# An Advanced Communication Toolkit for Implementing the Broker Pattern*

Cristian Francu and Ivan Marsic
*Center for Computer Aids for Industrial Productivity (CAIP)*
*Rutgers University, 96 Frelinghuysen Rd, Piscataway, NJ 08854-8088*
{*francu,marsic*}@*caip.rutgers.edu*

## Abstract

*The Broker pattern is a powerful solution when building middleware communication systems. Existing toolkits, such as BAST, GTS, and ACE, although useful, are insufficient to implement the Broker pattern architecture. These systems concentrate on wrappers for communication protocols, and on implementing auxiliary communication patterns, but address only some aspects of object communication. In this work we demonstrate how the Broker pattern can be easily implemented by using an Advanced Communication Toolkit (ACT). ACT model defines four layers according to the increasing degree of abstraction of exchanged information. The resulting systems are highly customizable, extensible, portable, and can communicate at any of the four layers independently. ACT supports various high-level communication protocols (e.g., HTTP, IIOP, SMTP) and can be used to implement Broker-based systems such as OMG CORBA, Java RMI, or Microsoft DCOM.*

**Keywords**
framework, design patterns, distributed system, broker pattern, reusable code, finite state machine

## 1. Introduction

Design patterns have recently become a strong trend in the effort to cope with software complexity [5]. Patterns are means for documenting recurring micro-architectures, which are abstractions of common object structures that expert developers apply to solve software design problems. One of the major distributed system patterns is *Broker Pattern* [3]. Broker pattern is an architectural pattern used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication. Figure 1 shows the objects constituting the Broker pattern and their interactions.

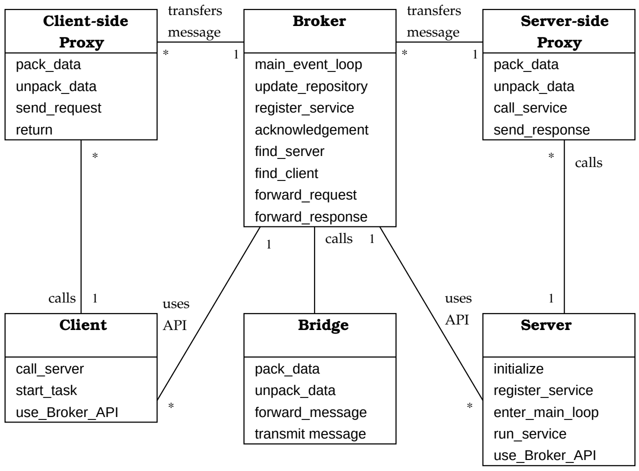


**Figure 1. Broker pattern components (Adapted from [3]).**

The Broker pattern has been proven effective in implementing distributed software systems, since it leads to greater flexibility, maintainability, and changeability of the resulting system. By partitioning functionality into independent components the system becomes potentially distributable and scalable. Objects are accessed through interfaces, defined in Interface Definition Languages (IDLs). This approach provides for programming language independence since an IDL mapping can be defined for any programming language. Broker pattern definition [3] describes its functionality and collaborations with other patterns, but does not give design or implementation guidelines. CORBA, RMI, DCOM and other state of the art

*The research reported here is supported by the DARPA Contract No. N66001-96-C-8510 and by the Rutgers Center for Computer Aids for Industrial Productivity (CAIP). CAIP is supported by the Center's Corporate Members and by the New Jersey Commission on Science and Technology.

object-oriented technologies for interconnecting objects on heterogeneous systems are examples of elaborating and implementing the Broker pattern.

This paper presents an Advanced Communication Toolkit (ACT) for implementing the Broker pattern in a structured manner. The theoretical and practical results provide for a rapid development of a large class of Broker-based communication systems, thus allowing construction of in-house customized Brokers. For instance, an ACT architecture for implementing the Broker pattern allows the replacement of primitive communication mechanisms, protocols, request demultiplexing policies, and data marshaling without interference with the rest of the system. The developer is not necessarily restricted to one standard, such as CORBA. Simplified versions of the Broker pattern can be implemented using ACT, such as Client-Dispatcher-Server, or related patterns, e.g., Forwarder-Receiver [3].

The rest of the paper is organized as follows. Section 2 describes the ACT architecture, interfaces and functionality. Section 3 shows how ACT can be extended with new protocols and multi-threading policies. Section 4 describes the two systems that were built using ACT: ACT-based CORBA Object Request Broker (ORB) and the Collaboration Bus (cBus) which is a part of a synchronous groupware system. Based on experience with ACT, Section 5 summarizes its advantages and discusses performance issues. Section 6 compares our approach with approaches taken in other projects. Finally, Section 7 summarizes and concludes the paper.

## 2. ACT Architecture

There are standard issues with which all the object communication systems have to deal. Examples are data marshaling and message passing in compliance with an application-level protocol (such as HTTP, or IIOP). CORBA, for instance, enforces the use of Common Data Representation (CDR) for the external data format, and Internet Inter-ORB Protocol (IIOP) in order to achieve interoperability between different vendor implementations.

