An Application Framework for TCP/IP Applications

Atsushi Kumazaki
Dept. Info. Sys. & Quant. Sci., Nanzan Univ.
18 Yamazato, Showa, Nagoya, Japan 466-8673
d00bb002@nanzan-u.ac.jp

Masami Noro, Han-Myung Chang, Yoshinari Hachisu
Dept. Info. & Telecomm. Eng., Nanzan Univ.
27 Seirei, Seto, Japan 489-0863
{yoshie, chang, hachisu}@nanzan-u.ac.jp

Abstract

TCP/IP applications have become increasingly important in this network generation. Their development, however, is troublesome in practice since reusable components are usually system-call-level service routines. We describe an application framework which we have developed for TCP/IP applications. The software architecture, as the model of the application framework, incorporates features of the layered and object-oriented style under the concept that a TCP/IP application consists of distributed objects communicating with one another. The architecture isolates state transition manipulation from such application logic as client user interface handling and server database access. The framework provides a large reusable component set including customizable components. We demonstrate the usefulness of this application framework through the implementation of several TCP/IP applications.

1. Introduction

In this new world-wide computer (IP) network generation, computers are no longer used in a stand-alone fashion. Operating systems must have TCP/IP applications as their basic components. From a software engineering perspective, however, the development of TCP/IP applications is chaotic.

Two major drawbacks can be pointed out in the development of TCP/IP applications.

- Knowhow for their development is limited to a few very *experienced* programmers.
- Implementation is carried out in a procedure-oriented fashion and, as a result, reusable components are at a level as low as operating systems' supervisor calls.

These problems coexist and are intimately related. The reason for the chaos is that UNIX hackers historically have developed the applications with no software engineering knowledge. In conclusion, current TCP/IP applications lacks flexibility, which limits their extensive use. Given these limitations, higher productivity and reliability are a dream that will never be realized.

Better software architecture yields better quality of software (easy to modify, understand, maintain, *etc*). Many research projects on software architecture [3, 5, 9, 12] reveal a need for productivity and indicate that it is one of the most important issues for software reuse.

Our approach to the problem above is to construct a domain specific software architecture for the applications and to implement the architecture. That is, for the purpose of higher productivity, and more reliable and flexible TCP/IP applications, we designed and implemented an application framework, program generator, and highly modulated IP communication components, based on the architecture. The key idea for constructing the architecture is that a client and a server are concurrent objects which communicate with one another.

This paper describes IPaf-R^2 (Internet Protocol application framework Revised twice), an application framework for TCP/IP applications. We also describe, in brief, the generator and the components for IP communication. IParch-R^2 (Internet Protocol architecture Revised twice), which is the base architecture of the framework, is also introduced.

2. Related Works

There have been several research projects on topics related to ours. Socket++[1, 2], for example, provides an object-oriented application interface of a socket. Although it enables the object-oriented development of TCP/IP applications, reuse still remains at the supervisor-call level. On the other hand, IPaf-R^2 reflects the architecture of TCP/IP applications and uses a class library which is akin to Socket++. In addition, it is designed and implemented as a lower-level component.

Conduit+[7] is called an application framework for TCP/

IP applications. However, it does not have a software architecture which provides a development guidemap. It has only several design patterns[6] such as the command, the state, *etc.* on as its background. The IPaf-R^2 model employs IParch-R^2 which includes a general system structure and development guidemap.

Existing Object Request Brokers such as CORBA[4] and so on are able to support IParch-R^2's key concept, which is that a client and a server in TCP/IP applications are concurrent objects running on isolated IP-connected machines. Although ORBs support TCP/IP application development under the key concept we have devised, the software may operate inefficiently. Moreover, they are incompatible with existing applications, since the communication protocol they use is not based on IP. We implemented an efficient ORB which conforms to de facto standards (IP), and then used it from code in IPaf-R^2.

To sum up, IPaf-R^2 based on IParch-R^2 1) gives development guide map of TCP/IP applications, and 2) is compatible to existing TCP/IP applications.

