SYNCHRONIZED DATA OBJECTS

Undergraduate Thesis
Department of Computer Science
Washington University in St. Louis

by Marin Bezic

Tuesday, September 05, 1995

Advisor: Kenneth J. Goldman

Abstract

Synchronized Data Objects (SDOs) are presented as a method of encapsulating, in the datatype definition, synchronization protocols that are used to control information exchange. SDOs are presented in the context of I/O abstraction, a programming model that seeks to separate communication from computation in order to support dynamic end-user configuration of distributed applications. SDOs can be used to implement a variety of synchronization paradigms, including remote invalidation, demand-driven data streams, remote procedure call, and promises.

An implementation of SDOs is described in the context of The Programmers' Playground, a distributed application development environment that supports the I/O abstraction programming model. Examples of SDOs for pairwise synchronization are presented, and generalization to other application development environments is discussed.

Keywords: distributed systems, synchronization, promises, futures, RPC

1. Introduction

Distributed programming has become increasingly utilized over the past years as the result of improvements in the network technology and emergence of multimedia applications. However, complex interprocess communication protocols make developing distributed and cooperative applications difficult. In this paper, we present the idea of encapsulating and abstracting communication protocols, thereby separating communication from computation.

Motivation: The Center for Air Pollution and Trend Analysis (CAPITA) heavily depends on data processing for air pollution research. We have developed a set of independent applications in Visual Basic for Windows that format, transform, and visualize data. In order to simplify and speed data processing, these applications have been enabled to exchange data among themselves. As a result, a data processing distributed pipeline application can be set up. It consists of one application that serves as data producer, a number of data transforming applications, and finally a data consumer application. The communication protocol is very simple and allows applications only to be connected in a form of pipeline (one input, one output), and does not take advantage of multiprocessing. Namely, only one application in a pipeline can run at a time.

Even though this pipeline implementation has served its purpose well, people at CAPITA have recognized many potential uses for cooperating modules/applications not only in data processing, but in data browsing (cursor sharing) as well. Because interprocess communication (IPC) can be separated from computation, it was decided to design software object that would provide high level interface to IPC.

Goal: Our goals were to facilitate communication and synchronization between distributed modules. We wanted to provide a high level interface to interprocess communication, and to allow programmers to change communication and synchronization properties at runtime.

To achieve these goals, we implemented Synchronized Data Objects (SDOs), objects that encapsulate interprocess communication and synchronization details of data streams, on top of the I/O abstraction model. The *I/O abstraction* model separates computation from communication. SDOs are implemented in the Programmers' Playground, a development software for distributed applications that supports I/O abstraction programming model.

This paper is organized as follows. In section 2, I/O abstraction, the underlying programming model, is described. Related synchronization protocols and primitives are examined. Section 4 defines functionality and implementation of SDOs. Examples of using SDOs are given in section 5. Section 7 concludes the paper by revisiting SDO features and by outlining the future work.

2. I/O Abstraction in the Programmers' Playground

The goal of the Programmers' Playground is to facilitate the construction and smooth integration of diverse distributed multimedia applications. The system is based on a programming model called I/O abstraction, a connection-oriented model of interprocess communication in which independent modules interact with an abstract environment. This section provides a brief overview of I/O abstraction. For the purposes of this paper, I/O abstraction is taken as a "given." Information on related programming models and details about the Programmers' Playground goals, design and implementation may be found elsewhere [Goldman 94, web-pages].

I/O abstraction is a model of interprocess communication in which each *module* in a system has a *presentation* that consists of data structures that may be externally observed and/or manipulated. An *application* consists of a collection of independent modules and a *configuration* of *logical connections* among the data structures in the module presentations. Whenever published data structures are updated, communication occurs implicitly according to the logical connections.

I/O abstraction communication is *declarative*, rather than *imperative*. One declares direct high-level logical connections among the state components of individual modules, as opposed to directing communication within the control flow of the module. Once the high-level relationships between state components are declared, if a particular state change in one module should be reflected in the state of another module, then this can be recognized by the system and the necessary communication can be handled implicitly.

This declarative approach simplifies application programming by cleanly separating computation from communication. Software modules written using I/O abstraction do not make explicit requests to establish or effect communication, but instead are concerned only with the details of the local computation. Communication is declared separately as high-level relationships among the state components of different modules.

With implicit communication, it is the configuration (and not directives from within the program) that determine whether or not communication will take place. This means that choices can be made at configuration time about whether modules will communicate and about how they will communicate (point-to-point, multicast, etc.) without modification of the modules themselves.

I/O abstraction is based on three fundamental concepts: data, control, and connections, presented here in terms of The Programmers' Playground implementation.

Data (the components of a module's state) may be kept private or they may be *published* so that other modules may access the data. Each Playground module has a *presentation* that consists of the data that it has published. The presentation may change dynamically. A Playground module interacts with an *environment*, a collection of other modules that may be unknown to this module but that read and modify the data items in its presentation (as permitted by the access privileges).

The *control* portion of a module defines how its state changes over time and in response to its environment. Insulated from the structure of its environment, a Playground module interacts entirely through the local data structures published in its presentation. A module may autonomously modify its local state, and it may react to “miraculous” changes in its local state caused by the environment. This suggests a natural division of the control into two parts: *active control* and *reactive control*. Playground modules may have a mixture of both active and reactive control.

