Enabling Distributed Corba Access to Smart Card Applications

The OrbCard framework uses Corba wrapper technology to extend smart card services to a distributed computing environment.

Alvin T.S. Chan, Florine Tse, Jiannong Cao, and Hong Va Leong
Hong Kong Polytechnic University

With the increasing use of the Web as the platform for online e-commerce applications, smart card technology presents an attractive solution for providing security and access control mechanisms for processing online transactions. A user can insert the smart card into a reader attached to a terminal that provides Web access and client-side processing capabilities; the card performs all processing of secured information for the transaction using an embedded chip. Upon completing the transaction, the user removes the card without having transferred any secured data or algorithms to the hosting terminal.

Smart card services have not been integrated into the networked environment in the way that other portable computing devices — notebooks, PDAs, mobile phones, and so on — have been. With the growing need for distributed Web applications that support personalized services in a truly secured platform,[1] the smart card's architecture makes it an ideal device for storing an individual's personal information and service requirements.

Designed primarily for identification applications, a smart card operates through a built-in integrated circuit. The programming environment therefore lacks the architectural flexibility to support networked computing applications. Such limitations motivate our design for a distributed computing architecture that uses common object request broker architecture (Corba) wrapping technology to integrate smart card services. The design allows us to exploit Corba's flexibility to provide a heterogeneous platform and promote rapid application development. Corba provides a distributed object paradigm for smart card application development, and allows us to interoperate with existing and evolving Corba-compliant services. In this article we describe the architecture and implementation of the OrbCard framework, which leverages Corba middleware services to integrate smart card services with the object request broker (Orb) bus.

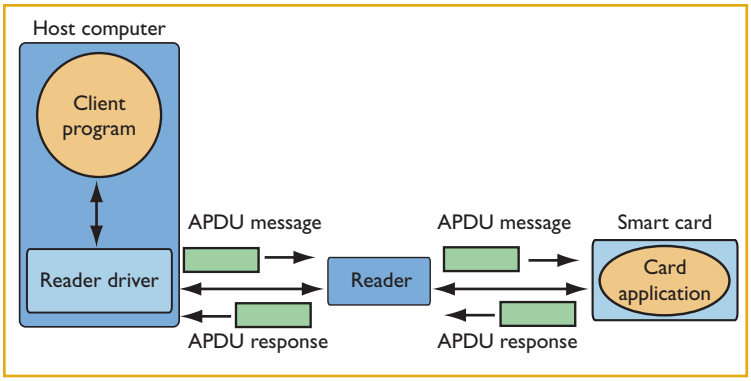

Figure 1. Communications using application protocol data units. The interactions between a host program, card reader, and smart card use APDU command set and responses.

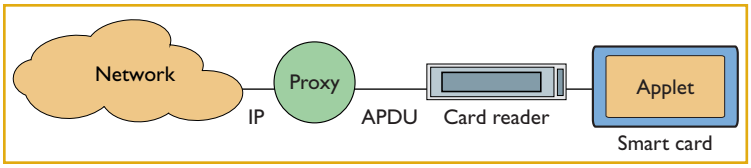

Figure 2. Socket approach. The proxy residing on the host acts as a gateway between the smart card applet service and the Internet.

Smart Card Application Development

Until recently, most applications of smart card technology were restricted to specialized domains and ad hoc systems, with little or no integration with mainstream information systems. A main inhibitor is the fact that, unlike the PC, smart cards lack a good operating system and development environment.[2] The absence of a common architecture and open programming interface has led to ad hoc implementations of smart card systems that are neither interoperable nor simple to develop.

Java Card

Sun Microsystems has recently developed the Java Card programming architecture[3] for smart cards to facilitate the concept of "write once, run on all cards." Java Card provides a lightweight Java virtual machine (JVM) that supports smart card application development using familiar Java language and programming paradigms. Importantly, the Java Card's JVM encapsulates the card manufacturer's proprietary technology through a common system interface.

Application Protocol

While Java Card resolves the important issue of providing a common programming interface, it maintains nonstandard and often ad hoc communications between the card's "applet" (not to be confused with Web browser applets) and the host terminal. Smart card implementations typically include both the on-card and off-card applications.[2] The off-card application, residing on the host terminal, sustains communications with the on-card application via the card terminal reader (see Figure 1).

To standardize the communication protocol between the off-card program and the on-card application, the International Organization for Standardization (ISO) has taken the initiative with standard ISO/IEC 7816,[4] which defines the command messages sent to and response formats returned by the card based on application protocol data units (APDU). Developing applications with this specification requires a detailed understanding of the card's command set, whereas the Java Card software package provides classes for constructing and parsing the primitive APDUs for message exchange. In either case, however, off-card and on-card applications still communicate using a nonstandardized, often unstructured, byte format.[5]

Networked Smart Card Applications

Traditionally, smart card applications developers have created stand-alone programs, dedicating minimal effort to incorporating smart card services directly into network environments.[5] With the increase in processing resources and the important role of smart cards in providing personalized and mobile services over the Internet, it is desirable that future smart card services can be added to interact and communicate directly over a distributed processing environment. Researchers are currently exploring several approaches to network-enabling smart card applications.