3. IParch-R^2: a Software Architecture for TCP/IP Applications

From the experience obtained when we defined domain specific software architectures[10, 11], we realized that a software architecture is not just a product model, but also a container including a process planning scenario. The insight we gained meets the definition by the SEI discussion group in '94 of a software architecture that "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." This results in the paradox that a product defines a process, and we conclude from this that:

- shapes of components vary vertically across development stages,
- interrelationships bridge between one stage and another, and
- they, in combination, implicitly show the order of component construction.

In Fig1, we abstract a software architecture which defines a process. We assume object-oriented software development here. As shown in the figure, the software architecture includes components from the software model, the implementation stage, and the design result. The software architecture shown in Fig1 indicates the process for developing the software abstracted in the architecture. We can easily see where design and implementation are needed component by component. The general scenario of the development can be stated as follows.

- Extract implementation-level component and plan to select the appropriate class or component library. Or, implement another subclass in a hierarchy.
- Implement classes and their hierarchy for the design-level component.
- Carry out the design and implementation for a component from the software model.

These three activities can be performed concurrently. Termination of all these activities triggers the coordination of the component. Then, we will proceed to testing and operation.

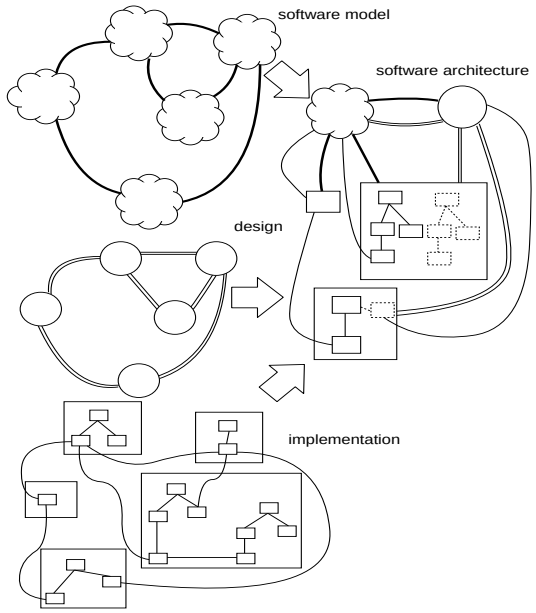

Figure 1. Software Architecture

Based on the discussions above, by definition our software architecture includes the following three views: abstract view, concrete view and process view.

It is an improved and revised version of Kruchten's definition[9]. The reason why we did not use Kruchten's definition of an architecture is that his definition assumes the same level of abstraction for the components of the architecture and a single type of connection among the components. In our model, the abstract view corresponds to component and connector model of the software architecture. As mentioned above, a component comes from one of any development stages, while a connector vertically and/or horizontally ties components together. The concrete view identifies a characteristic of a component. It shows which component is incomplete (that is, application framework), or what kind of technique can be applied to design and/or implement a component. In the process view, the partial order of component development is given.

3.1 Abstract View

Through our observations, we realized that a TCP/IP application is nothing more than distributed object-oriented software. That is, almost all TCP/IP applications consist of clients and a server which are concurrent objects on different machines. Once we could implement inter-object communication over a network, we could assume that half of the development is completed (although the ratio of time spent on communication versus features development varies from one development to another). The remaining half of the development includes implementation of state transition ma-

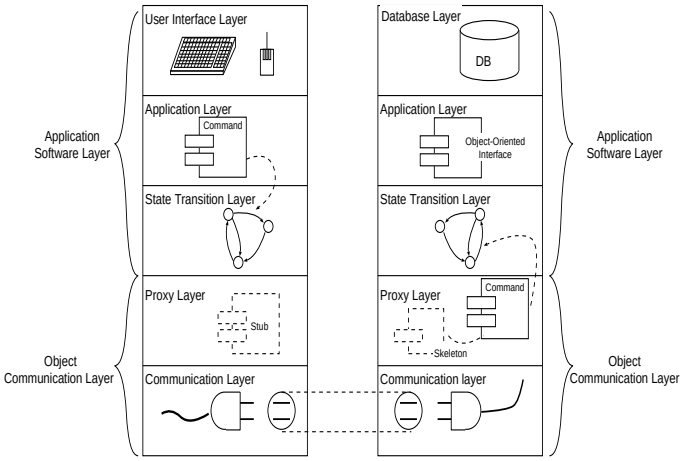

Figure 2. IParch-R^2**: Outline of Abstract View**

nipulation for the clients and the server. We should also consider the client-user interface and the server database.