The active control carries out the ongoing computation of the module. For example, in a discrete event simulation, the active control would be the iterative computation that simulates each event. External updates of simulation parameters could affect the course of future iterations, but would not require any special activity at the time of each change. Modules with only active control can be quite elegant, since input simply steers the active computation without requiring a direct response.

The reactive control carries out activities in response to input from the environment. A module with primarily reactive control simply reacts to each input from the environment, updating its local state and presentation as dictated by that input change.

The active control component of a Playground module is defined by the “mainline” portion of the module. Reactive control is specified by associating a *reaction function* with a presentation data item.

Relationships between data items in the presentations of different modules are declared with *logical connections* between those data items. These connections define the communication pattern of the system. Connections are established by a special Playground module, called the *connection manager*, that enforces type compatibility across connections and guards against access protection violations by establishing only authorized connections.

Connections are declared separately from modules so that one can design each module with a local orientation and later connect them together in various ways.

Playground supports two kinds of connections, *simple connections* and *element-to-aggregate* connections, but only simple connections are considered in this paper.

A simple connection relates two data items of the same type, and may be either *unidirectional* or *bidirectional*. The semantics of a unidirectional connection from integer x in module A to integer y in module B is that whenever A updates the value of x, item y in module B is correspondingly updated. If the connection is bidirectional, then an update of y 's value by module B would also result in a corresponding update to x in A.

3. Related Work

In this section, we review existing synchronization primitives: RPC, rendezvous, futures, and promises. Protocols implemented in RPC, futures and promises are also implemented in SDO. The rendezvous is described because SDOs communicate in a similar fashion.

3.1 RPC

Remote procedure calls are an extension of procedure calls to the network environment [Birell & Nelson 84]. As in the case of a regular procedure call, invoking an RPC causes suspension of the calling program. Arguments are encoded and passed across the network to the entity that contains the desired procedure, and that procedure is activated. When the procedure is done, the results are passed back to the calling program, which is then reactivated.

An RPC involves five pieces of software: the user program, the user-stub, the RPC communication package, the server-stub, and the server program. The relationships among these components are illustrated in **Figure 1**. The user treats remote calls in the same way as regular procedure calls. When a remote procedure is invoked from the user program, a corresponding procedure in the user-stub is invoked. The user stub creates packets out of the called machine specifications, procedure name, and procedure arguments, and forwards these packets to the RPC communication package. The RPC communication package is responsible for transmitting and receiving packets across the network. When the RPC communication package receives the packets at the called machine, it gives the packets to the server-stub. The server-stub extracts the procedure name and the arguments from the packets and makes a regular procedure call to the server program. When the procedure is done, the server-stub creates packets with the results, and sends them back to the caller machine. At the caller machine, the user-stub takes packets from the RPC communication package, and returns the control of the execution to the user program together with the results of the remote procedure call.

Remote procedure calls are common method of communication in distributed programming because of its easy-to-understand procedure-style semantics. However, use of remote calls results in lower performance than explicit message exchange because the calling program must wait for the reply before continuing.

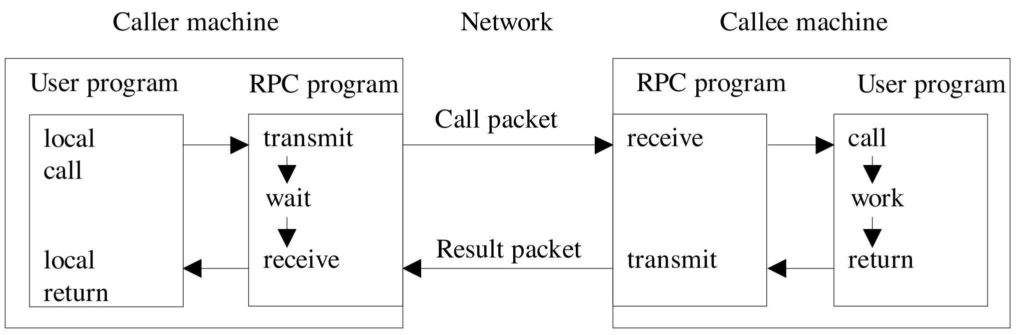


Figure 1. Implementation of a simple *RPC* call

3.2 Ada's Rendezvous

The Ada rendezvous a synchronization primitive similar to RPC [Perrott 87][Ben-Ari 82]. One task explicitly communicates to another task by calling a procedure defined in that task. The callee task accepts the call by executing an *accept* statement. As in RPC, data is passed in form of procedure arguments. The task that reaches its communicating statement first is blocked. The rendezvous occurs when both tasks are ready to communicate, i.e. when the caller task invokes remote procedure and when the callee task reaches an *accept* statement. The caller task is delayed until the *accept* statement, which constitutes the critical section, is executed. The two tasks then proceed independently. The synchronization is thus implicit. The synchronization is also asymmetric: the caller task has to name the callee task but not vice versa.

3.3 Futures

Multilisp is an extended version of Lisp with constructs for parallel execution [Halstead 95]. The *future* is the main construct for creating and synchronizing tasks. The call (future X) returns a future which is a place holder for the future value of X. When the value of X is calculated, that value replaces the future. The value of the future is undetermined, and an attempt to access a future's value causes suspension of the caller task until the value is determined. If the future is not accessed immediately, significant concurrency can be achieved between the calling and called program.