Socket approach. Socket programming for client-server distributed computing directly uses TCP/IP for end-to-end communications. As illustrated in Figure 2, using the socket approach to network-enable services on the smart card requires an off-card proxy to act as a bridge between the network and the on-card applet.

The proxy implements the socket interface to marshal and unmarshal network messages. The proxy parses and translates network messages into equivalent APDU messages for the on-card applet. Conversely, the proxy extracts and encodes data from the card's APDU response for the network. The application must provide its own format conventions for network communications and APDU exchanges. In addition, both the server and proxy

must marshal and unmarshal buffered messages communicated between the entities. The direct approach is manageable under a simple client-server communication model with a simple message format convention. Any upgrade to the service provision or application model requires direct modification of the source code and possibly the message and communication formats.

Java RMI approach. Java supports native distributed object computing through remote method invocation (RMI) (java.sun.com/marketing/collateral/javarmi.html). With RMI, a client object can call another object's method, irrespective of the object's physical location. In other words, a Java object can invoke an object in another JVM by using the published interfaces. The lightweight Java Card virtual machine, however, does not support RMI with its native, on-card services. Instead, we can use the proxy wrapper approach to map the on-card services to the RMI distributed object model. As in the socket approach, the proxy must bridge the RMI-specific protocol exchanges and the native APDU messages.

This higher level of distributed object services abstracts the entities' communication details from the distributed application. The programmer is thus freed from managing the processing and marshalling of messages across the distributed environment. While RMI provides a comprehensive distributed object model for remote object invocation, however, it applies only to Java objects. RMI's language-dependency represents a significant drawback because distributed computing must work across heterogeneous operating systems and languages.

Corba approach. The Object Management Group (OMG) developed Corba[6] as a middleware specification to realize the benefits of a truly interoperable and open distributed object-computing platform. Corba's interface definition language (IDL) separates an object's implementation (in native language) from the interface to the object bus, or Orb. Using Corba object brokering, our OrbCard framework can enable distributed services (possibly legacy, and written in different languages) to interoperate seamlessly with smart card applications.

OrbCard Programming Environment

With Java Card, programmers can employ well-known object-oriented design semantics and methodologies to develop on-card services and, compile and execute them on the Java Card virtual machine. However, detailed observation reveals that, although Java Card supports object-oriented development, the object services exposed by the on-card applications are far from object-oriented. As mentioned, the off-card program invokes the smart card application by communicating with on-card services through native APDU commands. Although developed as an object, the on-card service relies on the off-card program to construct those commands and to interpret its corresponding APDU responses. This violates the principle of object modularity, which promotes encapsulation and information-hiding through well-defined interfaces.

In designing OrbCard, we wanted to create a software architecture that incorporated the Corba framework seamlessly around smart card applications, so that on-card services would appear as part of the distributed Corba object community. Figure 3 shows how OrbCard maps smart card services to the Corba environment by leveraging Corba's object interactions and brokering services to integrate smart card services as part of the networked services.

The central element of the architecture is the OrbCard adaptor, which functions as a proxy gateway for on-card services. To communicate with an on-card service, the client Corba object invokes the OrbCard adaptor via the Orb bus. The adaptor captures all Corba-specific requests, translates them to their

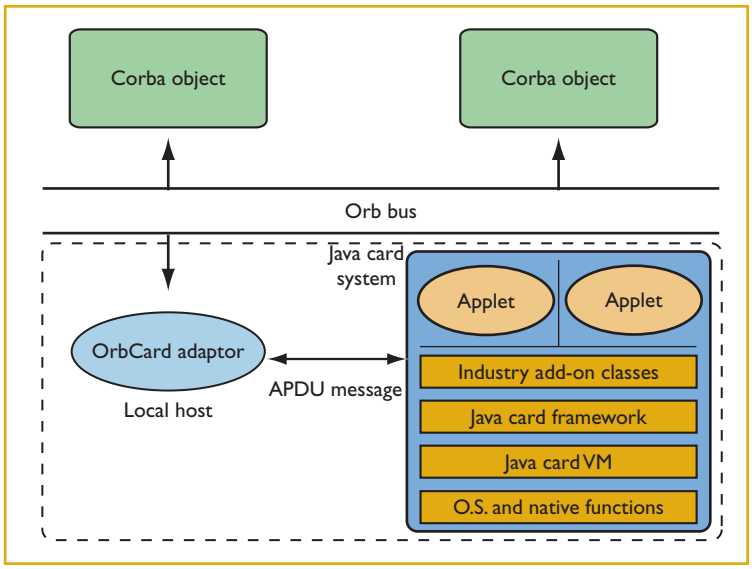

Figure 3. OrbCard architecture. To communicate with the on-card service, the client Corba object invokes the OrbCard adaptor, which captures all Corba-specific requests, translates them to their APDU representations, and routes them to the on-card applet.