IParch-R^2, shown in Fig.2, is a natural abstraction of the structure observed. As shown in the figure, the software is made up of two main layers: the object communication layer and the application software layer. The object communication layer is a platform on which applications run. All TCP/IP application logic excluding data communication are stored in the application software layer.

The object communication layer is further partitioned into two layers: the communication layer, which realizes a virtual circuit(TCP) and a datagram communication protocol(UDP), and the proxy layer, which provides inter-object communication service. Since the proxy (object request broker) approach is now the de facto standard for object-message exchange over a network, and virtual circuit and datagram communication are also standard for a computer communication protocol, we took a two-layered approach in designing an architecture for object communication. We decided not to choose other alternatives such as allocating distributed objects in unified memory space as distributed shared memory, implementing a specialized distributed object-oriented programming language, *etc.* Such approaches, which seem to be elegant, would lose practicality and popularity since most software of the de facto standard use TCP/IP rather than distributed shared memory.

The application software layer is further divided into three layers. The lower one is a state transition layer which represents automata of a client and a server. The intermediate layer, called the application layer, stores objects representing application logic in the form of the command pattern[6]. Input through the user-interface causes a state transition of a client on a state transition machine. The state transition machine receives a command object, and the transition, in turn, invokes the command action. The action invocation affects the state transition of the server and read-and-write operations on the database layer through the server's application layer, which provides object-oriented interface for database access.

3.2 Concrete View

The concrete view defines the implementation techniques that must be employed for the development of software based on the architecture. That is, the view indicates what kind of techniques must be used for the implementation of the components in the abstract view.

We have identified three important items that can help in the development of TCP/IP applications. They are a component library, an application framework, and a generator.

A component library is a set of general but small reusable components. An application framework is a domain-specific incomplete product which has holes. By losing generality, the reuse ratio becomes higher. Filling in holes completes the product easily. A generator is useful when there is a large semantic gap between a specification we give and generated code. It can be implemented if we rigorously write out the specifications and if translation is mechanical.

Component Library for Communication Layer

Objects in the communication layer are used to realize the transport layer's protocols (TCP and UDP). Since these two protocols are stable, we should provide a component library for this layer. Major operating systems provide socket system calls in a procedure-oriented way. We have to provide an object-oriented application program interface for the sockets.

Generator for Proxy Layer

Proxy creation is a case in which a generator approach is reasonable. Proxies (a stub for a client and a skeleton for a server) can be generated from the interface if a class (e.g. Java's interface, public entries of a C++ class), and the size of the generated code is not too small. Thus, it is better to implement a generator rather than to have component libraries. Designing and implementing an application framework for the proxy is impossible since the proxy is completely application dependent.

The Application Framework for Higher Three Layers

The higher three layers include application dependent objects. The common framework to all TCP/IP applications is that all of them are state transition machines which trigger application logic of their own. That is, state transition manipulation and a method for triggering applications can be abstracted as the application framework. Its details will be given in the next chapter.

3.3 Process View

The process view defines partially ordered steps for implementing objects in accordance with the concrete view

In IParch-R^2, the following scenario can be described:

1. Give the specification of a TCP/IP application to implement.
2. The following two steps are concurrent:

– State transition machine development.

1. Designing state transition machine according to the protocol specification.
2. Customize application framework.

– Proxy generation.
1. Select out of the library an appropriate component for the transport layer.
2. Indicate the object-oriented interface for the server.
3. Use the generator to create proxy codes.

Thus, by employing the process view of IParch-R^2, we can derive a process scenario for the development of the software.

We will discuss the development process in more detail after explaining the details of IPaf-R^2.

4. Design of IPaf-R^2: An Application Framework for TCP/IP Applications

The design guidelines of IPaf-R^{2}are as follows:
- The abstract view of IParch-R^2 is adopted for the model of IPaf-R^2 to realize that IPaf-R^2 is to be a part of the guidemap for the development of software.
- For the flexibility and maintenance of IPaf-R^2, an object-oriented concept is taken into the design.
- Design patterns[6] are actively used for making IPaf-R^2 the jargon of TCP/IP application developers.
- In order to ensure that it is popular, it must be designed to be compatible to existing TCP/IP applications.