In Multilisp, futures are implemented as objects with following properties: a Lisp value, a task queue, a *determined* flag, and a lock. When the value of an operand needs to be examined, Multilisp first checks if the operand is a future. If it is, then it checks if its value is determined. If it is not then the calling task is added to the queue of suspended tasks waiting for the value of this future. All of the suspended tasks are activated when the value is obtained.

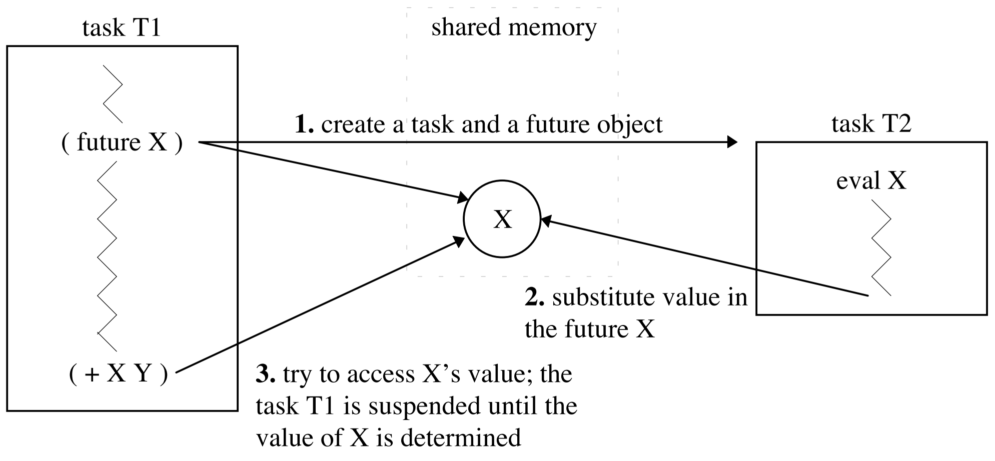

task T1
shared memory
(future X)
1. create a task and a future object
task T2
eval X
X
2. substitute value in the future X
(+ X Y)
3. try to access X's value; the task T1 is suspended until the value of X is determined

Figure 2. Implementation of *futures* in Multilisp.

3.4 Promises

Promise is a data type similar to the *future* [Liskov 88]. It also allows that evaluation of a variable is done in parallel with the caller program. While *futures* were designed and implemented for single multiprocessor machines, *promises* can be used in a program distributed over a network. The distributed program is made up of active entities distributed over a network. Each entity (a *guardian* in Argus) has ports (*handlers* in Argus). Ports are like procedures that can be called on the particular entity. A promise is defined for each port.

Promises protocol and implementation is very much similar to the implementation of futures illustrated in Figure 2, except that promises are used over network instead of a single machine. A promise type is defined as object that contains a results part, which is the list of objects returned by the handler call, and an exceptions part, which lists the exceptions raised by the handler in the case of an error. Operations available on a promise object are *ready* and *claim*. *Ready* returns the state of the promise object (either blocked or ready). The *claim* method waits until the promise is ready. It returns the result, or raises an exception in the case that something went wrong.

4. Synchronized Data Objects

4.1 Functionality

SDOs encapsulate synchronization protocols for exchanging data between modules in a distributed application. The communication takes place between two SDOs, a sender and receiver, contained in two different modules distributed over a network. Type of protocol that is used to exchange data and the data type are defined at the SDO instatiation time. All of the protocol details are completely hidden from the programmer. The advantage of having encapsulated communication protocols is that correctly implemented protocols can be easily reused. We think that this abstraction will simplify the development of distributed applications. Communication protocols can be modified and optimized without rewriting the modules' code. Communication protocols fall in two categories: data driven and demand driven. They are discussed in detail in the following section.

An SDO has a simple interface. Programmers use SDO *Set* method with a sender SDO to send data, and *Get* method with a receiver SDO to receive data. The programmer can also specify callback functions that are invoked when the state of a SDO changes. For example, when a sender SDO receives an request for data, the module that contains the sender object can react by producing new data. Some SDO objects provide properties that allow programmer to optimize the communication protocol.

4.2 Communication Protocol Types

Depending on the cause of the communication, the communication protocols can be divided into two basic groups, data driven and request driven. Data driven communication is initiated by a sender, and is caused by assigning a new value to the sender. Request driven communication is instigated by a receiver when a program attempts to read the data from the receiver.

4.2.1 Data Driven Communication

In data driven communication, the SDO sender controls the data exchange. The cause of communication is assignment of a new data value to the sender object. Two protocols are implemented: supply driven data streams and remote invalidation.

Supply Driven Data Streams (SDDS)

This type of interprocess communication and synchronization is similar to an unbuffered pipe. After the new data is assigned to the sender object, it is sent to the receiver as soon as it is ready. In the case that the sender module can produce data items more frequently than the receiver module can consume them, the sender will block until the receiver is ready.