Our model identifies the functionality an object communication system has to address and groups it in four layers according to the degree of abstraction of exchanged information (Figure 2). At the lowest level (*Endpoint*) the objects exchange data as streams of bytes using the OS-provided inter-process communication mechanisms (e.g. TCP or shared memory). At a higher level (*Protocol*), the objects communicate via messages defined by an application-level protocol. At this level, a message still carries an unstructured array of bytes. At both of these layers a message is delivered to a communication port rather than to the target object. The third layer (*Client/Server*) introduces the notion of target objects. It also handles the case when multiple messages are destined to the same target object via multi-threading. The topmost layer (*Presentation*) increases abstraction by allowing objects to exchange structured messages via method calls. At this level there is no difference in communicating to remote or local objects.

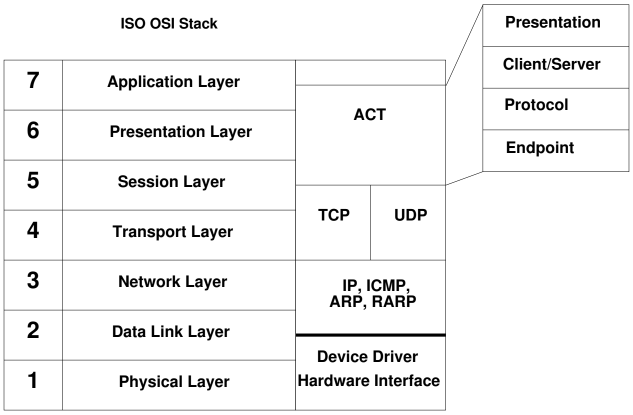


**Figure 2. ACT covers the same region of ISO OSI stack as CORBA or RMI. Four layers are defined within this area, endpoint, protocol, client/server and presentation.**

Even though there are other toolkits that to some degree abridge the gap between the objects and the transport protocol [1, 6, 11, 13], they usually don't provide an abstract model covering the entire distance between the OS network interfaces and the object oriented remote method calls. Some also have layered design, and common interfaces but only for (unstructured) message passing and transport protocol invocation. ACT extends this model to application level protocols, multi-threading policies, service management and data marshaling.

The model satisfies the following requirements:

**Common interfaces** for different protocols and inter-process communication (IPC) mechanisms. Patterns help inexperienced developers to build a better application, by following a set of rules proven beneficial by the experts. Analogously, common interfaces help by a small number of classes/ methods through which various functionality can be accessed. In case of layered systems, common interfaces allow for replacing modules even at run-time, and yield in better reuse and customizability by using only the necessary functionality in the bottom layers.

**Gradual increase in abstraction** from the ISO OSI transport layer to the application layer. Every layer has an interface to the layers above, and calls only the layers beneath. This allows communication at various levels of abstraction.

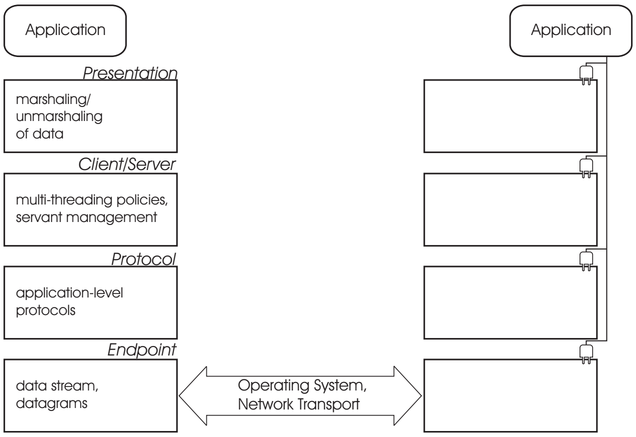


**Figure 3. The Advanced Communication Toolkit architecture. Each layer adds new functionality to the previous ones. The application programmer has access to any of the four layers as shown symbolically by plugging sockets on the right. When accessing one layer the application makes use of the functionality of all the layers beneath.**

One or more upper layers can be omitted to achieve better performance (Figure 3). The full Broker implementation needs to account for general object communication. Unlike this, a specific well-defined context may not require handling all cases. Communication at a lower level of abstraction yields in better usage of resources, in particular in better performance. For instance, the distributed application may gain in performance if it communicates using the CORBA IIOP protocol, but without object proxies and an object adapter.

ACT has the following layer-based vertical structure (Figure 3):

1. **The endpoint layer** delivers to and retrieves from the transport mechanism a byte stream or a packet; it deals with the specifics of the operating system.

2. **The protocol layer** provides high-level communication protocols, such as HTTP, IIOP, FTP etc., that use either TCP stream or UDP datagrams.

3. **The client/server layer** provides common services for servers and clients: main loop, service registration, and concurrency handling.

4. **The presentation layer** converts data from system-dependent format to standard data representation (e.g., CDR, XDR).

## 2.1. Endpoint Layer

The endpoint layer acts as a common API to various OS-level network interfaces. Internals of this layer are highly dependent on the operating system and low-level IPC mechanisms. This layer is needed in order to achieve platform independence at the layers above. The wrappers for these interfaces are common in many toolkits, e.g., [1, 6, 11, 13]. This is the only platform-dependent layer in our model.