4.1 Runtime Processes

First of all, regarding the design of an application framework, we have to consider what type of entity would be considered a heavy-weight process or a light-weight process. As we discussed above, almost all TCP/IP applications are a set of concurrent objects, a server and a client. This concurrency means that:
- a client has to wait for the user's input and the server's response at the same time, and
- a server can accept requests from multiple clients.

We design the client as a heavy-weight process and to do a selective wait for user's input and server's response. There are two more alternatives for designing the client. One way is that we design the client as a set of heavy-weight processes: one for the user's input and the other for the server's response. In this approach, the resultant client will be incompatible to the existing ones. The other way is to design the client as a set of light-weight processes. This approach, however, restricts the development to those environments which provide a light-weight process manipulation facility such as JDK for Java, Mach, *etc.*

We designed a class which selectively waits for the user's input and the server's response. The class is designed to communicate with objects in the user interface layer and communication layer. This implementation slightly modifies the IParch-R^2 structure. We adopted the modification for the purpose of compatibility to existing TCP/IP applications and for the generality of the development environment.

In general, a daemon process receives clients' requests on the server-side. After a request, a child process is spawned which serves to the client. There are no better ways to design a server. We decided to adapt this mechanism for the server in IPaf-R^2. That is, a server code in IPaf-R^2 is designed to create its copy when it receives a request from a client during runtime.

4.2 Application Framework for Clients

A client is a state transition machine which has a user interface. This means that we have to isolate user interface code from the machine in order that we may realize a client that is flexible in dealing with the change of a state machine's logic. Also, actions triggered by state transitions must be separated from the machine itself in order to ensure flexibility against the changes to the actions. That is why IParch-R^2's abstract view separates entities in these three categories into different layers.

Based on the structure given by the abstract view of IParch-R^2, objects in these three layers work as follows:
- An object in the user interface layer receives an input from a user and transmits the input to another object in the application layer.
- The object in the application layer creates an action in the form of the command pattern[6] from the transmitted input.
- The state transition machine triggers the action as its state changes.

4.3 Application Framework for Servers

A Server is a state transition machine including a server database. It is necessary to separate codes for database access from the machine in oerder to ensure that the server remains flexible enough to deal with access logic change. In addition to this, codes for database access must be partitioned into two parts, one for database management and the other for providing an object-oriented interface, since the database usually handles a file in the UNIX file system and IPaf-R^2 is object-oriented. These entities are in three different layers in IParch-R^2.

The server's objects in these three layers of IPaf-R^2 behave according to the scenario given below:
- The state transition machine changes its state, fires the action given by an object in the proxy layer, then triggers a database access.
- An object which provides an object-oriented database accessing interface in the application layer receives the access and interprets it into instructions in the form of a procedure-oriented database access.
- A UNIX file encapsulated in the database layer is accessed with the interpreted instructions.

5. Implementation of IPaf-R^2

IPaf-R^2 is written in C++ in order to make this framework widely accessible. In order to implement a flexible application framework, we have actively used design pat-

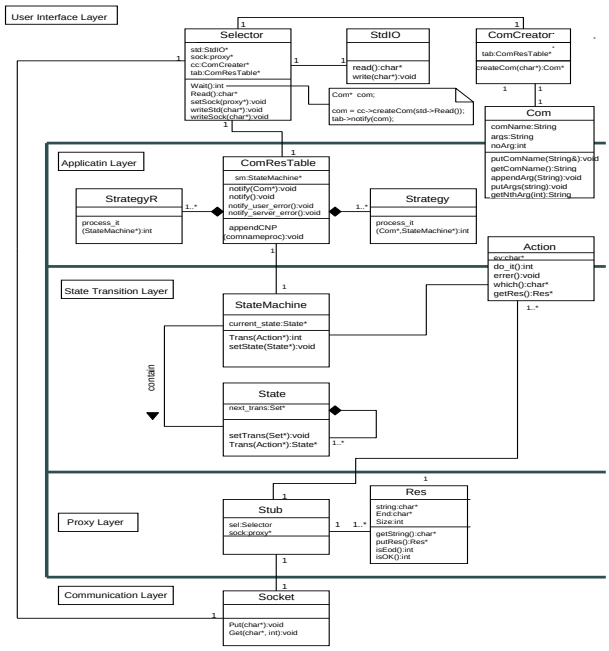

Figure 3. Outline of Application Framework for a Client

terns[6]. The patterns we have used are mainly the Command, the State, and the Strategy. This section gives details of the implementation of IPaf-R^2 and also discusses the patterns we used.

