreede & Dickson, 2000) GSS has been slow to transition from the laboratory into the workplace (Briggs, Nunamaker, & Tobey, 2001). Authors reasoning from the Technology Transition Model; (Briggs, et al., 1998-99) have argued that GSS technology might diffuse more readily if it were deployed in task-specific collaborative applications embedded in day-to-day work routines, rather than as a general purpose tool kit for facilitators (Briggs, Nunamaker, & Tobey, 2001; Gregory & Briggs, 2002).

The first generation of GSS technologies did not lend themselves well to the packaging purpose-built collaborative applications. They tended to be integrated suites of tools that could be manually configured on the fly to suit the needs of the moment. Their stove-piped architectures did not allow their basic components to be re-arranged quickly to suit the needs of a specific task, nor for new components and capabilities be added without extensive development efforts. They could not easily interoperate with nor be integrated with third party applications.

The Apollo architecture for GSS applications (Gregory and Briggs, 2002) suggests an open, extensible approach to the rapid development of collaborative applications from elementary collaborative components. The architecture specifies loosely coupled snap-together modules joined by an open XML-based API at all layers. It provides for extensibility through a software bus by which new software services may be snapped into

the system without affecting existing capabilities. The Apollo architecture integrates elements of five classes of middleware to address the special challenges of collaborative work. The Token Dispensers described in this paper extend the Apollo architecture by specifying a new snap-in server-side capability to address GSS race conditions.

2.0 The Technical Challenge of GSS Race Conditions

When two users cause a race condition by simultaneously moving the same item to different locations in an outline, a *last-in-wins* strategy might be a sufficient response. The system could place all incoming transactions on a queue, and execute them in order. In the example given above, it could move the item first to the top of the outline, then to the bottom. The outline would finish in the state dictated by the most-recently executed command (hence the label, last-in-wins).

A last-in-wins strategy might be less successful when a dozen users seek to edit the same block of text simultaneously. Each might devote considerable time and effort to crafting a new version of the text. As each user submitted her revision, the new version would overwrite the revisions that had gone before it. For simultaneous editing, *Edit Locks* with a *first-in-wins* strategy might be more useful than last-in-wins. All dozen users might request an edit lock on the text simultaneously. Their requests would go into a queue, and the server would grant an edit lock with the first request it processed. All other requests for permission to edit the text would be denied until the first user released the edit lock.

However, in order to lock an item for editing, the item must already exist. There are other race conditions that obtain with GSS where a resource in contention does not yet exist. Consider, for example, the shared form tool mentioned above. A team might use such a tool to track and update the status of various mission-critical resources (Figure 1). All users would view the same data record at the same time. However, when the first user opened the tool for the first time, that user (if record creation is triggered manually), or that user's tool (if record creation is triggered automatically) would find that no record yet existed, so record creation would be

triggered. A race condition would obtain when two or more users started working on the task at nearly the same time. Each user (or each user's tool) would find the database empty, so each would create the first record and submit it to the database. Even though the task requires that all participants share the same single record, the database would fill with redundant records.

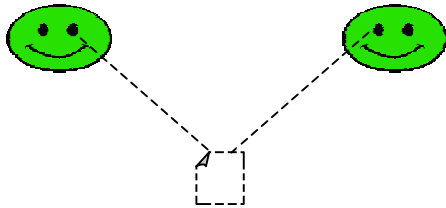


Figure 1. Contention for a resource that does not yet exist. Shared Record. In this example a team might create a scoreboard to track the status of key resources. The tools of all users would present the contents of the same record to the all users. When no record exists, the tool of the first user to enter the system could create the record automatically. However, a race condition obtains when two users enter the system simultaneously. The tools of both users find that no record exists, so both tools create the "first" record, resulting in multiple records when only one is wanted. It is not possible for the first user to lock the first record to prevent the second user from creating it, because the record does not yet exist to be locked.