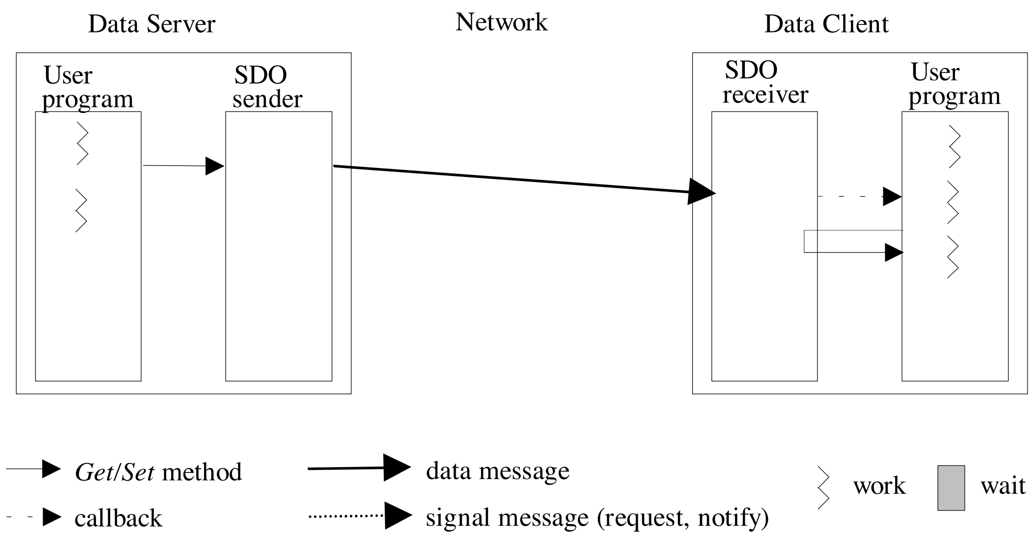


Figure 3. Diagram of the supply driven data stream communication.

When the receiver gets the new data, it invokes a callback function. The callback function is defined by programmer and is used to allow the receiver module to react to the arrival of new data. From the time the new data is received until the receiver module reads that data, the receiver is in *busy* state, unable to receive more data. When the data is read, the receiver sends back a *ready* signal notifying the sender that it is ready for new data.

This protocol is suitable in applications such as data (image) processing when each data item has to be seen by the receiver. Latency is in the worst case equal to the time needed for the ready signal to travel from receiver to sender plus the time needed for the data to travel from the sender to receiver.

Remote Invalidation

The remote invalidation protocol is a version of data driven communication in which the sender object notifies the receiver when data is updated. The receiving module polls the data whenever it finds it appropriate. This mode of communication allows the sender to update the data frequently without using communication bandwidth, unless the receiver requests the data.

When new data is assigned to it, the sender sends notification to the receiver. When the receiver object sees a notification it invokes the callback function that handles notifications. When the receiver module tries to read the data, the receiver object sends a request for data to the sender and waits for data to arrive. When the data arrives, it returns it to the receiver module.

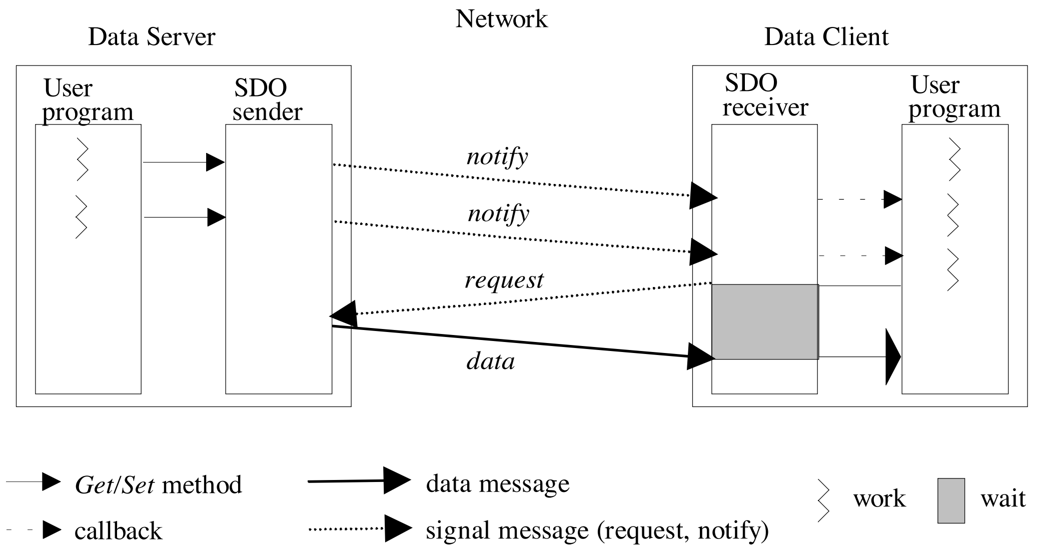


Figure 4. Diagram of the data driven protocol with notification.

The remote invalidation protocol is declared by setting the data driven sender object's *Notify* property to *true*. The sender and receiver objects are used in the same way as with the pipe protocol, except that there is a callback function invoked by the receiver when the notification arrives.

This type of communication can be utilized in cases when large data items are frequently updated but the receiver does not need to see each data item. This type of communication combines advantages of both data driven and request driven communication, because the receiver module always knows when the new data is ready and it has the ability to request it when it wants to. In this way the network traffic can be reduced by not sending every data item. However, the reduction in the communication overhead is countered by increase in latency. Latency can be as much as three times larger than in the ordinary data driven communication, because the receiver might have to wait while notify, request, and data message all make one trip between the sender and receiver.