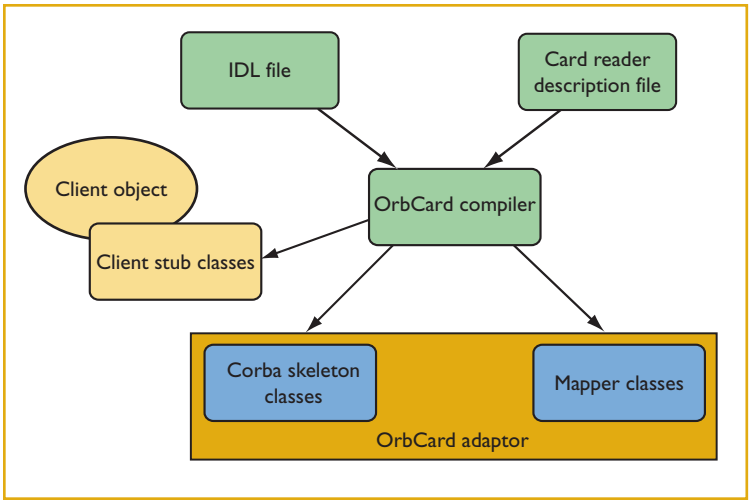

Figure 4. OrbCard compilation. This diagram shows the operational flow of the OrbCard compiler, which accepts IDL files and card reader description files as inputs. The class libraries generated from the compilation results in three groups of classes: the client stub, adaptor skeleton, and APDU mapper.

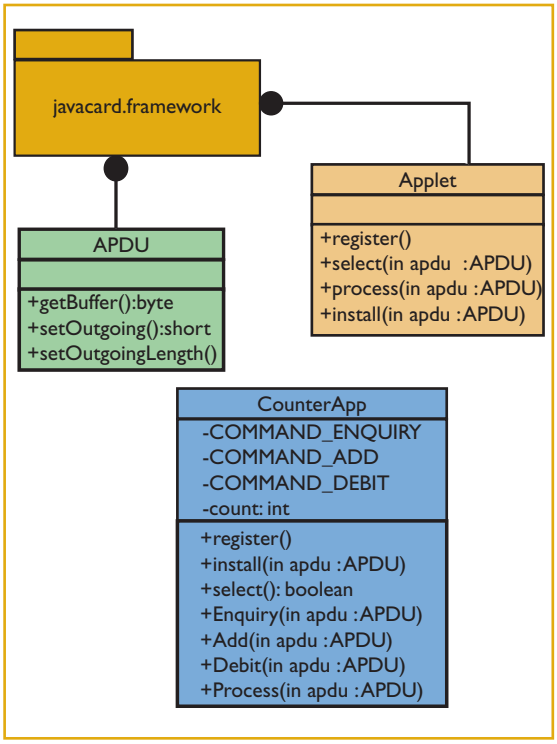

Figure 5. Class diagram of CounterApp applet. The on-card counter, implemented as a typical Java Card applet, offers three methods: add, debit, and enquiry.

APDU representations, and routes them to the on-card applet. Similarly, the on-card service presents responses to the adaptor as APDU representations. The adaptor maps and relays the Corba responses back to the calling client Corba object. The OrbCard adaptor's representation of the on-card service appears as a normal Corba object to typical Corba objects on the Orb bus. In fact, these off-card objects do not even need to be aware that the adaptor is functioning as a Corba wrapper over the service.

The development effort remains the same for on-card objects, which are implemented as normal Java Card applets. Our OrbCard compiler automatically generates all the necessary classes required to implement the adaptor functions, as well as the stub classes for the client object.

OrbCard's architecture enables us to specify interfaces to heterogeneous objects in a platform- and language-independent manner. The Corba IDL is a declarative language that allows objects to specify the contractual interfaces that will allow other client objects to invoke their services across the Orb bus. As with common Corba objects, OrbCard uses OMG IDL to specify interface attributes (parameters for generating the OrbCard adaptor and the interface stubs).

- The programmer defines an IDL file that describes the interface attributes and methods available from the on-card services. The file describes the object encapsulation of the service in terms of attributes, parameters, and available methods.
- The programmer runs the OrbCard compiler against the IDL file and the CardReader description file, which specifies the smart card reader type. The description file is required for the compiler to generate the correct classes to allow the adaptor to communicate with the reader using APDU as the native protocol.
- The class libraries generated from the compilation result in three groups of classes: the client stub, adaptor skeleton, and APDU mapper, as shown in Figure 4.

The client's object uses the client stub to statically invoke the smart card service residing on the Orb bus. The adaptor skeleton classes provide an interface to the Orb bus, which captures invocation requests and invokes the required methods of the mapper classes. The methods are proxy references to the actual services that the on-card applets offer. The mapper classes translate the invoked method to the equivalent APDU request-reply cycle to communicate with the requested on-card service.