5.1 Implementation of Client's Application Framework

Fig.3 outlines our application framework for a client in a UML class diagram.

Based on the discussion of IParch-R^2 above and the design of IPaf-R^2, we can make the following decisions:

- In order to meet the de facto standard of TCP/IP applications, it is necessary to design and implement an object which selectively waits for a user's input and a server's response.
- An object of the command pattern is created from a user event in the application layer.
- The command pattern object executed as state transition is performed in the state transition layer.

The User Interface Layer

In the user interface layer, we implemented a class for standard input/output (`StdIO` in the figure.) Since the requirement called for in the user interface varies from one application to another, it is almost impossible to design and implement a stable framework. That is, most of the framework is occupied by hot spots. It is meaningless to design a framework that has such a small number of frozen spots. For the design and implementation of a set of such application-dependent objects, we decided that only `StdIO` is provided and made this layer open-ended to an application designer or implementer. In the design and the implementation of the application in this layer, the designer might use other component libraries for building the user interface.

Interface between The User Interface Layer and The Application Layer

Class `Com` in the figure above plays a very important part in interfacing the user interface layer and the application layer. It encapsulates a user input and is passed from the user interface layer to the application layer. `ComCreator` in the user interface layer makes an instance of `Com` and passes it along to `ComResTable` in the application layer.

The Application Layer

This layer requires an object which receives `Com` and creates a command pattern object (`Action` in the figure). The object has to have a possible set of `Com` and `Action`, that is, is implemented as a table (`ComResTable` in the figure.) In fact, the table includes the algorithms for creating an `Action` object rather than simply action itself. We designed it this way because the algorithm for the creation procedure is application dependent and is to be a hot spot of the framework. The strategy pattern is for storing an algorithm in an object. We use the pattern to implement `Strategy` which makes an instance of `Action` from a `Com` object and `StrategyR` in which an `Action` object for handling the server's response is created.

5.2 Implementation of Server's Application Framework

The UML class diagram in Fig.4 shows the server's application framework.

In the case of the server as well as the client, the discussion above leads to the following conclusions:

- The state transition layer requires a mechanism that the command pattern created in the proxy layer at the time when state transition occurs.
- We have to implement an object-oriented application interface for the server database in the application layer.
- The database has to be stored in the database layer.

The Application Layer

`DB_Proxy` is designed to be a hot spot since the way of accessing the database varies widely from one platform to another and from one application to another. The class interface for `DB_Proxy` is also generated when a proxy (a skeleton and a stub) is generated.

The class, `Exclude`, is designed to implement mutual exclusion of server instances which simultaneously try to access the database. It gives an application interface for locking and unlocking the database.

The Database Layer

A server's database is usually a UNIX file. The file is manipulated as an object in our framework (`fstreambase` in the figure.) Standard methods for the file object are provided as frozen spots of the framework.