## 2.2. Protocol Layer

While the endpoint layer provides wrappers for transport protocols, the protocol layer provides a common API for communication using application-level protocols, such as HTTP, or IIOP. The protocol is platform independent since layer uses the endpoint layer. It is desirable to design a flexible class architecture such that the system can be extended to implement a variety of protocols. Implementing and plugging in new protocols should be as easy as possible.

The protocol layer has two major roles. One is to provide an object-oriented representation of the protocol messages. The second is to enforce the semantics of the protocol. For example, when using IIOP, the client can not send messages of type `Reply`. Any request from the application programmer to send such an illegal message should generate an error.

### 2.2.1. Client-Side Interface

The client-side interface is shown in Figure 4. It provides a single well known object, *ClientSender*, through which all requests are sent, and any number of *ClientReceiver* objects through which replies are received for each request. A sample of code that uses this interface is shown below.

```
// creates a ClientSender object
cs = new ClientSender(...);

// create a message to send
Message m = new ...;
cs.sendMessage(m);
cr = new ClientReceiver(
        m.getTransactionID()
);
Message answer = cr.receiveMessage();
```

The *Message* interface is presented in Section 2.2.3.

### 2.2.2. Server-Side Interface

Handling of sending and receiving messages is done through two objects kept inside *Protocol*: *ServerSender* and *ServerReceiver* (Figure 5). A sample code for a sequential server that uses this interface is:

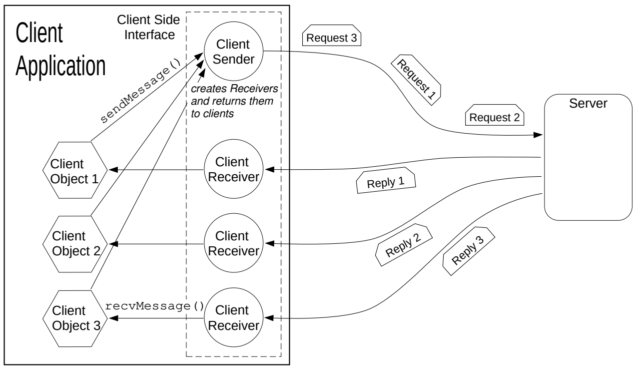


**Figure 4. Client-side interface for the protocol layer.**

```
ProtocolServer ps =
      new ProtocolServer(host, portnumber);
Protocol p = ps.accept();

// receive request
Message request =
       p.getServerReceiver().
           receiveMessage();
      ...   // code to process the request
Message reply = new ... // create the reply
// send message
p.getServerSender().sendMessage(reply);
```

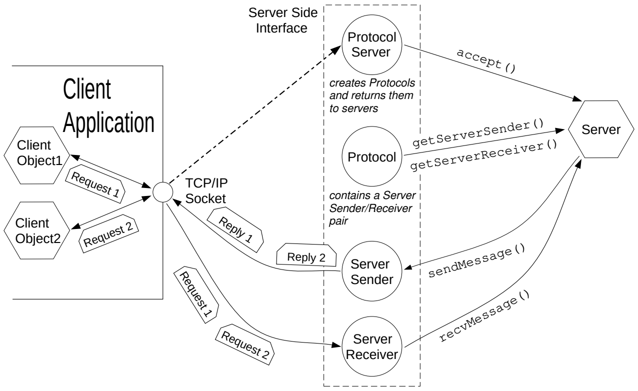


**Figure 5. Server-side interface for the protocol layer. Server represents a functional module that performs services. In our particular case, server symbolizes the entire client/server layer (Section 2.3).**

#### 2.2.3. Message Interface

The message object is a key element of the Protocol layer. A message contains a piece of data, the destination information, and a method for writing it to an output stream according to protocol-specific rules. Message implementations extend the common *Message* interface. They usually correspond to the messages specified by the protocol. For example, ACT implementation of CORBA IIOP contains seven message classes, matching the seven types of messages specified by IIOP [17].

The *Message* interface is designed to hide the protocol-specific details, thus making this layer protocol independent. The IDL description of this interface is:

```
interface Message {
    // method for writing a Message
    void write(in Writer os);

    // the destination of the message
    readonly attribute Endpoint endpoint;

    // message type (e.g., Request)
    readonly attribute int type;

    // coupling between requests and replies
    readonly attribute int transactionID;
}
```

Some protocols allow the client to send multiple requests to the server without necessarily waiting for the replies. Examples of such protocols are HTTP v1.1 and IIOP. In this case, the associated reply messages might come back in a different order from the one in which the requests were sent. Therefore, the identification of the original request has do be done for each reply message. Unlike other toolkits, ACT performs this pairing (Figure 4), relieving the application programmer from this responsibility. The programmer can explicitly demand to receive the reply for a specific request.

## 2.3. Client/Server Layer

The client/server layer resides on top of the protocol layer, and uses its interface to handle request messages. At this level, the programmer defines *services* to be called by clients. A *server* object contains several *service* objects, each providing a different service. The server allocates threads to requests according to the multi-threading policy set by the application programmer (Figure 6). It also registers/unregisters services and dispatches requests to the proper service. This layer provides common interfaces for deciding the multi-threading policy and for service registration.