Implementation

To demonstrate OrbCard's operation, we have

Related Work in Distributed Smart Card Applications

Distributed smart card applications development is not new. In fact, the smart card itself can be thought of as a dedicated machine that executes a service application. A host application follows a distributed client-server communication model to invoke a service by exchanging low-level messages. Vandewalle and Vetillard[1] propose an RPC-like communication protocol, called direct method invocation (DMI), for use in marshalling object method calls and returning results between the host application and the card's applet. DMI provides a distributed object-calling model in which communications between card applets and host applications are abstracted from low-level protocols. This approach treats the card's applet as a remote object whose services the host application invokes through an object-calling convention.

In developing a PDA-based smart card controller for managing digital signatures, Kehr, Posegga, and Vogt[2] propose using Jini as the infrastructure for seamless communication between devices. Jini provides centralized registration and service lookup. Potential clients can download a proxy object and use it as the basis for locating and invoking a desired service. In their implementation, the PDA and the card reader must register with a lookup service on the services offered. The smart card is primarily responsible for the signing service, and the PDA is a proxy for registering the service with the Jini lookup service.

In other work,[3] we have investigated building a proxy intermediary for Web-based access to a mobile repository of Web objects including html pages, medical data objects, and a record browsing and updating applet stored on a smart card. We developed the Java Card Web Servlet (JCWS) to provide seamless access between a Web browser and a Java-enabled medical smart card. As the patient changes location, the mobile smart card database can dynamically bind to the JCWS framework to facilitate ubiquitous access and allow medical staff to update the patient's information via a standard Web browser. Our OrbCard proxy gateway follows a similar approach to the others described here: the common goal is to provide an abstraction of the distributed computing model that seamlessly integrates with the platform, while encapsulating the low-level protocol exchanges in accessing smart card services.

References

1. J. Vandewalle and E. Vetillard, "Developing Smart Card-based Application Using Java Card," *Proc. 4th Smart Card Research and Advanced Application Conf. (CARDIS 98)*, Louvain-la-Neuve, Belgium, 1998.
2. R. Kehr, J. Posegga, and H. Vogt, "PCA: Jini-based Personal Card Assistant," *Proc. CQRE 99, LCNS 1740*, Springer-Verlag, Berlin, pp. 64-75, 1999.
3. A.T.S. Chan, "Web-Enabled Smart Card for Ubiquitous Access of Patient's Medical Record," *Proc. 8th Int'l World Wide Web Conf.*, World Wide Web Consortium (W3C), May 1999, pp. 1591-1598.

implemented a simple distributed client-server application that shows the complex interactions between objects within the framework. The client Corba object invokes a distributed on-card counter service via the adaptor object on the Orb bus using one of the on-card applet's three methods: `add`, `debit`, or `enquiry`. The counter service is implemented as a typical applet running over a Java Card virtual machine.

Figure 5 shows the class diagram for the implementation. The applet's IDL file describes the interface and methods available. The interface name and the IDL file name should match the applet's class name. In addition, the methods declared in the interface file should correspond to the APDU commands provided by the Java Card `CounterApp`. (Note that it is only necessary to declare the supporting methods that the application uses.)

Running OrbCard Compiler

Next, we process the IDL file with the OrbCard compiler tool, a Java application that generates Corba custom classes (including the client stub and implementation base classes) and a mapper class based on the input IDL file:

```
module CounterApp{
     interface Counter
     {
             void Add(int v);
             void Debit(int v);
             long Enquiry(void);
     };
};
```

All classes are generated in Java source. The syntax for executing the compiler tool is

```
OrbCard Counter.idl gemplus
```

The first argument specifies the required IDL file, and the second specifies the card reader type.

Corba implementation requires custom classes, such as `data marshalling`, `unmarshalling`, and `name service lookup`. The mapper class encapsulates the details of APDU commands and handles card reader communications. At execution, the OrbCard program generates the required Corba custom classes and processes the IDL file line by line to generate the mapper class.

