DESIGN AND IMPLEMENTATION OF META-PROTOCOL FRAMEWORK FOR DYNAMIC COMMUNICATION PROTOCOLS SPECIFICATIONS

S. Aljahdali

College of Computers and Information Systems,Taif University,
Taif, Saudi Arabia
aljahdali@tu.edu.sa

Abstract: *A communication protocol is an agreement among two or more parties on the sequence of operations and the format of messages to be exchanged. Standardization organizations define protocols in the form of recommendations (e.g., RFC) written in technical English, which requires a manual translation of the specification into the protocol implementation. This human translation is error-prone due in part to the ambiguities of natural language and in part due to the complexity of some protocols. To mitigate these problems, we divided the expression of a protocol specification into two parts. A high level specification of the protocol, expressed as a Finite State Machines (FSM), is written in XML and is based on Component Based Software Engineering (CBSE). Then, the components required by the protocol are specified in any suitable technical language (formal or informal). We developed a framework that provides a higher level of flexibility for the implementation and distribution of protocols than that of traditional autonomous protocols. Also, we built a system architecture and code generator to test the proposed Meta-protocol framework.*

Keywords*: Security protocols, XML, CBSE, FSM, Communication Protocols.*

1. Introduction

Communicating parties in computer communication systems must share a common protocol. A protocol is a set of rules shared by two or more communicating parties to facilitate their data communication. These rules have two parts: 1) syntax, i.e., the format of the messages to be exchanged and 2) semantic, i.e., the sequence of operations to be performed by each party when events (e.g., reception of messages, timeouts) occur. The traditional process of implementing a protocol may be tedious and error prone. Protocol specifications in natural language are often ambiguous and may lead to defective implementations. Moreover, implementations have to be thoroughly tested, a time consuming effort due to the timing dependencies of events processed by protocols. The problem addressed in this paper deals with communication protocols in general and with security protocols as a special case. For security protocols, the current approach to implementing protocols is centered on designing a protocol as a single package comprised of two layers: control and a library of algorithms. This approach is based on the assumption that every protocol is complete and does not need to be integrated with other security protocols. This assumption may not be valid in situations where several security protocols need to coexist. In such cases, redundancy and conflicts may occur. The goal of this paper to propose a framework that makes protocols free from being dependent on the autonomous implementation of a limited number of capabilities. Communication protocols in general, as well as security protocols, need a higher level of flexibility to handle evolution. The proposed approach increases the level of code sharing and reusability.

The previous and related works in the area of communication protocols is found in several research projects: x-kernel [1-6] and the Click router [7]. The x-kernel is an operating system communication kernel designed to provide configurable communication services in which a communication protocol represents a unit of composition [1-2], [8-10] are extensions of the x-kernel architecture. On the other hand, Horus, and Coyote are extensions and applications of the x—kernel architecture in the area of group communications [8], [11-12], [4-5]. There are four differences between my approach and the research work just discussed. The first distinction is in the application domain. Prior research work focuses on communication protocols without any consideration for security protocols. My work is applicable to communication protocols in general but is more suitable for security protocols. Second, in the case of the other projects mentioned above, the configuration information of a newly designed protocol has to be shared manually among the parties involved. I provide support for online negotiation and distribution of protocol specifications. Third, those related works do not provide a precise methodology, as I do, to express the specification of a protocol configuration. Finally, the proposed solutions are either associated to a single layer of the network stack as in [13], or they replace the entire network stack as in [9, 2, 8, 14]. The rest of the paper is organized in the following manner. In section 2, the proposed Meta-Protocol Framework in this paper is described. In section 3, we provide architecture for meta-Protocol Framework implementation. Finally, section 4, concludes the paper and pointes for future research.

2. Proposed Meta-Protocol Framework

The Meta-Protocol provides a higher level of flexibility for selecting a protocol that fits the user's exact needs. The framework consists of four layers: protocol negotiation, protocol specification and component distribution, protocol automatic implementation, and protocol execution. Figure 1 shows the layers of the framework.