This layer consists of the thread engine and the object adapter. The thread engine implements multi-threading

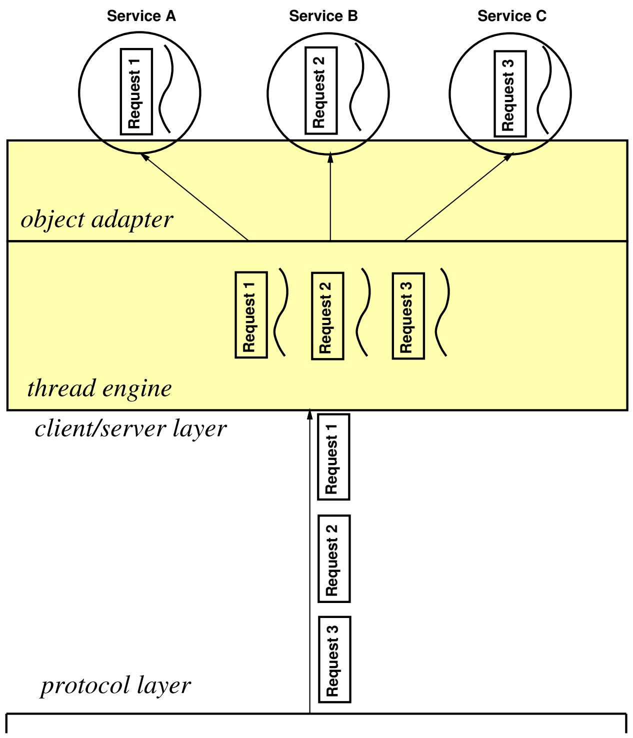


**Figure 6. Client/server layer architecture. Request messages come from the protocol layer. They first get assigned threads according to the multi-threading policy, then they get dispatched to the proper service.**

policies. The ACT core contains the most common policies *thread-per-session*, *thread-per-request*, *pool-of-threads* [14]. New user-defined policies can be added. The thread engine uses protocol layer to receive requests. The object adapter is called by the thread engine. Its role is to activate/deactivate services, dispatch requests to services, and create/destroy service object references.

The role and functionality of client/server layer is similar to the role of the Portable Object Adapter (POA) defined by CORBA 2.$x$ [9]. We could not use POA as-is since it is specific to CORBA, being IIOP aware and dependent. Instead, we defined a set of protocol-independent interfaces that support POA functionality. POA does not define multi-threading policies and delegates this choice to the ORB object. ACT allows to choose a policy at run-time and, unlike ORBs, allows to add new policies when needed (Section 3).

### 2.4. Presentation Layer

Presentation layer is the top-most layer of the ACT architecture and is responsible for marshaling/ unmarshaling of data, creating protocol messages, and extracting the message body. At this level the application programmer uses standard object oriented calls to access remote objects, thus achieving the transparency between local and remote objects. The presentation layer is the only layer that has knowledge about the semantics of the data inside messages. The other layers simply transport data without interpreting it.

Since the remote calls are made the same way as local calls it is not possible to define a common API for the application programmer at this level. Calls will be made using the interface defined by the service provider. However, it is possible to define common interfaces which help the system programmer in designing and implementing various marshaling engines. Presentation layer interfaces for the system programmer are designed to allow different standards of data representation (such as XDR for SUN RPC, or CDR for OMG CORBA [17, 15]).

The presentation layer interface for the system programmer is based on streams. The streams are specialized in reading and writing primitive data types and arrays of primitive data types. Objects that call this layer do not know about the protocol layer messages. All they know is the input/output stream where the unmarshaled data comes from/goes to.

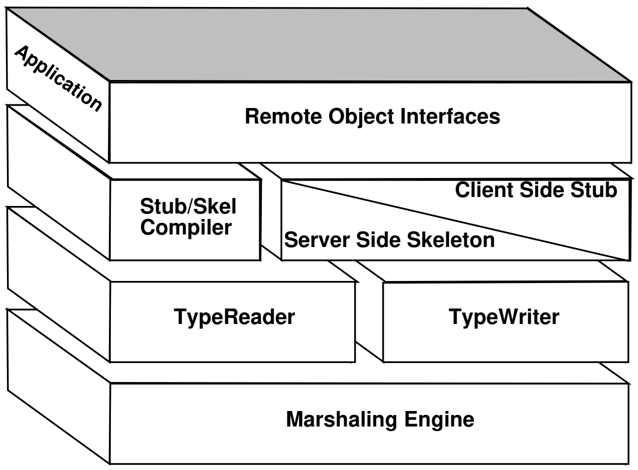


**Figure 7. Presentation layer architecture.**

There are two interface objects to this layer, *TypeReader* and *TypeWriter*, that represent the streams of primitive types to be read or written (Figure 7). An example of typical *TypeReader* and *TypeWriter* methods is:

```
// IDL
interface TypeReader {
    octet readOctet();
    short readShort();
    ...
}
```

```
interface TypeWriter {
    void writeOctet(in octet o);
    void writeShort(in short s);
    ...
}
```

## 3. Extending ACT

The issues presented in this section are of concern to the system programmer that needs to extend the functionality of ACT, e.g., to add a new application protocol to the system.