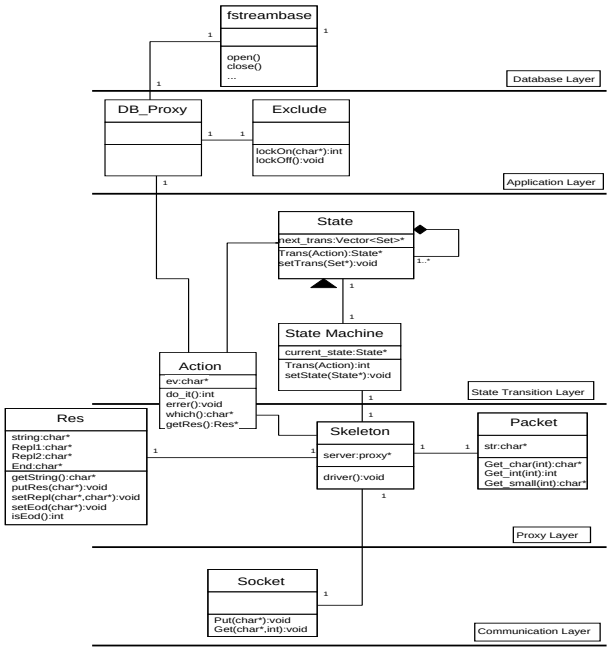

Figure 4. Outline of Application Framework for a Server

5.3 Hotspots

Here we enumerate a set of hot spots which IPaf-R^2 provides for an application programmer. They are class interfaces and all customization is performed by redefining the methods in subclasses.

1. Client-Side
   `ComCreator`, `ComResTable`, `Action`, `Strategy`, `StrategyR`, `State`, `StateMachine`
2. Server-Side
   `Exclude`, `DB_Proxy`, `Action` `State`, `StateMachine`

6. TCP/IP Application Development with IPaf-R^2

Fig.5 shows a software process using IPaf-R^2 (mostly step 2 in the process given in section 3.3.) The figure is a kind of flow chart. Each box shows a subprocess. Concatenation control structure is denoted by an arrow (e.g. selecting a component must be done before writing a specification for a server.) A multiplexed arrow means that subprocesses in its destination boxes are enacted concurrently (e.g. to customize the server database, the `ComCreator`, a user interface, and transition rules are concurrently performed.) An arrow having two source boxes denotes a multiple wait (e.g. customizing `DB_Proxy` has to wait not only for generating an interface of `DB_Proxy` but also for customizing the server database.) Thus, the figure shows a software process in a partial order.

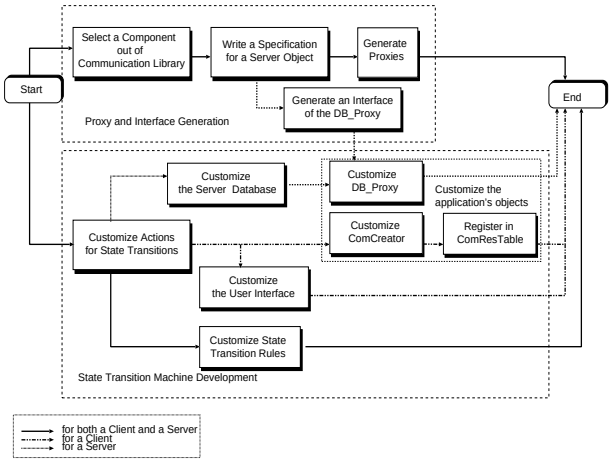

Figure 5. Software Process with IPaf-R^2

6.1 State Transition Machine Development

The step for customizing the framework is divided into the following substeps which are in a partial order.

1. Customize actions for state transitions.
   The class, `Action`, is realized in the command pattern[6] and is a hot spot. `Do_it` method of the class is to be redefined in its subclass.
2. The following steps are enacted concurrently.

– Customize state transition rules.
  The constructor of `StateMachine` is a hot spot and is redefined in its subclass. Instances of `State` are passed as parameters to the constructor.

– Customize the application layer's objects.
  In the case of a client, an algorithm for application logic is encapsulated in `Strategy` class attached to `ComCreator`. The `Strategy` is a hot spot and is to be registered in `ComResTable`.

  The interface of `DB_Proxy` on the server-side is generated in the generation process described below. The class is a hot spot. An implementer or a designer is forced to wait for the interface generation. Actually, this customization must be done after "customizing the server database."

– Customize the user interface.
  The class, `ComCreator`, is tailored to create an appropriate instance of `Com`.

– Customize the server database.
  If the database is a UNIX file, one simply uses the `fstreambase` as a component in a black-box framework. A subclass of `fstreambase` must be defined as a hot spot customization if the database is designed to be one other than a UNIX file.