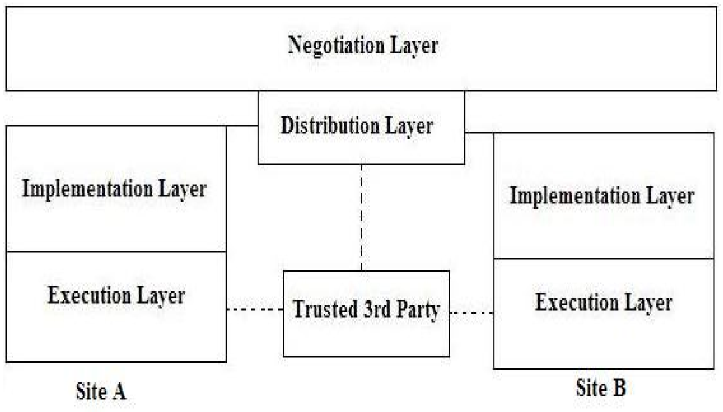


Figure 1: The four layers of the Meta-Protocol framework.

The Meta-Protocol framework delivers a proposal of a protocol XPSL specification, retrieves the document that contains the specification and any missing components, generates executable code, and runs the protocol. In addition to that, the framework has to be immune against attacks such as identity theft, Denial of Service (DoS), anti-replay, and connection hijacking. Thus, the framework requires:

1. A generic negotiation mechanism to determine a protocol specification.
2. A delivery mechanism for retrieving the protocol specification and any missing components.
3. A repository system to store specifications and components.

4. A mechanism to transform XPSL protocol specification to executable code.
5. A system that manages executable code by loading it and running it as needed.
6. Security measures against expected forms of attack at each part of the framework.
7. Maintaining a layered approach.

The layered approach shown in Figure 1 is found to be necessary in this framework to maintain flexibility.

2.1 The Proposed Framework Layers

The design principle behind the Meta-Protocol framework lies on the separation between a protocol specification and its implementation. A protocol specification according to this framework is a high-level abstraction based on a Finite State Machine (FSM) and is expressed in XPSL. This approach achieves two goals: machine-readable protocol specifications can be easily transmitted to the interested party and an implementation of the protocol can be automatically generated from the specification. Our meta-protocol consists of four layers that can be stated as follows:

2.1.1 The negotiation layer

The goal of this layer is to establish an agreement between two parties on the specification of a communication protocol. The two parties have to know the name and the location of that document. The initiator of the communication is responsible for providing a proposed set of protocol specifications. The responder is required to choose his preference and notify the initiator. The goal of this layer can be achieved in several ways: manual, secure channel (e.g., SSL or IPsec), or using ISAKMP. The manual approach may be reasonable for local areas but not for long distances. On the other hand, using secure channels complicates the approach because secure channels do not provide the required negotiation mechanism. Therefore, ISAKMP has been chosen for this layer because it provides a negotiation mechanism in addition to several important security features.

Figure 2 shows an example of an ISAKMP packet carrying a protocol proposal. In an ISAKMP exchange, such a packet is sent by an initiator to deliver a proposal. A proposal defines a protocol specification name, its location, and version. The protocol specification name and location are the only mandatory pieces of information in a proposal. The protocol version is optional. In this example, the protocol specification name (Pr0t0c0l_X) and versions (MjVer and MinVer) are placed in the first payload. The location (http://www.xyz.com/dirl) is placed in the second payload. An initiator may send multiple proposals linked together to a main ISAKMP header using the Next Payload field. The list is terminated by a null value in the Next Payload field. The initiator also has the option to append a key exchange payload to the packet. Key exchange payloads carry information about the proposed key exchange technique (e.g., Oakley or Diffie-Hellman) [15]. The key exchange also carries the required data to generate session keys. Additional information on the format of the payload is available in [16].