Standard database locking mechanisms would not be sufficient to prevent this problem, because, until a key exists for a record, the database cannot lock it. Database keys are typically generated by the server, so until the record is submitted by the client, the database cannot lock it, so multiple submissions of the first record would still be possible. Occasionally, a database system is set up so that a client side, rather than by the server side, generates a record key. In that case, there would be no way for both clients to generate the same key, so each would create and receive a lock on a different record key, and the multiple-record problem would still prevail.

A similar contending for resources that do not yet exist occurs if the client of a collaborative tool supports seed data. Seed data are data that must be automatically entered into a shared tool if no other

data are present. Consider the case of a collaborative Action Item tool like that modeled in Figure 2. The tool has two collaborative elements – a list interface and an outline interface. Users may enter action items into the list interface. Double-clicking any Action Item opens an outline interface, where the users may negotiate and record the details of the action item. However, suppose the designer of the tool did not want participants to start with a blank outline. Suppose, instead, that the first time a user opened the outline under a given action item, the tool was configured to automatically enter the following headings:

Action:
 Leader:
 Start Date:
 Deadline:
 Deliverable:
 Deliver to:
 Measures of Merit:

A race condition would obtain when two or more users double-clicked the same action item at about the same time. Finding no contributions in the outline, each user's tool would enter the seed data. Multiple copies of the seed data would therefore appear in the outline.

In such cases, last-in-wins, first-in-wins, and edit-lock strategies will not suffice. The GSS client must be able to request and receive a lock on the opportunity to create data that do not yet exist. A *token dispenser* can fulfill this need.

A token dispenser provides a general semaphore capability for addressing certain GSS race conditions. The purposes of a token dispenser are:

- To grant a user permission to access a resource
- To grant a user the opportunity to create a resource
- To distribute existing resources evenly among the users of a GSS.

A token dispenser operates by responding to messages from client software requesting tokens from a particular token dispenser. To clarify the

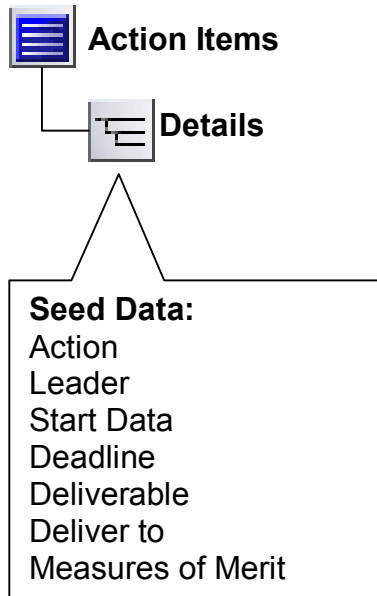


Figure 2. Contention for a Resource That Does Not Yet Exist. A simple collaborative action item tool with “seed data”. Users contribute action items into the top-level list interface. Double-clicking any action item opens an outline interface into which team members may enter the details of the action item. However, the users do not start with a blank outline. If no contributions appear in the outline, the interface is configured to automatically contribute its own “seed data” of seven main outline headings. A race condition obtains when two (or more) users open the outline simultaneously. Both user’s interfaces, finding no data, may simultaneously contribute seed data, resulting in multiple copies.

purposes of a token dispenser, consider the following simplified, humanized examples of how the dialog between collaborative client software and a server-side token dispenser might transpire:

Example 1. Locking an existing resource.

Client1: , I’d like an exclusive token to edit Item 100

Dispenser1: You have an exclusive token to edit for Item 100.

Client 2: I’d like an exclusive token to edit Item 100

Dispenser1: Client1 has an exclusive edit token for Item 100. Your request is denied

Client 2: Darn.

Example 2. Locking the opportunity to create a resource

Client3: I’d like to add seed data under ActionItem 312

Dispenser2: You have an exclusive token to add seed data under ActionItem 312

Client4: I’d like to add seed data under ActionItem 312

Dispenser2: Client3 holds an exclusive token to add seed data under ActionItem 312. Your request is denied.