The ACT protocol layer can be used in two ways by a system developer. If the application protocol is already supplied by ACT the ACT's classes are used "as-is." Otherwise, the developer has to implement it following the generic protocol-implementation API. An advantage of ACT is that the design of the protocol layer provides another set of APIs for the system developer. Protocol rules for exchanging messages are generally implemented using *finite state machines* (FSM) and/or by hard-wired programs. Our model is based on FSM where the inputs and outputs are protocol messages, while states describe the current status of the transactions [4]. The programmer needs to provide the FSM transition table. All the protocol messages have to implement the *Message* interface, in order to be handled by the FSM engine. This interface specifies what is the desired destination of the message, and the transaction ID used to pair requests with replies. The interface method `write()`, which converts the structured message object into a byte-stream to be sent by the endpoint layer, has to be also implemented.

ACT can be extended by adding multi-threading policies. Two interfaces have to be implemented when adding a new policy, *ConnectionFactory*, and *RequestHandlerFactory*. They specify the way connections and requests are handled concurrently. The decisions they make are to queue the connections (requests) to be handled sequentially, to create a new threads, or to wait for available threads from a pool of threads.

## 4. Example Systems Based on ACT

In order to verify the claimed properties, ACT was used to develop the following two communication systems.

### 4.1. ACT ORB

ACT ORB is an OMG CORBA 2.$x$ compliant ORB. CORBA-specific components come as an extended part of ACT, and ORB is easily implemented using these components (Figure 8). Besides compliance to CORBA, the ACT ORB also provides the ACT interfaces to the internal layers, allowing the application to communicate through IIOP protocol without necessarily using the other CORBA components, such as stubs, skeletons, ORB object, or POA (Portable Object Adapter). This way high-speed communication with a non-ACT party can be achieved.

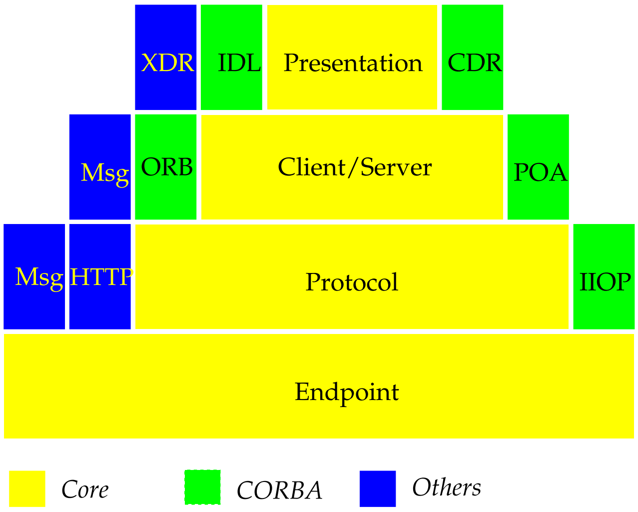


**Figure 8. Act core and supplementary components.**

### 4.2. cBus

The Collaboration Bus (cBus) is a distributed architecture for automated capturing and distribution of user events. cBus and ACT ORB are used to implement a collaboration enabling desktop that allows Java beans to be shared by multiple users simultaneously in a collaborative session [12]. cBus is structured in four layers, according to ACT's architecture (Figure 9). Using the Java delegation event model, cBus captures user events via event adapters. Event adapters transform events to/from a common format, much like stubs and skeletons. cBus Object Adapter dispatches remote events to the proper event adapter, similar to CORBA POA. Communicator deals with multi-threading policies and concurrency control for resolving conflicting actions of multiple users. cBus protocol layer uses either TCP/IP or LRMP (Light Reliable Multicast Protocol) multicast protocol [10] for communication between the collaborating users.

cBus has quite different communication requirements from a CORBA ORB. Having the ACT abstract model allowed us to implement cBus in a very efficient way, embedding group communication features, such as multiple concurrency control algorithms, application coupling and awareness, into the ACT layers. Other group communication systems build group- specific features on top of an ORB (e.g. [2] uses Java RMI) thus increasing system complexity

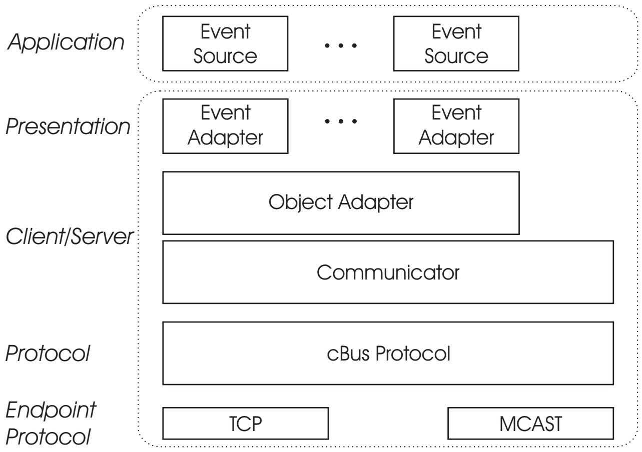


**Figure 9. Collaboration Bus architecture.**

and degrading its performance. A major insight from this work is that the design of both collaboration and communication infrastructure can be shrunk to a Broker pattern.