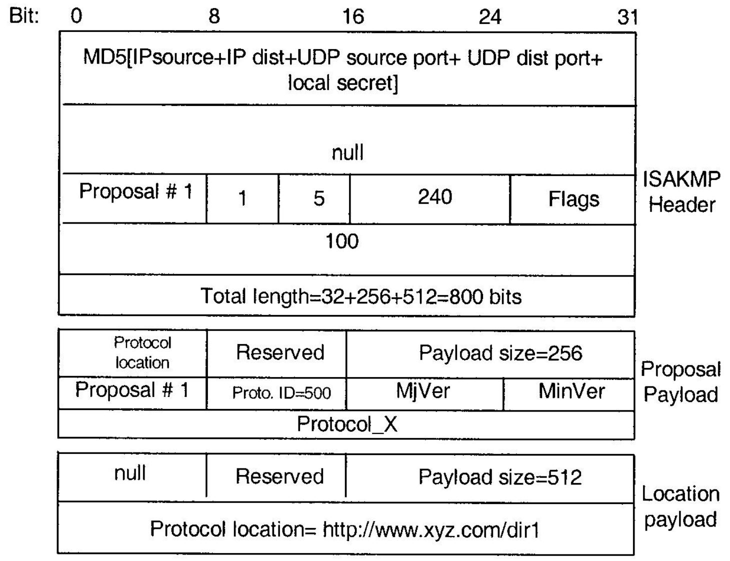


Figure 2: Example of an ISAKMP header carrying a proposal payload

2.1.2 The Distribution Layer

The set of specifications and components in the framework does not need to be stored locally. Communicating parties can exchange specification or component’s code directly or through a trusted third party. To save space and reduce duplications, specifications and components that are needed for short period of time can be deleted from the system after their use. To protect these specification documents and components from malicious alteration they have to be signed by their authors and verified by the receiving party before being added to local systems. Trusted providers may place their specifications and components at public repositories on the Internet so users may retrieve them whenever they need. Newer versions may also be pushed from those providers to subscribed users to keep their systems up to date.

The distribution layer uses UDDI, which is basically a directory service that provides advertising and discovery services. Each service advertised in a UDDI directory has the freedom to specify the transport mechanism to access that service (eg., SMTP, FTP, HTTP, and SOAP). Every UDDI user checks the transport mechanism of the service he or she is looking for before connecting to that service. Communicating parties in the Meta-Protocol framework use UDDI to advertise a protocol names, the location of their specifications, the location of the components needed by the specification, and the preferred transport mechanism to download the specification and its components. Using UDDI at the distribution layer also serves those users who are planning to start a negotiation but do not have specifications to propose to the other party. These users can consult UDDI directories to discover protocol specifications, their providers, the location of the specifications, and their components for download.

2.1.3 The Implementation layer

In this layer, I developed a new model to produce protocol implementations automatically through protocol specifications. The proposed approach is based on eXtensible Markup Language (XML), eXtensible Style-sheet Language for Transformations (XSLT), and on CBSE. XPSL is used to specify protocols described through Finite State Machines (FSM). XSLT is used to transform the specification description into actual code. XSLT style-sheets can be designed to produce code in different programming languages (e.g., C++ or Java). CBSE is used to build the set of operations needed by the protocol. Figure 3 shows the proposed automated protocol production process [17]. Every component used in an XPSL specification is an executable program that is designed based on CBSE principles. A component may be shared by more than one XPSL protocol specification, For example, an RSA encryption algorithm is a single component. Such a component may be used in a single protocol several times as well as shared by many protocols. On the other hand, a component may also comprise several subcomponents. For instance, a security envelope component consists of a header processing subcomponent, a MAC subcomponent, and an encryption algorithm subcomponent.