4.2.2 Request Driven Communication

The request driven communication is initiated by the receiver module at the moment when the program tries to read data from the receiver SDO. Two types of data driven protocols are implemented: demand driven data streams and promises.

Demand Driven Data Streams (DDDS)

The DDDS protocol is semantically similar to the remote procedure call. An attempt to read data from the receiver object in DDDS is equivalent to invoking a remote procedure. When a program tries to read data from the receiver object, the receiver object sends a request to the sender object and blocks until data arrives. When the sender SDO receives the request, it first invokes the callback procedure that handles incoming requests. Programmers can use this callback function to initiate computation of a new value and to assign it to the sender object. After invoking the request callback function, the sender object returns supplied data. Upon receiving data, the receiver object returns data and the execution control to the calling module.

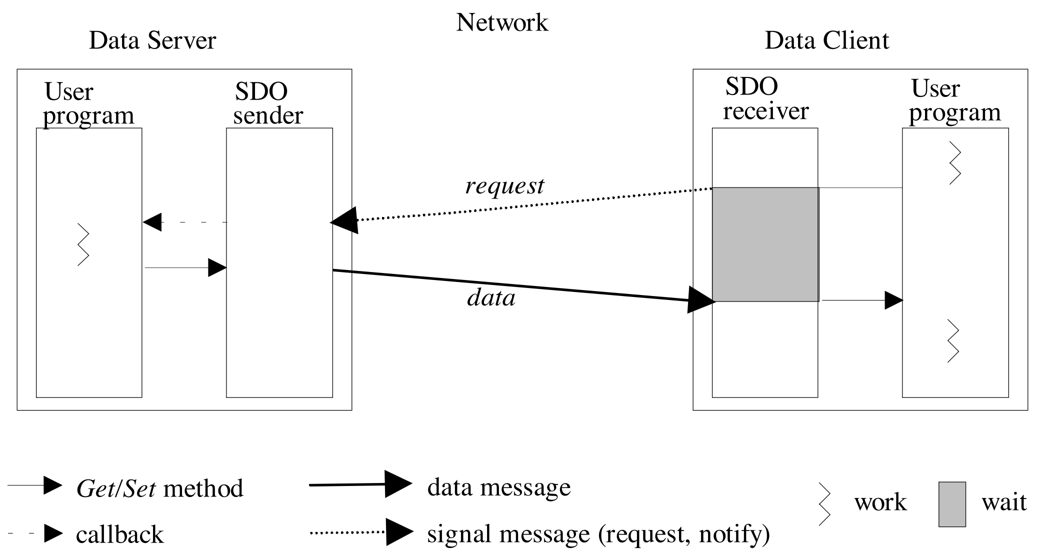


Figure 5. Diagram of the DDDS communication protocol.

In the DDDS communication protocol, the interface between SDO sender and receiver and their modules is rather simple. Data is read from the receiver module using *Get* method. On the sender end, data is assigned to the sender object using *Set* method. The programmer also needs to supply a callback function that handles incoming requests and modifies data stored in the sender. This function may, for example, calculate new data and assign it to the sender object.

The DDDS protocol is suitable for demand driven data processing distributed applications. As opposed to supply driven data processing which is usually close to the batch job, the demand driven data processing can be non-deterministic. For example, it can be driven by a user at the runtime. The DDDS communication could be extended to support efficient deterministic demand driven data processing by allowing the receiver SDO to *prefetch* data. In this mode, the receiver SDO would send a request for next data item, immediately after the current data item is read. Without prefetching, latency is always twice the time a message needs to get from the sender to receiver plus time the sender module needs to compute the new data item. With prefetching, latency can be reduced, if not completely masked, because the request is sent before the receiver tries to read the data.

Promise

The promise communication protocol is a version of the DDDS protocol that allows more concurrency between sender and receiver modules. After the SDO receiver issues the request signal, it immediately returns the control to the caller module. This enables the module to do other tasks in parallel with the computation of data. The caller module tries to read data at a later point. In the case that data has not arrived, the receiver object blocks, otherwise data is immediately returned.

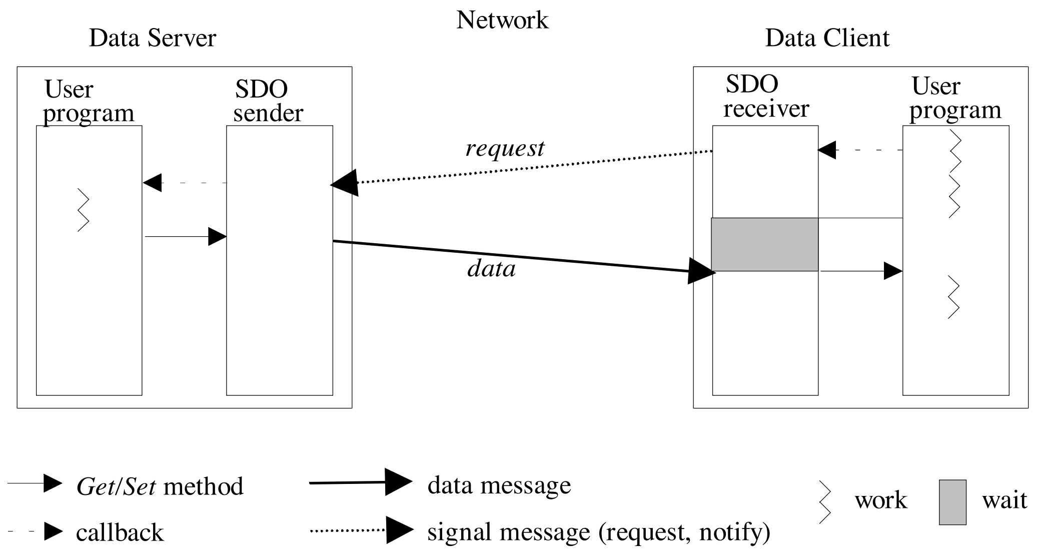


Figure 6. Diagram of the promise communication protocol.