## 5. Practical Experience

### 5.1. Features of ACT-based Communication Systems

In the process of implementing ACT ORB and cBus we had drawn the following conclusions about the features of Broker-based communication systems built using ACT:

**Portability** It is estimated that about 10-20% of the programmers' effort concentrates on platform dependent code [7]. The Java version of ACT doesn't deal with platform dependency. The C++ platform dependent code is written by following the Java design.

**Modularity** One of the competitive aspects of ACT ORB compared to other CORBA implementations is modularity. ACT can be accessed at different levels of functionality, and still be interoperable with other CORBA compliant applications. Due to its modularity, one can detach the top-most two layers, and still keep the IIOP protocol layer to implement an application. Although the programmer now has to deal with low-level issues such as IIOP messages, the resulting application can be more efficient.

**Reusability** The ACT core implementation in Java has $2790$ non-commented lines. CORBA extensions account for $5005$ non-commented lines. Thus about $35\%$ of the ACT ORB implementation is the ACT core (Figure 10), which can be reused for building other communication systems. We expect this percentage to grow in the future as ACT gets extended and new features get incorporated in the core. In fact, a system programmer using ACT to build his/her own CORBA implementation can reuse accompanying CORBA specific components, e.g., IIOP protocol. In this case, the percentage of reusable code increases to more than $70\%$. Similarly, cBus extensions consist of $1919$ non-commented lines code. The percentage of ACT core in cBus system is $59\%$.

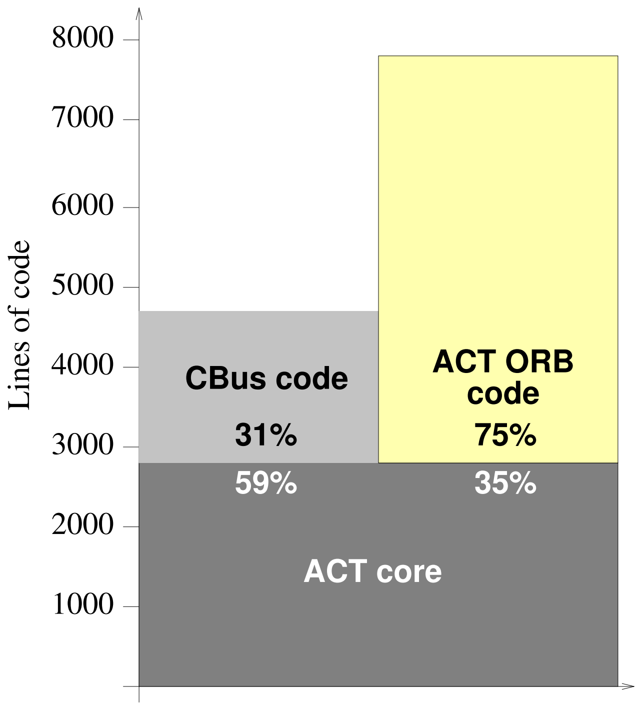


**Figure 10. Reusable code in ACT.**

**Extensibility** Implementation of a new high-level protocol requires no changes in the rest of the code. The expected amount of code to be written when implementing a new protocol is about 2000 code lines. This estimate is based on IIOP implementation that is 1200 non-commented lines of code. Implementing a new multi-threading policy can be done in 100 to 500 non-commented lines of code, since all three currently implemented multi-threading policies are less than 300 lines together. The implementation of new data marshaling should take about 400 lines since the current CDR marshaling is 383 non-commented lines.

**Adaptability** ORBs are usually written as monolithic code (except for TAO [13]). They might have more functionality than needed, resulting in poor performance, when, in fact, the developer needs a system having less features and high performance. In contrast, ACT can be used at any of the four levels of abstraction, by removing the unnecessary functionality. Incorporating built-in support for particular developer's needs is impossible in nowadays ORBs. On the contrary, ACT ORB allows the developer to build new primitives into the Broker's implementation core.

### 5.2. Performance

One of the major concerns when proposing the ACT model was to not make any decisions that could compromise performance. We believe that the model itself does not contain any inherent flaws and a carefully optimized implementation of ACT should yield in a competitive communication system. The current ACT implementation is optimized at the level of internal code design. Good design includes careful management of resources (memory, threads, sockets, objects).

Testing can not be performed outside a communication system that uses it. In our case we used ACT ORB as a test bed. The tests were conducted using Sun Ultra-sparc machines, connected by a 10Mb Ethernet network. Figure 11 shows the average round trip time per method call for both ACT ORB and Sun's JavaIDL ORB [16]. We measured the time of a remote call with no arguments and no return value. For small number of requests ACT ORB performs better than JIDL, due to a smaller time needed for establishing an IIOP connection. When the number of requests increases, both times stabilize asymptotically, ACT's time being about 2.5 ms larger. This graph shows that there is no significant impact on speed in an ACT-based ORB compared to a monolithic ORB such as JIDL.