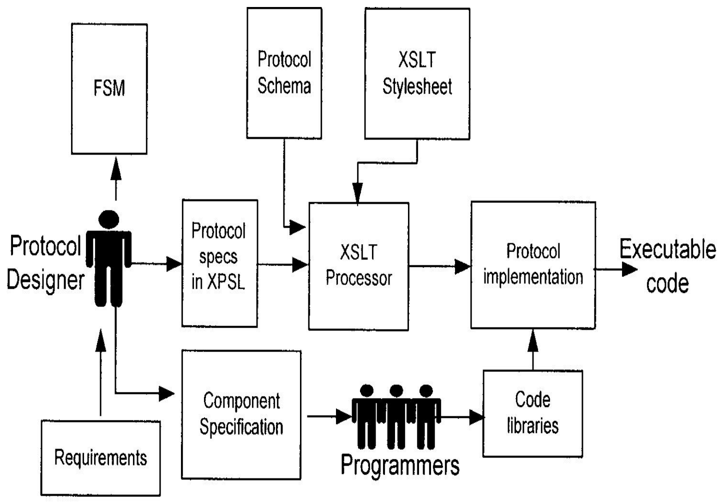


Figure 3: Proposed automated protocol production process

2.1.4 The Execution layer

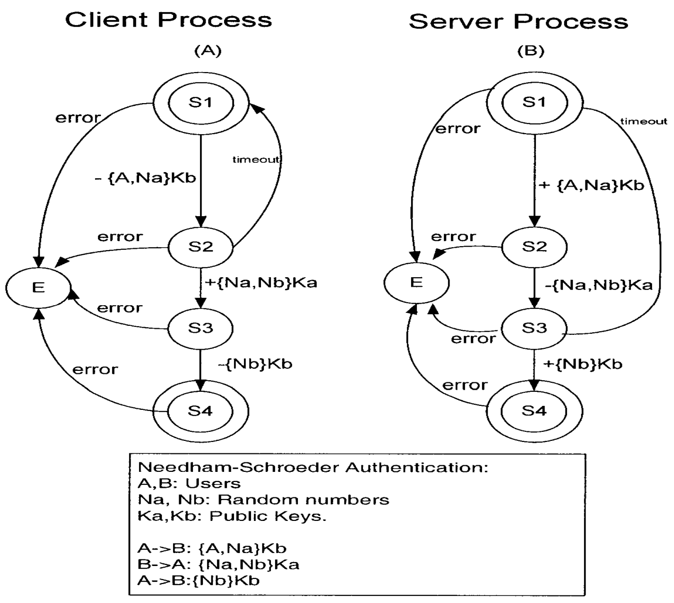
This layer deals with the executable code of a protocol and depends on the services provided by the underlying operating system, and the network. All kinds of message exchanges between parties are performed by the code running at this layer. Protocol code must be loaded as a privileged process by the operating system. This process and its address space must also be protected against any malicious alteration or termination. The execution layer plays a central role in the framework because it integrates all the different components of the framework; the execution layer consists of a set of interfaces and control operations. The interfaces are responsible for delivering requests and feedback between the external world and the execution layer, to, for example, deliver user requests to start a protocol or to terminate it. Control operations manage the internal affairs of the execution layer. Examples include authenticating a user to access a protocol, checking the availability of a protocol, loading or terminating a protocol [18].

2.2 Automatic code generation

Automatic code generation is not an easy process. It requires enough knowledge about the domain of the problem, enough knowledge about the detailed design, and enough knowledge about the implementation [19]. My approach was successful because I limited the domain of the problem to the area of communication protocols. In addition to that, I separated the implementation from the design of the protocol to further reduce the complexity of the problem.

2.3 Use of FSMs in protocol design

Finite state machines (FSM) are commonly used for describing protocols. The graphical representation of an FSM is a visual aid to the designer of the protocol. This representation helps in testing and verifying the correctness of the design. An FSM consists of a set of states (represented by circles) and a set of transitions between these states (represented by directed arrows connecting the states). At each state, a set of events may occur. Each event triggers a specific transition out of a state. Before a transition occurs, a set of actions (possibly an empty set) is taken. One of the states in an FSM is designated the initial state. Figure 4 shows an FSM for the Needham-Schroeder authentication protocol (NSAP), a two- way authentication protocol based on public key cryptography. A client starts the process by using the public key of a server to encrypt his identity and a random number; Na, The server receives the message, generates a random number Nb, encrypts the concatenation of Na and Nb using the client's public key, and sends the message back to the client. The authenticity of the server is verified if the client successfully retrieves his Na. Then, the client returns Nb encrypted with the server's public key to prove his identity. Also Figure 4 shows the steps of the protocol.



Needham-Schroeder Authentication:
A,B: Users
Na, Nb: Random numbers
Ka,Kb: Public Keys.