The compiler reads the interface name in the IDL

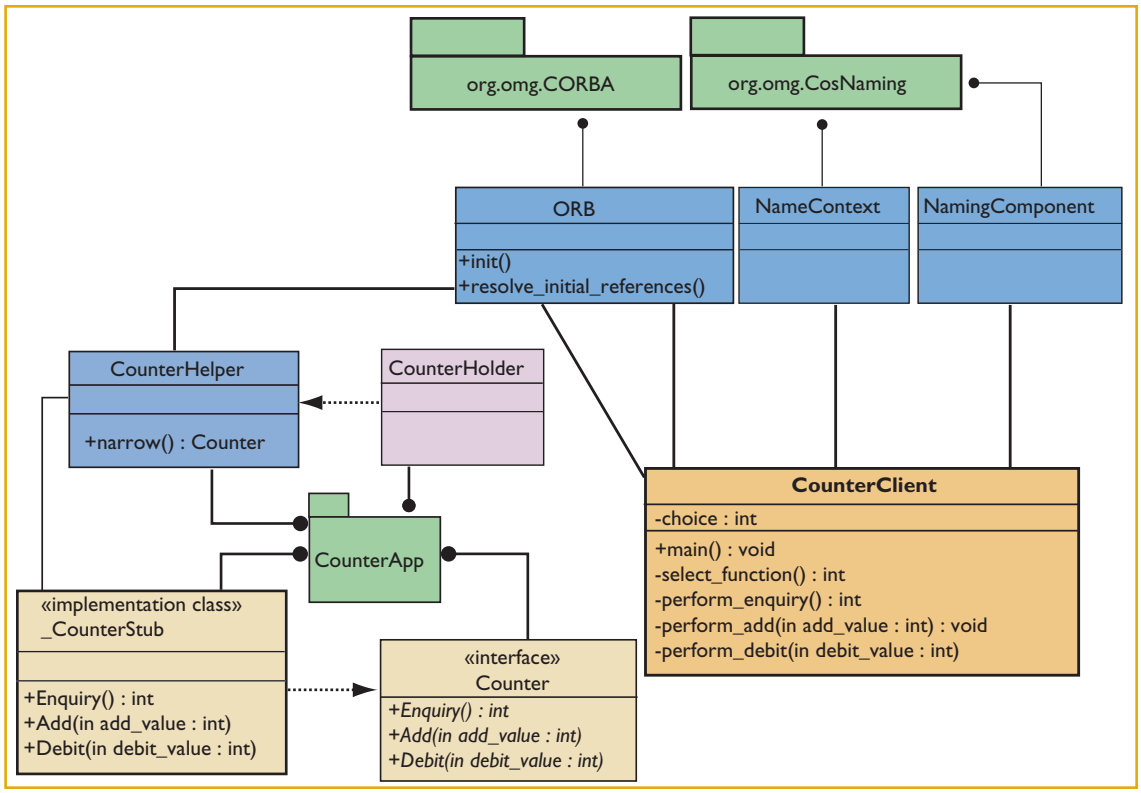

Figure 6. Class diagram for Corba client. The CounterApp package generated from the IDL compiler forms the core of the Corba wrapper classes.

file to determine which applet the IDL references. It must identify the applet because the mapper class will send a `select APDU` command with the applet's application identification (AID) to select the applet. An adaptor can handle a variety of applets, but only applets with a qualified AID will be generated in the mapper class. After identification, the adapter scans each method in the interface file and matches the method name with the applet's APDU command specifications. The compiler then creates a method with a syntax name that corresponds to the one specified in the IDL file. (`Enquiry` is a method in the IDL file as well as an APDU command, for example.) The method contains instructions to perform APDU constructions and handles card reader communications. To communicate with the card reader, the mapper class creates a `CardReader` instance with the reader type specified in the compiler's input argument.

Developing Client and Server Programs

Figure 6 shows the UML class diagram design for the client program, which implements the client object that invokes the methods of the Corba-wrapped, on-card services.

Figure 6 shows a typical Corba client, `CounterClient`, that imports Corba-specific packages such as `org.omg.CosNaming` and `org.omg.Corba`, and application packages such as `CounterApp`, for communication between client and server programs. The IDL file generates the `CounterApp` package via the compiler tool. This package comprises six classes: `Counter` (interface class), `CounterHolder`, `CounterHelper`, `CounterStub`, `CounterImplBase`, and `CounterMapper`. The first five are Corba custom classes; the last is required in server applications only.

`CounterClient` contains a main method and several private methods. When a client starts up, it calls the `resolve_initial_references` function in the Orb class to obtain an object reference for the name server — the Java IDL name server, `tnameserv`, in this case. In our implementation, the name server runs in the same terminal as the smart card reader. Next, the client invokes the `narrow` function to narrow the generic object reference to the `Counter` object reference. The mechanism is based on supplying the server's name path to the name server, which is similar to a file path in a typical hierarchical file system. Our project uses a single-level server name path: Counter. The returned object reference is an instance of the `_CounterStub` class, which implements the `Counter` interface class.