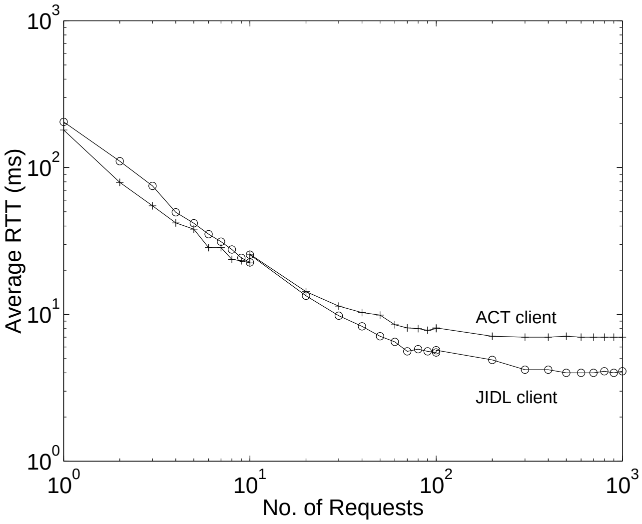


**Figure 11. Average round trip time per method call for ACT and JIDL ORBs. Note the logarithmic scale on both axes.**

Further improvements can be achieved by tuning the code with performance measurement tools, such as Rational Inc. Quantify, and this is part of our current work. Future work includes determining the amount of the performance impact due to the particular implementation, versus the one due to the ACT abstract model.

## 6. Related Work

As of now, there are few available toolkits that support current technologies (e.g., CORBA). We believe the closest to this work is ACE [13]; related systems are also BAST [6], CIF [1], and GTS [11].

The Adaptive Communication Environment [13] (ACE) is an object-oriented framework that implements many core design patterns for concurrent communication software. ACE provides a rich set of reusable C++ wrappers. A great amount of effort in ACE is focused on C++ wrappers for IPC mechanisms, concurrency mechanisms, threads, streams, etc. Java provides native support for most of these. Since ACE has to maintain it's own wrappers, when porting ACE to Java ACE, many inconsistency problems appeared [7]. ACT does not face these problems, since it uses directly the Java wrappers.

Although there are similarities between ACT and ACE, there are also major differences. ACT has a core skeleton around which customized components are added in order to build a functional communication system (Figure 8). This approach facilitates rapid building of communication systems [8]. ACT layers are encapsulated and each layer provides well defined functionality. A layer can have several implementations at the same time, tuned to different requirements. ACT provides extensive support for integration of user-defined application-level protocols, multi-threading policies, and data marshaling engines.

GTS system [11], is a highly configurable communication system, which supports the implementation of group-based applications. Compared to GTS, ACT covers more of a middleware-type functionality, such as thread-control, service management, and data marshaling. A comparison can be made between GTS and the first two layers of ACT. The protocol layer of ACT differs in that it effectively supports a large class of high-level protocols, by providing not only a generic interface, but also a powerful mechanism based on finite state machines, which helps the system programmer reducing the effort when implementing a new protocol. Another difference is compatibility with other systems. For instance, an ACT-based HTTP server can communicate with non-ACT HTTP clients. It is not the case in GTS, since GTS was not intended to help implementing the HTTP protocol.

BAST [6] is an object-oriented framework for building fault-tolerant applications. In many ways BAST is very similar to ACT's first two layers, providing different interfaces for the application programmer, and system programmer.

However, since ACT provides more middleware functionality at upper layers, the comparison has to be made between BAST and the protocol layer of ACT. BAST focuses rather on providing a great deal of flexibility, by allowing the system programmer to define his/her own interfaces to the system. In ACT, we decided that common interfaces have great benefits, such as easy replacement of one protocol with another, and seamless coupling with the application. In addition to semantics of protocols supported by BAST, ACT introduces the *finite state machine* model which requires less work from the programmer when implementing a new protocol.

CIF [1] project aims to develop a distributed computing software bus which supports software components for scientific collaboratories. The comparison is only possible between CIF and ACT's protocol layer, since the top-most two layers of ACT add extra-functionality non-existent in CIF. ACT's protocol layer offers structured protocol messages, taking the burden of converting the message data into a byte-stream from the application programmer. Furthermore, in addition to message passing, ACT enforces the protocol rules via an FSM and throws exceptions in case an illegal message is attempted to be sent. CIF does not provide support for the system programmer to develop a new protocol. In contrast, ACT offers powerful interfaces for the system programmer, which makes protocol implementation much easier. In case the protocol allows sending multiple requests, ACT interfaces allow the application programmer to block waiting for the answer to a specific request. This way, using complex protocols, such as CORBA IIOP, becomes easy. CIF API does not provide this feature to maintain simplicity.