A->B: {A,Na}Kb
B->A: {Na,Nb}Ka
A->B:{Nb}Kb

Figure 4: Proposed automated protocol production process

2.4 Approaches to Protocol Specification

An essential concept in my work is the use of a domain-specific specification language in lieu of technical natural language for the specification of protocols, as in IETF RFCs. This approach facilitates automatic protocol implementation from specification. The process passes through three stages: design, verification, and implementation. The first two stages produce a specification document. Standard protocols (e.g., TCP/IP, FTP, SSL, and IPsec) specifications are written in natural English (e.g., RFC). Programmers must translate these documents into code. In my approach the manual design and verification process is almost the same as in the traditional approach. However, the design process uses FSMs to produce protocol specifications in XPSL. These specification documents have to be well—formed XML documents and have to comply with the rules specified in the protocol XPSL schema. If both of these conditions are satisfied, the XSLT stylesheet automatically produces implementation for the protocol without human intervention. Many different implementations in several programming languages can be developed for the same protocol specification.

2.5 XPSL Protocol Specification

The process of writing down protocol specifications follows the sequence:

1. Place the <protocol> element as a root of the XML protocol specification document.
2. Add a <name> element as a child of the <protocol> element to specify the name of the protocol
3. If needed, add a group of elements that prepare the environment (e.g., define public objects as <objects> or <instances>)
4. After that, add the <first-state> element to specify the starting state of the FSM.
5. For each state in the FSM insert the <state> element.
    5.1. Inside each <state> element, add a group of <action> elements to perform the actions required by the state.
    5.2. At the end of each state, add a <moveto> element to indicate the next state.

2.5.1 The <Protoc0I> Element

This is the root element of the XPSL document. It is a mandatory element according to the XML specification. This element has an optional child element <name>, which can be used to identify the protocol; the XSLT processor can ignore or replace it with any other user-supplied identifier. The root element is also the parent element of the following elements: <object>, <instance>, <first-state>, <action>, and <state>. The first two elements have a global scope and are used to prepare objects shared by the states of the protocol.

2.5.2 The <first-state> Element

The first-state, used to identify the starting state, is a single occurrence. All other elements may have multiple occurrences. The reason for making the rule of having single occurrences for the first-state element is because any protocol only has a single point of entry for execution. The first state element is used to identify that starting state. This element is mandatory; otherwise the design would be ambiguous. Often, the starting state is the state that produces the "Hello" message to start a communication.

2.5.3 The <Object> Element

The object element is used to define data needed during protocol operations such as: messages, keys, constants, and random numbers. Every object consists of a name and a set of fields. Every field has a type attribute associated with its name type. Data types are predefined types, such as: Keys, Strings, Integers, and Booleans. As needed in the operations, these objects are transferred from one action to another as parameter passing. In the example provided here, an object called Sessi0nState is used to hold shared information between states. In other words, there is no public information unless it is defined as an object and passed as a parameter to the actions of a state.

2.5.4 The <Instance> Element

The instance element is similar to the concept of instance in object- oriented programming; objects represent definitions of classes and instances are the actual realization of the definition. An object may have several instances, as needed, but every instance represents one object and there is no shared information among instances. For example, an object called error-message can be used to create several instances of error messages, as needed in the protocol states. Every error message instance can hold different error codes and an identification of the component that generated the message.

2.5.5 The <moveto> Element

This element is similar to a switch—case statement in a procedural programming language. This element provides the mechanism that controls the transitions from one state to another. The <m0vet0> element consists of a logical expression and a list of cases. The logical expression is evaluated to produce a set of predefined values. Corresponding to each value, there is a case element to point to the next state. A <moveto> element is either a child of the <state> element or of an <action> element. Being a child of an <action> element gives the flexibility for early branching out of the state before finishing all the actions listed in the state. This feature is useful in case of errors or timeouts raised by some actions. However, when a <moveto> is used as a child of a state, it has to be the last element in the state after all the actions.

2.5.6 The <actl0n> Element

The action element is used to define the actions that take place inside a state. Typically, actions are calls to components in a library. ln some situations, elements such as instances and <moveto> may be nested inside an action element. These situations occur when a component needs an instantiation of a data object that was not passed to the state for temporary local usage. The <moveto> element may be nested inside an action in situations such as branching out of the state before completing all the actions in a state due to errors.