Example 3. Locking the opportunity to create a resource

Client 5: I’d like to add the first record to the data set called “Critical Resources”

Dispenser 3. You have an exclusive token to add the first record to the data set called “Critical Resources

Client 6. I’d like to add the first record to the data set called “Critical Resources”

Dispenser 3: Client6 is already adding the first record to that data set. Request Denied.

Example 4. Distributing resources among users

Client7: I wish to share the use of one of the following resources: 1, 2, 3, 4, 5, 6

Dispenser4: Resource 3 has the fewest current users. You are added as an owner of the Resource 3 token.

Client8 : I wish to share the use of one of the following resources: 1, 2, 3, 4, 5, 6

Dispenser3: Resource 3 is getting crowded. You are added as an owner of the Resource 6 token.

3.0 Components Token Dispenser Technology.

A Token Dispenser implementation has three components (Figure 3):

- Tokens
- Token Dispenser
- Token Service

A *token* is a named object that represents a resource for which users may contend. The attributes of a token are

- Token Name
- Token Type (Exclusive vs. Shared)
- Token Owner List

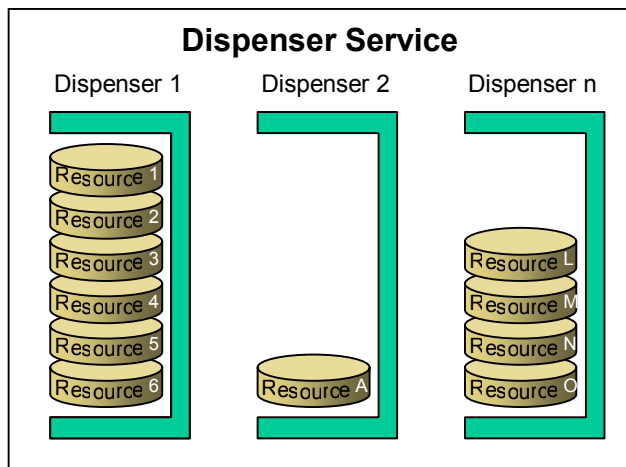


Figure 3. Dispenser Service. The Dispenser Server is a plug-in service of the Apollo Collaborative Server Architecture. At the request of user tools, it creates any number of named dispensers. Each dispenser represents a context in which contention for resources must be managed. Each dispenser can create any number of named token. Each token represents a resource for which users may contend. A user's tool can request a token for a resource that already exists, or for the opportunity to create a resource that does not yet exist.

A token's *name* uniquely identifies the resource or opportunity it represents. A detailed discussion of token naming conventions appears in Section 4. A token's *type* may be designated as either exclusive or shared. If a token is designated as exclusive, then the token dispenser will grant ownership of the token to only one user at a time. The token dispenser will deny all subsequent requests for the exclusive token until its owner releases it. If a token is designated as shared, the token dispenser will grant ownership of the token to more than one user, following a distribution algorithm like the one detailed in Section 6. A token's *owner list* is a vector of the IDs of the user or users to whom token ownership has been granted.

A *token dispenser* is a named object that can create, persist, modify, and destroy tokens. The token dispenser contains logic for granting or refusing to grant ownership of exclusive tokens to users, and for distributing ownership of shared tokens among multiple users. Each token in a dispenser represents a unique resource for which users might contend. The only attribute of a token dispenser is its name. The token dispenser's name indicates the *context* in which the resources it contains are to be managed. A detailed discussion of dispenser naming conventions appears in Section 5.

The *token service* is a server-side capability that can create, maintain, and destroy token dispensers. The token service would be instantiated as a plug-in module of a collaborative server built under the Apollo Collaborative Server Architecture. It could be implemented in many other ways under other architectures. A description of an Apollo implementation of a token service appears in Section 6.