## 7. Conclusions

The Advanced Communication Toolkit presented here is an effective solution for rapid development of Broker pattern-based communication systems. At theoretical level, ACT proposes a general architecture and interfaces allowing gradual increase in abstraction of communication. Major achievements consist of the uniform interfaces for using IPCs, high level protocols, multi-threading policies, and data marshaling formats. The generality of the interfaces is demonstrated by building ACT ORB and cBus. At a practical level, ACT addresses implementation issues like:

- *performance optimizations* independent of the protocol at the protocol layer. FSMs are used both to efficiently manage threads and open sockets and to smoothly integrate user-defined protocols [4]
- *multi-threading policies* (e.g., thread-per-request, thread-per-connection, and pool-of-threads) for implementing different types of servers
- *stream-based data marshaling* currently implemented for Common Data Representation (CDR) specified as part of IIOP in CORBA 2.$x$

Systems like CORBA, RMI, DCOM, or RPC are not customizable. Even if the application developer does not need the full functionality of an ORB, it is not possible to remove the unnecessary modules in order to gain memory and speed. This case occurs with hand-held computers, which usually have a small memory (currently up to 16MB), and almost no permanent storage. The operating system and necessary software is loaded at startup time from the network. Under these circumstances it is critical to have a light-weight communication system, implementing strictly the needed functionality, while maintaining compatibility with the desktop communication system. Suppose that, in our CORBA example, such an environment does not need DII/DSI, repositories, or CORBA services, but only high-speed object communication. In this case ACT can help in devising such a light-weight, customized CORBA-compliant system. This can be achieved by removing any number of the top-most layers.

ACT's novelty is that it offers an architecture for rapid implementation of the Broker pattern. ACT-based communication systems can be easily customized by the application programmers at different levels of abstraction of communication, thus providing flexibility for particular user needs or application requirements. ACT by itself does not enforce any application protocol, multi-threading policy, or data marshaling. Therefore an ACT-based system can communicate not only with other ACT-based systems, but also with any other system using the same protocol, data marshaling, etc. For instance, ACT can be used to develop an HTTP server, as well as an FTP client.

ACT source code, documentation, and examples are freely available at the following URL:

`http://www.caip.rutgers.edu/disciple/`


### Acknowledgements

The authors are grateful to Weicong Wang who made significant contributions to the implementation of ACT.


## References


[1] D. Agarwal, K. Berket, N. Narasimhan, P. Schabert, I. Foster, and S. Tuecke. The Collaboratory Interoperability Framework (CIF) Common Application Programming Interface . Lawrence Berkeley Laboratory, 1998. Available at `http://www-itg.lbl.gov/CIF/`.

[2] R. Burridge. Java Shared Data Toolkit (JSDT). Sun Microsystems, Inc., Mountain View, CA, February 1998. Available at `http://www.javasoft.com/people/richb/jsdt`.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern–Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, 1996.

[4] N. Dronavalli, C. Francu, W. Wang, and I. Marsic. An Optimized Design For Multi-Protocol Communication Systems. 1998. Available at `http://www.caip.rutgers.edu/~francu/perfv30.doc`.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley Publ. Co., Reading, MA, 1995.

[6] B. Garbinato and R. Guerraoui. Using the Strategy Design Pattern to Compose Reliable Distributed Protocols. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, Portland, OREGON, June 1997. Available at `http://lsewww.epfl.ch/bast/`.

[7] P. Jain and D. C. Schmidt. Experiences Converting a C++ Communication Framework to Java. *C++ Report*, 9(1), January 1997.

[8] R. Johnson. Frameworks = Patterns + Components. *Communications of the ACM*, 40(10), October 1997.

[9] Joint Submission. ORB Portability, Part 1, orbos. Technical Report 97-04-14, Object Management Group, Inc., Framingham, MA, April 1997. Available at `ftp://ftp.omg.org/pub/docs/orbos/97-04-14`.

[10] T. Liao. Light-Weight Reliable Multicast Protocol. INRIA Rocquencourt, France, June 1996. Available at `http://webcanal.inria.fr/lrmp/`.

[11] S. Maffeis, W. Bischofberger, and K.-U. Matzel. A Generic Multicast Transport Service to Support Disconnected Operation. In *ACM Wireless Network Journal*, 1996. Available at `http://www.olsen.ch/~maffeis/publications.html`.

[12] I. Marsic and B. Dorohonceanu. An Application Framework for Synchronous Collaboration using Java Beans. In *Proceedings of the 32nd Hawaiian International Conference on System Sciences (HICSS-32)*, Wailea Maui, Hawaii, January 1999.

[13] D. C. Schmidt. ACE: Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the $6^{th}$ USENIX C++ Technical Conference*, Cambridge, MA, April 1994.

[14] D. C. Schmidt and S. Vinoski. Comparing Alternative Programming Techniques for Multi-threaded Servers (Column 5). *C++ Report*, 8(2), February 1996.

[15] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., New York, NY, 1996.

[16] Sun Microsystems, Inc. Java IDL. Available at `http://java.sun.com:80/products/jdk/idl/`.

[17] The Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.1. Technical Report 97-09-01, Object Management Group, Inc., Framingham, MA, September 1997. Available at `http://www.omg.org/corba/c2indx.htm`.