2.5.7 The <state> Element

The state element is used to describe the different states of a protocol. A <state> element is not allowed to be nested inside another <state> element and every state in the FSM has to be described by exactly one <state> element in the XPSL specification document. Each <state> element has an optional child element called <arg>. The <arg> element describes the external objects that are passed to the state. The <urg> element requires two attributes: the name of the object to be passed to the state and the type of the object. A <state> elements can have one of the following child elements: <object>, <instance>, <action>, and <moveto>. The purpose of these elements is to express the operations that are performed inside every state of the FSM. Typically, the branching out of the state is performed as the last action in a state. However, in case of errors, branching may take place in the middle of the state. Developers should anticipate such cases and place <moveto> elements after the actions that raised such errors.

2.5.8 Protocol Data Types

Every programming language follows a specific approach to defining the syntax and semantics for data types. An XPSL needs to adopt the minimum common denominator to make up a unified approach to serve the most popular programming languages. This need has been addressed by two means. First, introducing a syntax and semantic for six primitive data types that are common in programming languages. Table 1 shows a comparison of data types between several programming languages. Second, the adoption of an objected—oriented approach provides flexibility in defining new data types for additional needs of an XPSL protocol specification.

Table 1: Data type comparison between programming languages

Data Type	Java (Bits)	C (Bits)	C++ (Bits)	IDL (Bits)
Byte	8	8	8	N/A
Short int	16	16	16	16
unsigned short int	N/A	16	N/A	16
unsigned int	N/A	32	N/A	N/A
int	32	32	32	N/A
long int	64	32	64	32
float	32	32	32	32
double	64	64	64	64
long double	N/A	96	128	N/A
Char	16	8	8	8
String	Variable	Variable	Variable	Variable
Boolean	1	1	1	Unspecified

2.6 Code Generation

I used XSLT to transform protocol specifications, given as an XPSL document, into implementation code. I developed an XSLT style-sheet to produce Java code for any protocol. Every element in the XPSL document causes the XSLT processor to produce the required code. In my experiments, I used Sun’s XML Pack, which includes an XSLT transformer and XML Schema checker. XSLT rules and filters are used to identify different XML elements, convert them to their correspondent program code, and place them in the proper XSLT output tree. Every programming language requires the design of its own XSLT transformation sheet. Once that sheet is available, any protocol represented as an XPSL document can be transformed to that language and compiled at the appropriate system. Figure 5 shows the steps required to generate a protocol implementation from an XPSL specification. As a result, any system on the Internet will be able to automatically generate protocol code as and when needed. Every system that has the proper XSLT style-sheet will be able to download the XPSL specification and use the style-sheet to produce the code in the preferred programming language, compile it, and run it. The Java XSLT style-sheet and the XPSL specification of the NSAP are available at appendices B and C, respectively, in this XSLT style sheet, 16 templates were used to make the transformation from XPSL specification to Java code. The number of templates depends on the target language for the implementation. Each XML element in the specification relates to one or more template to produce code. The following sections describe how these 16 templates produce protocol implementation. Testing the correctness of this style-sheet is similar to testing any software product. This style-sheet was successfully used on several correct protocol specifications. The generated code was successfully compiled and thoroughly tested.