Table 1. Our Implementation (LOC)

		all	user (spec.)	generated	FS	CL	reusable ((FS+CL)/(all-generated))
BBP	client	1036	260(7)	84	388	311	699(73.4%)
	server	762	168(7)	72	218	311	529(76.7%)
HTTP	client	885	155(5)	36	388	311	699(82.3%)
	server	666	84(5)	58	218	311	529(87.0%)
POP3	client	1071	241(11)	142	388	311	699(75.2%)
	server	1009	379(11)	112	218	311	529(59.0%)
FTP	client	2496	1238(38)	597	388	311	699(36.8%)
	server	1401	587(38)	323	218	311	529(49.1%)

6.2 Proxy and Interface Generation

In the process for proxy and interface generation, the following three steps are enacted sequentially.

1. Select a component out of the communication library.
   Either `TCP_Socket` or `UDP_Socket`, which are subclasses of `Socket`, is selected. `TCP_Socket` realizes a TCP communication, and a UDP communication is implemented in `UDP_Socket`.
2. Write a specification for a server object.
   A specification of a server object has to be written. The specification is an interface (signatures of methods) of the server object.
3. Generation
   The generator makes a stub, a skeleton ("proxies" in the figure), and an object-oriented interface of the server database ("`DB_Proxy`" in the figure.) The generation proceeds customization of the server's application layer.

7. Discussion

In this section, we develop TCP/IP applications using IPaf-R^2, and discuss their usefulness. To check if IPaf-R^2 is helpful for implementing a new protocol, we first tried to implement BBP(BulletinBoard Protocol), an application protocol of our own making. We also tried to implement existing protocols such as HTTP, POP3, and FTP in order to check the compatibility of IPaf-R^2 to the de facto standard and to see if it contributes to productivity compared with traditional development.

The BBP we designed is an application protocol which implements an electronic bulletinboard. A BBP's server is the bulletinboard which holds the text. Clients try to get text on the bulletinboard (server).

Bottom Line of Comparison

Here, we compare applications in two different programming languages: C++ (with IPaf-R^2) and C (existing code). As a guideline to interpret the number of lines, we determined that a lower number means less effort, and therefore higher productivity.

We used the universal function points(UFPs)[8] that one line of C++ code corresponds to 2.2857 lines of C code. Jones *et al.* calculated the value by using the relationship between UFPs and the number of lines. That is, they tried to discover how many lines of code are needed to implement one UFP.

Code Categorization

The code for TCP/IP applications with IPaf-R^2 can be categorized into 1) frozen spots of an application framework, 2) hot spots of an application framework, 3) a specification of a server object, 4) generated code, and 5) a component.

Code must be written for hot spots of an application framework and for a specification of a server object.

The number of lines of the code for BBP, HTTP, POP3 and FTP applications is shown in Table 1. FS stands for Frozen Spot, and CL is for Component Library. Most FS codes are executed at runtime. Only lines of used code are counted and listed in the CL column.

On the other hand, code for TCP/IP applications developed in a procedure-oriented way can be categorized into:

- system-call-related, and
- other.

The system-call-related code includes system calls for TCP or UDP, and its environment setting such as setting up parameters and response processing.

Table 2 shows the number of lines of code for procedure-oriented TCP/IP applications. Since we could not extruct HTTP code from the available Web browser and server, Table 2 does not include the case for HTTP. In C language, we wrote BBP with the same functionality as the one we developed with IPaf-R^2. The POP3 client is the `inc` of `mh` on LINUX. For the POP3 server, we carefully looked at code for Qpopper4.0.3 and extracted code for implementing our POP3 server with IPaf-R^2. The FTP client is the `ftp` on LINUX. The FTP server is wu-ftpd-2.6.1.

Table 2. C Implementation (LOC)

		all	system-call-related(SRC/all)
BBP	client	708	35(5.0%)
	server	453	26(5.7%)
POP3	client	1290	86(6.7%)
	server	1608	98(6.1%)
FTP	client	6265	312(5.0%)
	server	3320	280(8.4%)

Does IPaf-R^2 Ease Implementation of New Application

Table 3. Comparison of LOC

		LOC (IPaf-R^2)	Adjusted (IPaf-R^2)	LOC (C)
BBP	client	260	594.3	673
	server	168	384.0	427
POP3	client	241	550.9	1204
	server	379	866.3	1510
FTP	client	1238	2829.7	5953
	server	587	1341.7	3040

Protocol?