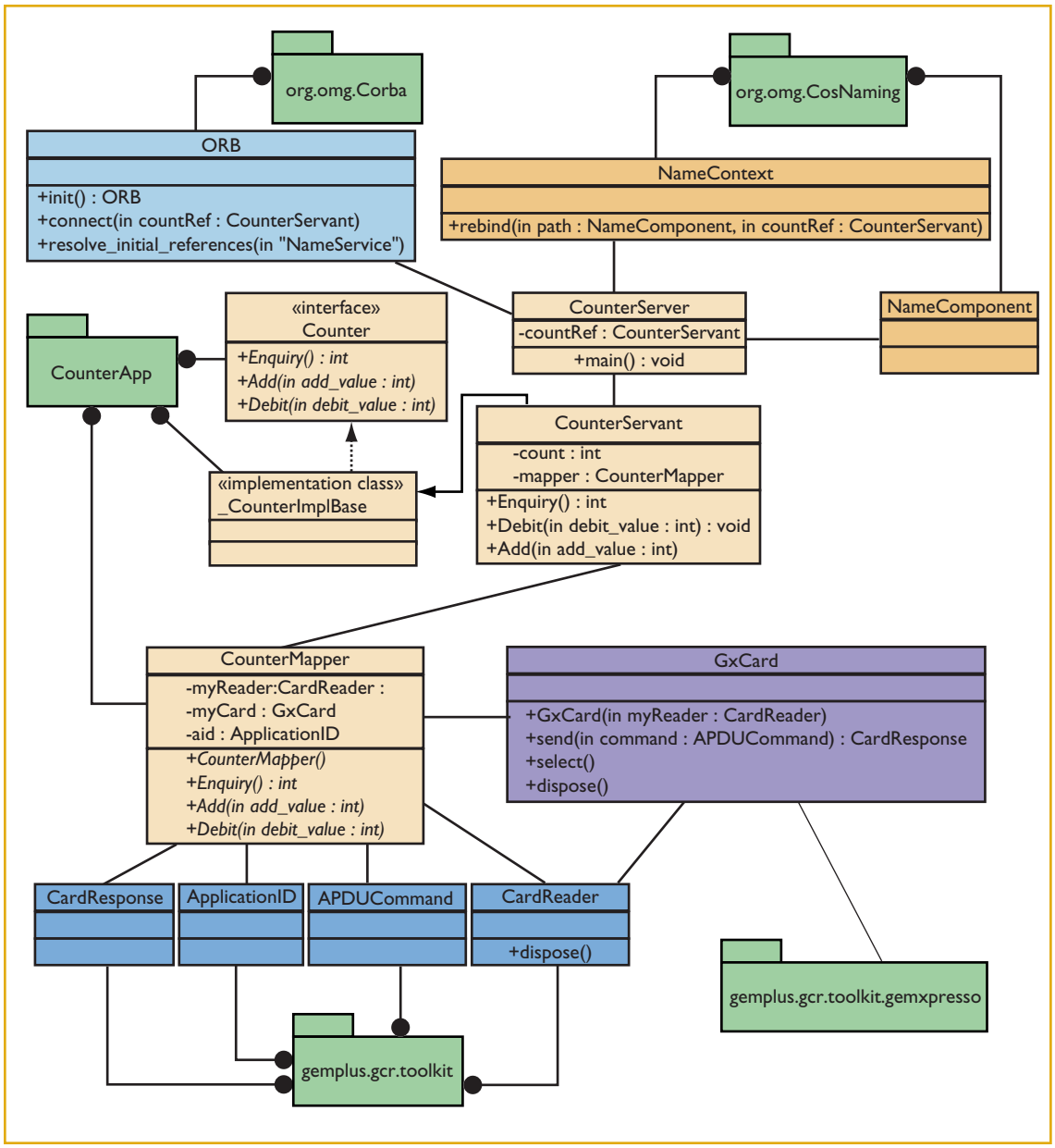

Figure 7. Class diagram for server. The server program references CounterApp and the Corba-specific packages. CounterServer, the main class in the server program, performs Corba initialization.

The client application presents a user menu with three selections: `Add a value`, `Debit a value`, and `Enquire`. The user selects an item and enters the corresponding parameter value, and the client application invokes the corresponding on-card service object method. Specifically, the proxy method passes parameter data to the `Counter-Stub`, which marshals the data and routes the request to the server-side Orb.

Figure 7 shows the class diagram for the server implementation. As with the client program, the server program references `CounterApp` and the Corba-specific packages. `CounterServer` is the main class in the server program, and it performs the necessary Corba initialization. `Counter-Servant` contains the implementation details of the IDL interface. `CounterMapper` maps invocations to APDU commands. `CardReader` performs low-level communications with the reader. When the server bootstraps, it performs two steps: registering the server name with the name server, and entering a loop waiting for the client request.

When the server receives a client request, it invokes the corresponding method in the servant class, based on the request's property. The servant class first checks whether there is an instance of the