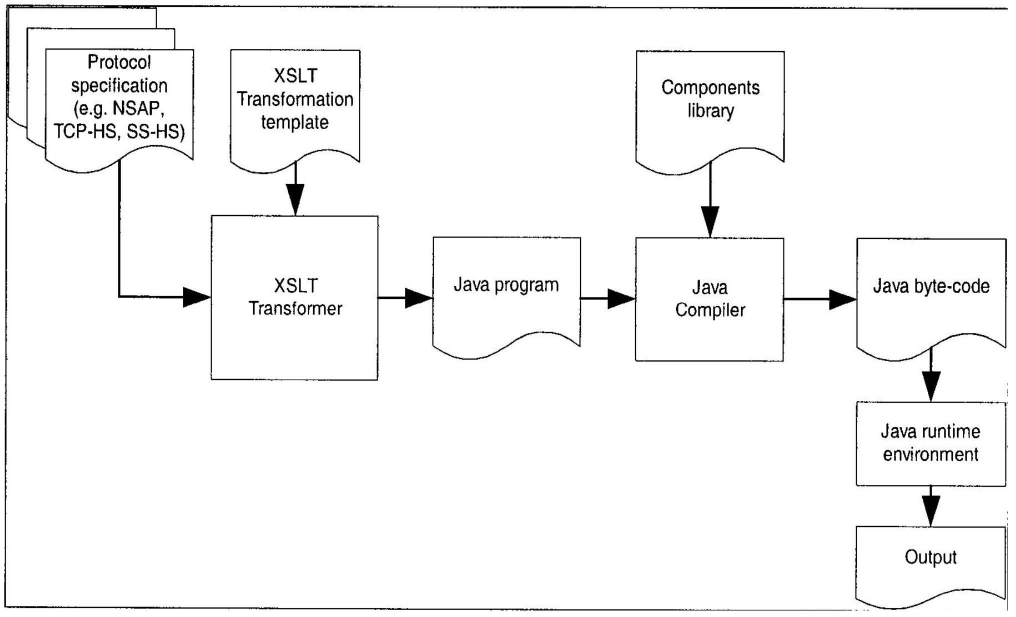


Figure 5: Steps to generate protocol implementation from specification

2.6.1 The style-sheet structure

Producing Java code requires writing a main program and a set of classes. The main program is responsible for starting the execution. Classes are either objects or states. Therefore, the first few lines in the style-sheet describe the XML version, the style-sheet version, the XSL name space, and the type of output. Then, the main template determines the structure of the style-sheet. This main template pushes to the output a header for the main Java program and then calls five other templates: /protocol/name, action, first-state, object, and state template. Notice that Java statements are embedded inside the template as needed. The /protocol/name template is responsible for producing Java code for the identifier of the public class of the program which comprises all other classes.

2.6.2 The first-state template

The first state template is a trivial template to produce the name of the entry point of the state machine. This template is called once from the main template of the style-sheet. Therefore, the main Java program only has one class to call, which is determined by this template.

2.6.3 The object template

This template produces code for data objects. This template produces a Java class that defines an object interface and a construct for the instantiation of the data members. This template may be called from the main template or from any state. If it is called from the main template, the object becomes a public object. However, if it is called from within a state, it becomes local to that state.

2.6.4 The instance template

In this template, the transformer pushes Java code that corresponds to instantiating an object. The data object name, type, and initial values are picked up from the XPSL protocol specification and arranged into the appropriate Java code.

2.6.5 The moveto template

This template is responsible for producing a Java switch statement for branching. This template calls two other templates: (1) the test part of the Java switch statement, and (2) the expected cases as presented in the protocol specification.

2.6.6 The action template

The goal of the action template is to push out the interface statement of a component as listed in the protocol specification. However, since this element may be nested with other elements, the template has to test for three cases: instance, moveto, or object. If none of these cases exist, the default action is to list the interface of the component.

2.6.7 The state template

The state template is similar to the main template that has been described earlier. Every state template represents a state of the FSM. Every state consists of a set of objects, instances, actions, and should end up with a <moveto> statement. The state template produces a Java class that corresponds to an FSM state. Typically, the <moveto> statement comes toward the end of the state to determine the next state. However, in some cases (e.g., errors) control flow may branch earlier. Such cases should be expected by the developer of the protocol specification and therefore the <moveto> statement should be embedded inside an action statement. Otherwise, the sequence of operations will be faulty.

3. Architecture for Framework Implementation

In order to implement the Meta-Protocol framework I need two environments: development and execution. The development environment is responsible for writing down protocol XPSL specifications from protocol designs, developing components, and verifying specifications and components. Once a specification is complete along with all the required components, the protocol is deployed into the execution environment. The development environment architecture and tools are widely open and they vary from a simple standalone editor and compiler to complex visual (e.g., GUI) systems that integrate editing, coding, and verification. I chose not to propose specific system architecture for the development environment because it limits the developer choices. The distinction between the two environments is important from the operational and performance points of view. The objective of the development environment is to prepare the system and produce the necessary code. This can be done in the background; i.e., under no time constraints. It involves human being interaction and may take several cycles to produce the correct XPSL specification and components code. On the other hand, the operations in the execution environment take place at run time. Therefore, by separating these two environments, protocol code is executed with minimum possible overhead.