The interface for the promise protocol is same as the interface for the DDDS protocol, with one additional method, *Request*. The *Request* method causes the SDO receiver to request data from the sender without waiting for the data to come back. The control of execution is immediately returned to the calling module so that it can perform other tasks while new data is being computed at the sender. The *Request* method is invoked prior to *Get* method. Setting the *Promise* property to true changes the current protocol from DDDS to Promise.

In the best case, use of the promise protocol can achieve significant concurrency and completely mask network latency. For example, a client module can issue requests for new data immediately upon receiving data. By doing so, client can process the data it received and server can compute new data at the same time. And if the server produces new data sufficiently faster than the client can process it, the client does not have to block waiting for new data. However, if the caller program tries to read data immediately after issuing a request, promises have same latency as DDDS.

4.3 Implementation

Synchronous Data Objects are implemented in C++ on top of the Programmers' Playground. The Programmer's Playground provides applications written in C++ with asynchronous communication implemented on top of the SunOS UNIX operating system with sockets as underlying communication mechanism. The Programmer's Playground also takes care of the connection management. The connection manager provides graphical user interface for configuring connections between modules. For each connection request, it checks for type compatibility and access protections. The connection manager is not a communication bottleneck since it simply sets up connections that are thereafter handled individually by the Programmers' Playground endpoint protocols [Goldman, Anderson, and Swaminathan 93].

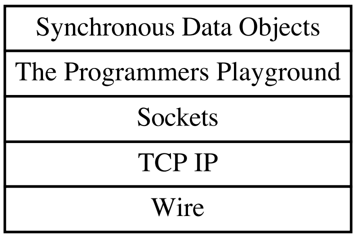


Figure 7. Communication layers under SDOs.

The Programmers' Playground specifics, such as publishing message data structure and ways of sending and receiving data, are encapsulated in the class **CAsynchComm**. The advantage of having underlying asynchronous communication mechanisms encapsulated in one class is that it simplifies implementing SDOs across different platforms.

```
class CAsynchComm
    void SendData(typeData*)
    void SendRequest(typeRequest*)
    void SendNotify(typeNotify*)
    void SendNonBusy(typeNonBusy*)
    void SetReactOnNewData(typeFunc*)
    void SetReactOnRequest(typeFunc*)
    void SetReactOnNotify(typeFunc*)
    void SetReactOnNonBusy(typeFunc*)
    void Sleep(typeSecs*)
```

Figure 8. CAsynchComm class interface.

All SDO classes contain an instance of **CAsynchComm** class which they use for accessing underlying asynchronous communication mechanism. The figure above lists public methods of the **CAsynchComm** class. Methods that start with ‘Send’ are used to send messages between sender and receiver object. Messages are sent by value. Methods starting with ‘SetReact’ register reaction functions with different messages. Reaction functions are invoked when a new message of the corresponding type arrives. Finally, the *Sleep* method is used to block a module for the given time period.

Classes **CDDSender** and **CDDReceiver** implement the **D**ata **D**riven communication protocols, and **CRDSender** and **CRDReceiver** implement **R**equest **D**riven communication protocols. These classes are instantiable by application programmers. Following figure outlines the public interface of these classes.

```
class CDDSender                          class CRDSender
    bool Notify                              void Set(typeData*)
    void Set(typeData*)                      void SetReactRequest(typeFunc*)

class CDDReceiver                        class CRDReceiver
    void Get(typeData*)                      bool Promise
    void SetReactNewData(typeFunc*)          void Request(typeRequest*)
    void SetReactNotify(typeFunc*)           void Get(typeData*)
```

Figure 9. SDO class interfaces.

4.4 Summary

The following table describes how configuring SDO senders and receivers in different ways can produce communication protocols that were discussed in this section.

		SENDER	
	properties	send on update	send on request
RECEIVER	request before read		Promise
	request on read	SDDS with notify	DDDS
	no request	SDDS	

Figure 10. Communication properties.

The sender SDO reacts to two types of events: update of data, and arrival of a request. Its behavior can be described in terms of how it reacts to these events. It either sends data immediately upon update, or it does not send anything until a request arrives. The receiver SDO reacts to when a program attempts to read its data. Its behavior is determined by when it sends requests for data in relation to attempted reads. It can wait for data without requesting it, or it can request it prior to a read, or after it.

By combining differently configured senders and receivers, different communication protocols can be implemented. A sender that sends on update connected to a receiver that does not send requests results in the SDDS protocol. Replacing the receiver that does not send requests with the one that requests data when someone tries to read it, gives us the SDDS with notify protocol. For the request driven protocols, we need a sender that send data only on requests. Combining it with a receiver that eagerly requests data before a read attempt produces the Promise protocol. If the receiver is lazy in the way it requests data, and it requests data only after a read attempt, then we have the DDDS communication.