4.0 Token Naming Conventions

A token's name must provide *sufficient information to uniquely identify the resource or opportunity for which users may contend*. A developer may assign any string as the name of a token.

If users contend for an existing resource, say, for edit rights to an existing record, then the record ID would suffice as the token name. Likewise, if the propose of the token dispenser were to share records among a set of users, then the record ID would

suffice as the name for each token. If, on the other hand, finer-grained locking were needed, say, to edit lock a particular field in a record, then the token name might be a concatenation of the record ID and the field name.

If, instead of locking an existing record, a user wished to lay claim to the opportunity to create the first record in a data set, the token name might be the name of the data set in which the first record was to reside.

5.0 Token Dispenser Naming Conventions

A Token dispenser's name must provide *sufficient information to uniquely identify the context in which users may contend for the resources its tokens represent*. A developer may assign any text string as the name of a token dispenser. In the case where users contend for the right to create the first record in a data set, the developer might choose to create a dispenser named CreateFirstRecord. Such a dispenser might hold a token for any data set to which a user (or a user's tool) claims the opportunity to add the first record.

Depending on circumstances, it might be desirable to have a separate dispenser for each data set rather than one dispenser for all data sets. In that case, one might choose to name with a concatenation of a string like CreateFirstRecord and the name of the dataset the dispenser will control (for example, CreateFirstRecord*Dataset1). One might make a choice like this if one wanted to be able to destroy the dispenser for one dataset without affecting the disposition of resources for another dataset. In this case, the name of the data set might also serve as the name for a token.

It falls to the developer of a collaborative tool to design the naming conventions for tokens and token dispensers so that all client software will derive the same name for the same context and for the same resource or opportunity. If the dispenser name were to define the context too narrowly, then race conditions might go unaddressed. For example, if a dispenser that combined the string, "EditLock" with the current day of the week, (for example, EditLock*Monday) then a second person could obtain an edit lock on a locked item just after midnight. If the context for a dispenser is defined

too broadly, then users might be blocked from legitimate simultaneous actions.

6.0 An Implementation of a Token Dispenser Service

This section describes a token service implemented by the authors as a server-side plug-in module for a collaborative server built with the Apollo architecture. In this implementation, the token service responds to only three messages from the client software:

- Request Dispenser
- Release Token
- Drop Dispenser

6.1 The Request Dispenser Message. The Request Dispenser command both creates a dispenser and requests that the dispenser grant tokens with certain names. If the requested dispenser does not exist, the dispenser service creates the dispenser, and passes it the request for tokens. If the requested dispenser already exists, the dispenser service simply passes the dispenser the client's request for tokens.

The Request Dispenser Token message includes four parameters –

- Dispenser name
- List of requested token names
- Type of token requested (exclusive or shared)
- UserID

The dispenser service processes a Request Dispenser message according to the following algorithm.

```

IF token dispenser of this name does not
exist
THEN
    Create token dispenser
END IF
Pass the message to the named dispenser
Wait for the response message from the
dispenser
Return the response message to the client
    
```

The token dispenser processes the request dispenser command according to the following algorithm:

```

Remove the user making the request from
ownership of any tokens in the dispenser
    
```

```

IF any of the requested tokens do not exist
in the dispenser
THEN
    Create the tokens without owners
END IF
IF user requests exclusive token for a
resource
THEN
    IF requested token already has an
owner
    THEN
        Delete any tokens without
owners
        Return Failure Message
    ELSE
        Assign user as exclusive
owner to the requested
token
        Delete any tokens without
owners
        Return a Success Message
    END IF
END IF
IF user requests shared access to one of a set
of tokens
    Identify which of the requested
tokens has the fewest owners
    If there is a tie, randomly select one
of the tokens with the fewest owners
    Assign the user ownership of that
token
    Delete any tokens with no owners
    Return a success message
END IF

```

Notice that any tokens that have no owners are automatically deleted at the end of the any transaction.