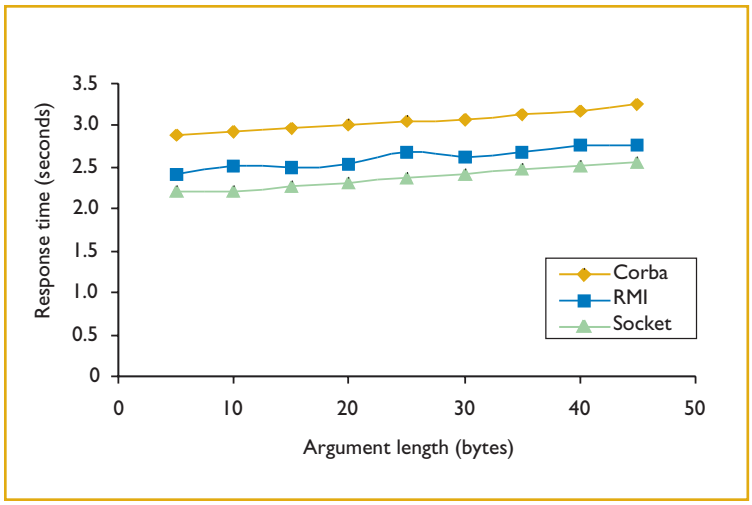

Figure 8. Response time vs. argument length. OrbCard's performance cost is around 19 percent, which we believe is acceptable given that Corba is an industrial distributed object standard that is interoperable across heterogeneous environments and languages.

mapper class and creates a new one if needed. The mapper class constructor, in turn, creates an instance of `CardReader` and selects the Java Card applet immediately by sending a `SELECT` APDU command. Next, the servant class passes input arguments to the corresponding mapper class method. The mapper class constructs an APDU command with the input parameters and invokes `CardReader`'s send method to transmit the APDU to the Java Card counter applet. The counter applet shown in Figure 5 was developed as a standard Java Card applet. It references the javacard.framework and java.lang packages. When the applet receives an APDU command from the `CounterMapper`, it checks the instruction byte stored in APDU and executes the requested function. Meanwhile, the mapper class waits for the response APDU, decodes the embedded data, and returns the data to `CounterServant`.

Performance Evaluation

To evaluate the performance of the three approaches we considered for our distributed smart card implementation — socket, RMI, and Corba — we compared response times for each. In the experiments, the client ran on a remote machine and communicated with the server application across a token-ring network. The server program formed a proxy to the Java Card applet, which implemented the counter application.

The performance evaluation measured the distributed application's response time from when the client program issued the request to when it received the reply. Hence, the response time includes the network transmission time and processing time for all the software components along the request-response path.

$$t_{resp} = t_{network} + t_{client} + t_{proxy} + t_{APDU} + t_{applet}$$

To test the approaches, we performed two experiments using two identical Pentium 500 PCs with 64 Mbytes of RAM and 20-Gbyte hard drives, connected via IBM token ring 16/4 network interface cards. The first experiment measured response time for increasing argument lengths. In this case, the argument represents controlled-length data (in bytes) sent as part of the invocation argument. The second experiment measured response time with an increasing number of requests.

Argument Length

In the first experiment, the client application invokes the `add` service in the smart card with argument lengths varying from 5 to 45 bytes, in 5-byte steps. For each sample, we conducted 500 cycles and averaged the results. As the diagram in Figure 8 shows, response time generally increases with argument length, mainly because of increasing data transmission time across the network and into the smart card.

The socket approach, which uses native TCP/IP directly, performed best among the three approaches because it does not incur extra processing overhead at the middleware layer. The program natively handles the marshalling and communication of data across the network. Between the middleware approaches, RMI performed slightly better than the Corba approach because RMI is the "native" middleware layer for Java-based distributed application environments. As such, the services are better integrated to the framework and incur minimal overhead. When comparing the RMI and Corba approaches, the performance difference is around 19 percent, which is acceptable considering the fact that Corba is an industrial distributed object standard that is interoperable across heterogeneous environments and languages.

Number of Requests

In the second experiment, the client application invoked the `add` service on the smart card continuously with a loop count of 0 to 5500, in steps of 500. We measured the response time for every loop size and computed an average over 500 testing cycles. As in the previous experiment, the invocation from the client to the applet was synchronous, such that the client proceeded to the next invoca-

tion only upon receiving a reply from the adaptor proxy. Figure 9 shows the results of the experiment.

As expected, the response time for all three invocation methods increased almost linearly with the number of invocations per loop count cycle. As in the first experiment, the socket approach performed best. Given Corba's benefits as a widely accepted and interoperable distributed object computing platform, we believe that the slight performance drawback is acceptable. This is particularly true for OrbCard applications that do not incur extensive invocations on smart card services over the network.

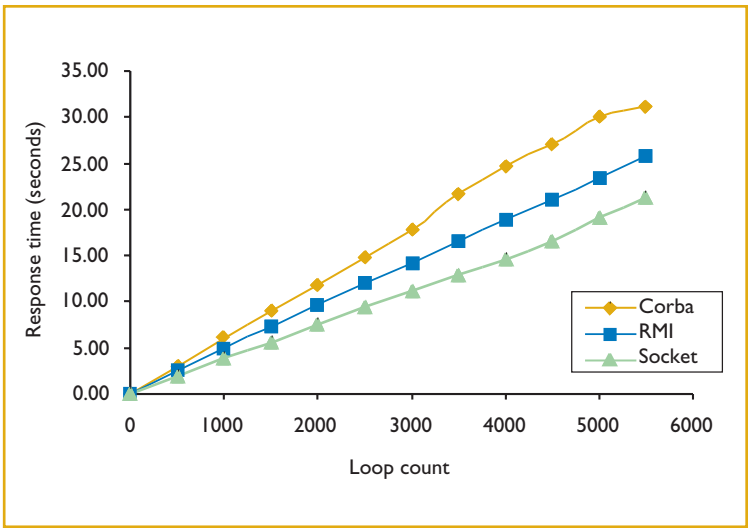

Figure 9. Response time vs. loop count. The client application invoked the add service continuously with a loop count of 0 to 5500, in steps of 500. Averaging response time over 500 testing cycles, we found that Corba generally performed modestly poorer than the other approaches with increasing invocations.