5. Using SDOs in CAPITA

In this section, I give examples of how can SDOs and different modes of communication be applied in the CAPITA environment.

Research activities in the CAPITA department heavily depend on the information obtained by processing raw data. Several stand-alone applications have been developed to handle different types of data manipulation. For example, *Voyager* is a server of tabular data, *Contourer* converts tabular data into the grid format, *MapEditor* displays data and manipulates its graphic properties, and so on. Because processing data from its original state to the state in which it becomes more meaningful involves passing it through several of the data processing applications, there is a need for collaboration and data exchange between these applications.

The number of data processing applications is increasing as our demands change. Consequently, the number of ways applications can be interconnected increases as well. SDOs should be a tool that will facilitate creation of distributed applications.

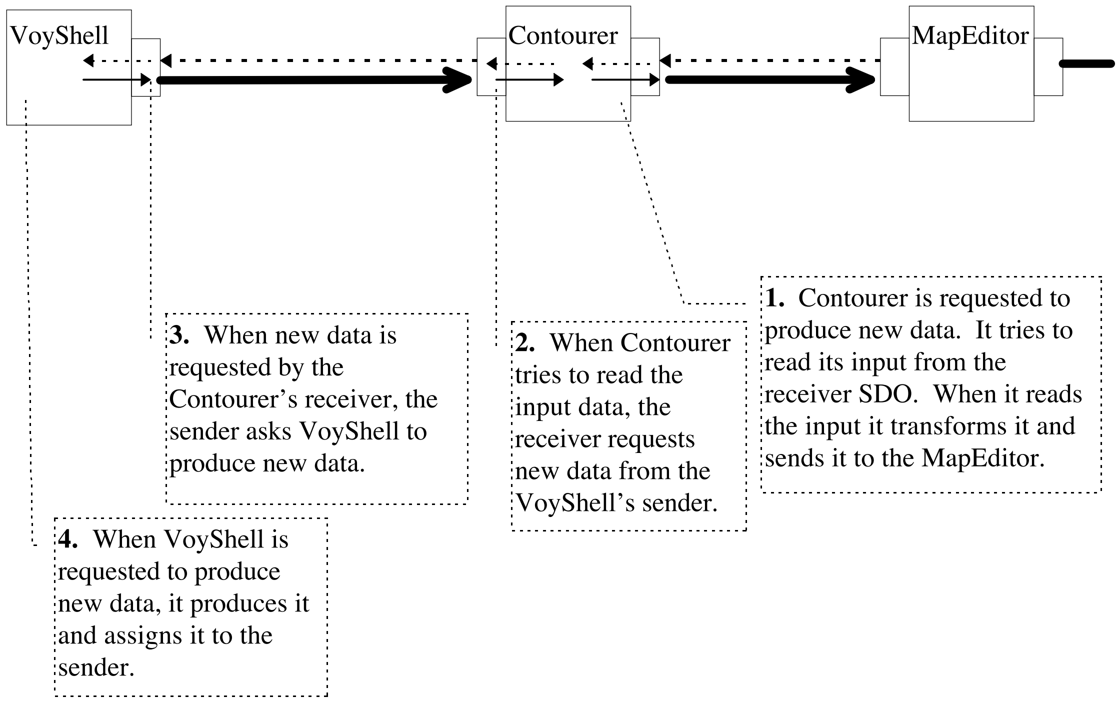
I present here two examples using CAPITA applications. The first example describes distributed data processing pipeline, and the second one describes a distributed data browsing application.

Following notation is used for data and control (request, notify) messages

$\longrightarrow$ data flow $\qquad$ $\dashrightarrow$ control message flow

5.1 Demand Driven Pipeline

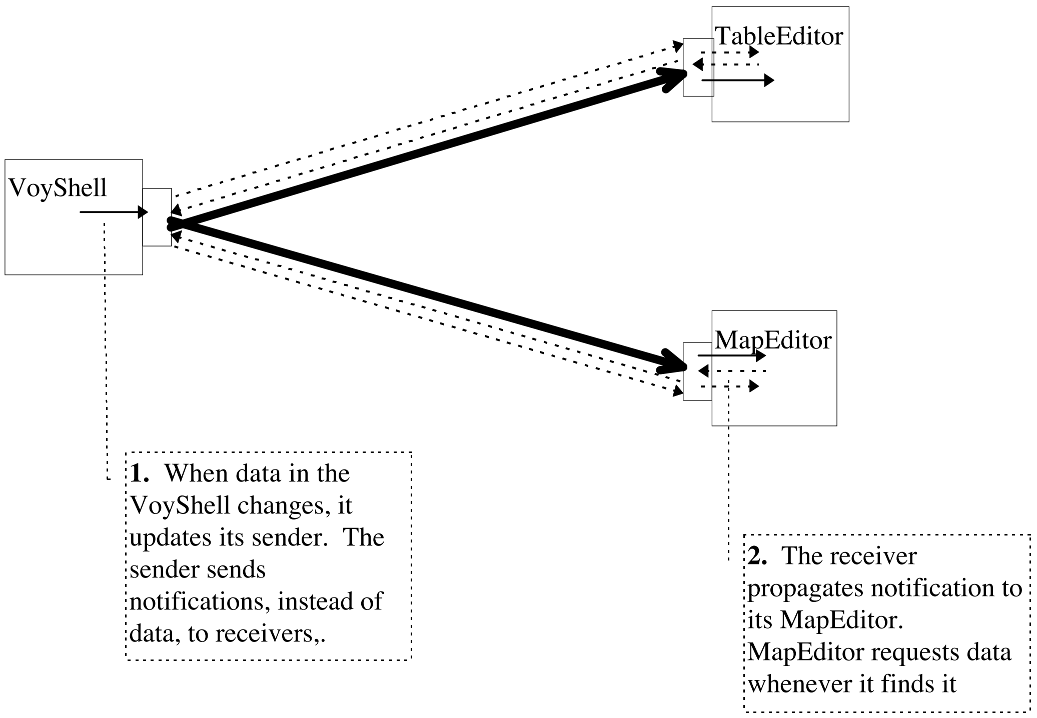
In a demand driven distributed pipeline, data is produced as a result of a user action. Since user actions are nondeterministic, the DDDS communication protocol seems to be the most natural way of data exchange. The configuration below depicts a pipeline that produces contoured data out of data tables and displays it in *MapEditor*.