6.2 The Release Token Message. The Release Token Command removes a user from ownership of a named token. The parameters of the command are:

- Dispenser Name
- Token Name
- User ID

The purpose of this command is to assure that the user does not hold ownership of the named token. Therefore, if the dispenser named in the command does not exist, the dispenser service returns a

success message. If the dispenser exists, but the named token does not exist, the dispenser returns a success message. If the named token does exist, but the user is not currently an owner, the token dispenser returns a success message. If the token exists, the token dispenser removes the user from the owner's list for that token and returns a success message. If the token has no more owners, the dispenser deletes the token. There are no failure messages for this command.

In this implementation, users can only remove themselves from ownership of a token. They cannot remove others from ownership of a token. However, the service is implemented such that, if a user drops off line, or logs out of the system, any tokens held by that user in any dispenser are automatically released.

6.3 The Drop Dispenser Message. The purpose of the drop dispenser command is to assure that a named dispenser does not exist. When the dispenser service receives a drop dispenser message, it checks to see whether the named dispenser exists. If not, it returns a success message. If so, it deletes the dispenser with all of its tokens, and returns a success message.

Any user's client may drop any dispenser, regardless of which user created it. This is an important capability, because the person who creates a dispenser may not be on line when the time comes that a dispenser must be removed. Consider the case mentioned above of the tool that uses a shared record. Before creating the first record, a user's client creates a dispenser and secures an exclusive token on the opportunity. That dispenser remains throughout a session to prevent other users from accidentally creating an additional record. Suppose however, that a user deletes that record for a good reason. Now, the record is gone but no user can create a new record because the dispenser lock on the first-record opportunity still exists. To prevent this unwanted blocking, the command to delete the last remaining record should be accompanied by the command to drop the dispenser lock for creating the first record.

When the last owner of a token relinquishes ownership, the token automatically deletes itself from the dispenser. When the last token is removed

from a dispenser, the dispenser automatically deletes itself from the token dispenser service.

In this implementation, dispensers persist only for the duration of a session. When the last user logs out, all dispensers are dropped, because no opportunities for race conditions remain. When users log in again, new dispensers are created as needed.

7.0 Field Trials

To test the usefulness of the server-side token dispenser technology, we incorporated it into a general purpose collaborative server using an XML messaging API, and made the server available to 30 developers, who produced 13 different collaborative tools based on this technology. The experience level of the developers ranged from novice to expert. Developers reported that the three-message API was simple to use and that the run-time capability was robust, and that the availability of the service reduced their development time.

There was initial concern that network latency over the Internet might create unacceptable delays for end users while the client software waited for responses to its requests for tokens from the server. To conduct usability testing, we installed servers in Tucson, Arizona, Annapolis, Maryland, and Bangalore, India. Approximately 100 end users from Europe, India, and the United States connected to servers at all three sites over a period of 3 months via a variety of channels. Tests with high-speed Internet (1MB/SEC or higher) produced no noticeable delays for users who were physically separated from the server by 3,000 to 12,000 miles. Users on dial-up connections (28.8 KBS) reported latency delays in the half-second range. Users on very low speed connections (9 KBS or less) reported latency delays up to 2 seconds. All users reported that the delays did not disrupt their work.

8.0 Future Directions

The token dispenser implementation reported in this paper provided proof-of-concept and proof-of-value for a general-purpose semaphore service to address a variety of race conditions that that can occur in client-server collaborative technology. Most

usefully, they can be used to lock the opportunity to create resources that do not yet exist.

The implementation reported in this paper supports only two kinds of tokens – shared tokens, and exclusive tokens. The implementation hard-coded the logic for each of those tokens. However, to improve extensibility, it would be useful to implement an open-sided token dispenser that accepts plug-in logic modules. Each plug-in module could provide the logic for a different kind of token. In this way, it would be possible to extend the capabilities of the token service without having to recompile the server, and, indeed, without even having to stop the server to upgrade its token-handling capabilities.