With IPaf-R^2, we have to write 260 lines for a client and 168 lines for a server (see table 3.) These numbers are the total code for hot spot customization and specification to the generator.

In the case of procedure-oriented implementation, we did not count system-call-related code since it corresponds to code for frozen spots in IPaf-R^2 and the component library. The number of lines required is 673 lines for a client, and 426 lines for a server (also see Table 3.)

If we use the value 2.2857 for C++-and-C adjustment, the ratio is 594.3:673 for a client, and 384.0:427 for a server. At least in this one case, our study shows that IPaf-R^2 can indeed ease implementation. However, we realized only about a ten percent gain in productivity since the application size of BBP is too small to show the advantage of our architecture.

Is IPaf-R^2 Good for Implementation of Existing Application Protocols?

For POP3 and FTP code, the same comparison is performed. In Table 3, you can find the ratio 550.9:1204 in POP3 client's row, and 866.3:1510 in POP3 server's. Also, the ratio for a FTP client is 2829.7:5953, and 1341.7:3040 for a FTP server.

Since the number of samples is small (two or three actually), we may not make any broad conclusions about the advantages of IPaf-R^2. However, we may at least say that it is good for implementation. The reason such good results were achieved is that the reusable components in IParch-R^2 are so large.

8. Conclusion

We designed and implemented an application framework for TCP/IP applications IPaf-R^2 grounded upon a domain specific software architecture IParch-R^2 which we constructed. Several TCP/IP application protocols are implemented with IPaf-R^2, and we were able to show that IPaf-R^2 contributes to the higher productivity of TCP/IP applications.

Future research topics on IParch-R^2 and IPaf-R^2 include the following.

- To implement other TCP/IP application protocols.
  From the experience gained through the development of all existing TCP/IP application protocols, such as NFS, We could refine IParch-R^2 and IPaf-R^2. Moreover, we may draw the conclusion that IParch-R^2 enhances productivity through the implementation of such protocols.
- To design and implement a software development environment for IParch-R^2 and with IPaf-R^2.
  Since IPaf-R^2 is a white box application framework, we need more precise and rigorous steps and documents for filling in the box. It implies that a more detailed process view of IParch-R^2 is needed to permit widespread use of the framework.
  In order to improve productivity, a state transition machine could be automatically generated.
- To refine IParch-R^2, to design and to implement IPaf-R^2 for WIS(Web-based Information System).
  We could further refine the user interface layer in order to allow as the design and implementation of WIS. If we find that a Web browser is a component in an application framework of the refined user interface layer, IPaf-R^2 could be a WIS development environment.

9. REFERENCES

[1] S. Böcking, "Socket++ – a Uniform Application Programming Interface for Communication Services," *IEEE Communications Magazine*, Dec. 1996.

[2] S. Böcking, "Object-Oriented Network Protocols," *Proc. Infocom'97*, Apr. 1997.

[3] F. Buschman, *et. al*, *Pattern-Oriented Software Architecture – A System of Patterns*, Wiley, 1996.

[4] Object Management Group, *CORBA: The Common Object Request Broker: Architecture and Specification.* 1994.

[5] D. Garlan and D. E. Perry Eds., "IEEE Transaction on Software Enginnering, Vol. 21, No. 4," 1995.

[6] Erich Gamma, *et. al*, *Design Patterns*, Addison Wesley, 1995.

[7] H. Huni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," *Proc. OOPSLA'95*, pp.358-369, Oct. 1995.

[8] C. Jones, "Table of Programming Languages and Levels, Version 8," Software Productivity Research white paper (Burlington, Massachusetts), June 1995.

[9] P. B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, Vol. 12, No. 6, pp.42-50, Nov. 1995.

[10] M. Noro, "A Software Architecture for Vending Machine Control," *Proceedings of ISFST '97*, Nov. 1997.

[11] M. Noro and K. Goto, "An architecture and a Framework for IP Applications," *Proceedings of APSEC97*, Dec. 1997.

[12] M. Shaw, *Software Architecture, Perspective on an Emerging Discipline*, Prentice Hall, 1996.