VoyShell
Contourer
MapEditor
1. Contourer is requested to produce new data. It tries to read its input from the receiver SDO. When it reads the input it transforms it and sends it to the MapEditor.
2. When Contourer tries to read the input data, the receiver requests new data from the VoyShell's sender.
3. When new data is requested by the Contourer's receiver, the sender asks VoyShell to produce new data.
4. When VoyShell is requested to produce new data, it produces it and assigns it to the sender.

The demand driven data stream protocol allows that the data processing is driven by user requests. It also ensures that each data item is seen exactly once by the displaying application.

5.2 Shared Data Viewers

In the following configuration, *Voyager* is the data server and *TableEditor* and *MapEditor* are data views. Changing data in *Voyager* causes update, or notify, messages to be sent to the views. The views can later request the data when they become active or when user explicitly wants an update.


TableEditor
VoyShell
MapEditor
1. When data in the VoyShell changes, it updates its sender. The sender sends notifications, instead of data, to receivers,.
2. The receiver propagates notification to its MapEditor. MapEditor requests data whenever it finds it

The SDDS with notify suits well this kind of application. It saves communication traffic at times when the data views do not need to see each data item. The another advantage of this type of communication is that if the receiver knows that there is new data at the sender, it does not have to send an unnecessary request, instead it can simply return the data it received previously.

6. Conclusions and Future Work

In this paper we have presented an object oriented approach to interprocess communication abstraction in the context of the I/O abstraction, a connection-oriented model of interprocess communication seeking to separate communication from computation. Synchronized Data Objects are synchronization primitives that encapsulate details of communication protocols, providing a high level interface for communication properties. Using the interface, communication protocols can be changed without modifying the body of the program. Resulting separation of communication from computation promises reuse of protocols and easier development of distributed applications.

SDOs are designed to pairwise communication: communication takes place between a sender SDO and a receiver SDO. Two types of protocols are encapsulated in SDOs: data driven and demand driven. The type of protocols to be used is defined at the instantiation time. Each communication type can be parameterized at the runtime using object properties. Currently there are four different communication protocols implemented: supply driven data streams, supply driven data streams with notification (or remote invalidation), demand driven data streams, and promises.

A test implementation of SDOs is done in C++ using the Programmer's Playground for underlying asynchronous communication. We are looking into implementing SDOs in the Windows^{TM} environment, specifically as Visual Basic objects. Further understanding of communication properties between modules in a distributed application will be pursued, as well as the configuration patterns in distributed applications.

Acknowledgments

I want to thank Dr. Ken Goldman for discussing distributed programming concepts and synchronization primitives, and for providing help and guidance throughout the research.

Thanks to Dr. Rudolf Husar for discussions about distributed processing that led to the idea of abstract communication objects.

Bibliography

Birell A.D., and Nelson B.J. **Implementing Remote Procedure Calls**. *ACM Transactions on Computer Systems*, pages 39 - 59, 1984.

Goldman K.J., Anderson M.D., and Swaminathan B. **The Programmers' Playground: I/O Abstraction for Heterogeneous Distributed Systems**. *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 363 - 372, 1994.

Halstead R.H. **Multilisp -- A Language for Concurrent Symbolic Computation**. *ACM Transactions on Programming Languages and Systems*, pages 501 - 538, 1985.

Liskov B., and Shrira L. **Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems**. *Massachusetts Institute of Technology, Programming Methodology Group Memo 60-1*, 1988.

Aksit M., Wakita K., Bosch J., Bergmans L., Yonozawa A. **Abstracting Object Interactions Using Composition Filters**. In Guerraoui R., Nierstrasz O., Riveill M., eds. *Object--Based Distributed Programming*. *Proceedings of ECOOP '93 Workshop, Kaiserlautern, Germany*, 1993.

Perrott R. H. **Parallel Programming**. *Addison-Wesley Publishing Company*, 1987.

Ben-Ari M. **Principles of Concurrent Programming**. *Prentice Hall International*, 1982.

Goldman J. K. et al. URL *http://www.cs.wustl.edu/cs/playground*