The tokens in the implementation presented in this paper had very few attributes. The next generation of token dispenser technology must support tokens with an unlimited vector of attributes. Plug-in logic modules on the server side could then provide more sophisticated token-granting schemes based on the attributes of the resource or opportunity in contention. For example, consider the case of users working on an internal audit of business risks and controls. It might be useful to implement a token granting mechanism that gave a user the highest risk for which no controls had yet been implemented.

It may also be useful to provide a means to inform the token server of the attributes of the user who makes the request for a token. It would then be possible to implement a logic module that conducted more sophisticated matching of resources to the users who request them.

At some point in the future, it may be worthwhile to implement a token server capability where the logic for processing a token request arrives with the token request. Such a capability could support the addition of new client-side capabilities without any additional server-side capabilities. It may also be useful in the future to implement a capability where the resources themselves serve as tokens in the dispenser.

10. References

- Adkins, M.; Sheare, R.; Nunamaker, J.F.; Romero, J.; and Simcox, F. Experiences using group support systems to improve strategic planning in the Air Force.

- Proceedings of the Thirty-First Hawaii International Conference on Systems Sciences, 1998, 515-524.
- Briggs, R.O., Vreede, G.J. de, Nunamaker, J.F., and Tobey, D. ThinkLets: Achieving Predictable, Repeatable Patterns of Group Proceedings of the 35th Hawaii International Conference on System Sciences, IEEE, 2002, 1-10.
- Briggs, R.O.; Adkins, M.; Mittleman, D.; Kruse, J.; Miller, S.; and Nunamaker, J.F., Fr. A technology transition model derived from field investigation of GSS use aboard the U.S.S. Coronado. *J. of Management Information Systems*, 15(3) 1998-99, 151-196.
- Briggs, R.O., Nunamaker, J.F. Jr; and Tobey, D. The Technology Transition Model: A Key to Self-Sustaining and Growing Communities of GSS Users. Proceedings of the 34th Hawaii International Conference on System Sciences, 2001, 1-9.
- Davison, R. The role of groupware in requirements specification. *Group Decision and Negotiation*, 9(2) 2000, 149-160.
- Dennis A.R.; Tyran, C.K.; Vogel, D.R.; and Nunamaker, J.F. Group support systems for strategic planning. *Journal of Management Information Systems*, 14(1), 1997, 155-184.
- Fjermestad, J., and Hiltz, S.R. An assessment of group support systems experimental research: methodology and results. *Journal of Management Information Systems*, 15, 3 (Winter 1998-1999), 7-149.
- Fjermestad, J. and Hiltz, S.R. Group support systems: A descriptive evaluation of case and field studies. *Journal of Management Information Systems*, 17, 3 (Winter 2000), 112-157.
- Pinsonneault, A., Barki, H., Gallupe, B., and Hoppen, N. *Information Systems Research*, 10(2) 1999, 110-133.
- Gregory, T. P. and Briggs, R.O. An Approach to Middleware for Repeatable Collaborative Processes. Proceedings of the 35th Hawaii International Conference on System Sciences. IEEE. 2002. 1-10.
- Quaddus, M.A.; Atkinson, D.J.; and Levy, M. An application of decision conferencing to strategic planning for a voluntary organization. *Interfaces*, 22, 6 (1992), 61-71.
- Vreede, G.J. de, G.W. Dickson, Using GSS to Support Designing Organizational Processes and Information Systems: An Action Research Study on Collaborative Business Engineering, *Group Decision and Negotiation*, Vol. 9, No. 2, March 2000, pp. 161-183.
- Vreede, G.J. de, Bruijn, H. de, Exploring the Boundaries of Successful GSS Application: Supporting Inter-Organizational Policy Networks, *DataBase*, 30(3-4), 1999, 111-131.