Future Challenges

The pocket portability of smart cards translates to the benefit of strong binding to the owner's identity. A card can carry relevant authentication information, such as a digital certificate, as the cardholder moves without restriction. Practical smart card applications must address various security concerns, however, in order to operate over public networks such as the Internet. To make it broadly successful, the OrbCard framework could be extended to include several possible solutions.

A two-stage approach to security in distributed Java Card applications can exploit digital signature and authentication capabilities on the card and in the proxy. While digital certificates and public key encryption possess numerous advantages, their high computational overhead makes them unacceptable for some smart card contexts because the typical smart card has limited processing capabilities (although some include a dedicated hardware encryption engine to accelerate processing). On the other hand, a trustworthy proxy (at least with respect to the smart card holder) could reduce the encryption requirement. The card and proxy can communicate in clear text, and the system can generate a session key for the cardholder for each smart card connection session and encrypt Internet communication for messages from the proxy to other Corba entities. This is reasonable, especially when the OrbCard proxy is collocated with the card reader.

To further improve security, we need to enforce end-to-end authentication before remote service invocations. Performing authentication within the Java Card would require public key encryption and message digest computation, as with the MD5 algorithm, which imposes higher computational demands on the card. We could employ digital certificates, timestamps, and a message digest signed by both parties, which could be exchanged and verified by the card and the networked service. Alternatively, the Java Card could pass its certificate to a trusted proxy, which could attach its certificate, sign it, and forward it to the appropriate Corba module. A known trusted party could issue certificates for Java Cards that the Corba system could easily authenticate. After the end-to-end authentication procedure, the trustworthy proxy would encrypt all other service messages using the session key. This message encryption could be overloaded to the relevant classes generated by the OrbCard compiler by extending the functions for the various interfaces, especially those related to argument marshalling.

Acknowledgments

The authors would like to thank the reviewers for their critical comments and helpful advice. This project is supported by the Hong Kong Research Grants Council (RGC) competitive grant under the account code B-Q453.

References

1. I. Cingil, A. Dogac, and A. Azgin, "A Broader Approach to Personalization," *Comm. ACM*, vol. 43, no. 8, Aug. 2000.
2. U. Hansmann et al., *Smart Card Application Development Using Java*, Springer-Verlag, New York, 2000.
3. S.B. Guthery, "Java Card: Internet Computing on a Smart Card," *IEEE Internet Computing,* Jan./Feb. 1997, vol. 1, no. 1 pp. 57-59.
4. *ISO/IEC 7816: Integrated Circuit(s) Cards with Contacts –*

Part 3: Electronic Signals and Transmission Protocols, International Organization for Standardization (ISO), Geneva, Nov. 1994.
5. A.T.S. Chan et al., "Web-based Smart Card for Application in Health Services," accepted for publication in *Comm. ACM*.
6. M. Elenko, J. Jones, and G. Palumbo, *Distributed Object Architectures with Corba*, Cambridge Univ. Press, June 2000.

Alvin Chan is currently an assistant professor at the Hong Kong Polytechnic University and director of a university spin-off company, Information Access Technology, Limited. He received a PhD from the University of New South Wales, Australia, in computer engineering. His research interests include computer networking, mobile computing, and context-aware computing. Chan is a member of the ACM and IEEE.

Florine Tse received her MSc degree in software technology from the Hong Kong Polytechnic University in 2001. Her research interests include distributed computing in Corba, Java technology, and smart card applications.

Jiannong Cao is an associate professor in the department of computing at Hong Kong Polytechnic University. He received a BSc from Nanjing University, China, and MSc and PhD degrees from Washington State University, all in computer science. His research interests include parallel and distributed computing, computer networks, Internet computing, fault tolerance, and mobile computing. Cao is a member of the ACM, the IEEE, the IEEE Computer Society, and the American Association for the Advancement of Science.

Hong Va Leong is an assistant professor at the Hong Kong Polytechnic University. He received a PhD from the University of California, Santa Barbara, in computer science. His research interests are in distributed systems, distributed databases, mobile computing, Internet computing, and digital libraries. Leong is a member of the ACM and the IEEE Computer Society.

Readers can contact the authors at cstschan@comp.polyu.edu.hk.