3.1 Protocol Control System

In the execution environment, the Protocol Control System (PCS) controls the internal activities of the system and provides several interfaces to connect the layers of the framework to the external world. Figure 6 shows my proposed system architecture for the implementation of the Meta Protocol framework. The rest of the chapter describes the components of the PCS and how it operates.

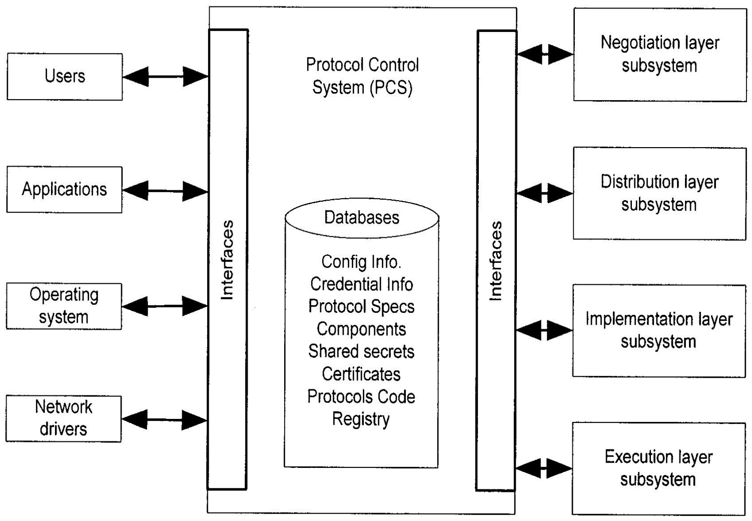


Figure 6: System Architecture

3.2 Components of the PCS

The PCS consists of a controller, cache memory, code generator, a set of interfaces, a set of databases (including protocol specifications and components), and a collection of components. All these components are protected by the operating system. There are two types of allowable access to the PCS: administrative access and user access. Users access the PCS to request operations such as execution of a protocol or adding a protocol specification or component. Administrative access is granted to a few legitimate users for administrative and maintenance tasks such as registering users, registering a trusted site, and changing the configuration of the system (e.g., audit policies). In highly secure environments, administrators may also be classified into several levels according to the environment requirements such as: supervisor, backup only, and users' management administrator.

- **The Controller Subsystem:** The controller is the core of the PCS. It is responsible for the coordination of the internal activities of the system. To maintain the integrity of the system, all type of accesses to the internal subsystems should be limited to the controller. The controller is responsible for receiving protocol invocation requests, verification of user's access privileges, configuration management, storage of specification and components, protocol code creation or removal, loading or terminating the execution of a protocol, and auditing operations. Requests for system services may come to the PCS from: a user, the operating system, an application, or a network [20, 21]. The controller is also responsible for the authentication of users to access protocols or to modify databases (e.g., for registering, adding or deleting a specification or component). To achieve this task the controller needs to consult internal or external user directory services and access lists determined by the PCS configuration. Management of the configuration information is also part of the responsibility of the controller. Only a supervisor administrator is allowed to change the configuration information. Configuration information includes: time synchronization source, host identity and address, administrator contacts, maximum disk quota allowed for the various system databases, audit policies, and access information to external databases (e. g., locations, user accounts, and access methods). Audit and I/O management are two important roles of the controller. Audit logs are very important for problem analysis and security investigations. I/O services include handling and protecting inter-process communication and the communication with network interfaces. The interfaces with the execution layer may also maintain their own logs of activities.

- **The Cache Memory Subsystem:** The cache memory is a temporary memory used by the PCS to improve the efficiency of the system. Copies of frequently used protocols, components, or security related information (e.g. keys and passwords) are temporarily stored in this cache to reduce the preparation time and to speed up the execution of protocols.

- **The Code Generator Subsystem:** The code generator consists of two processors: the XSLT processor and a compiler. The code generator acts on behalf of the implementation layer. The XSLT processor takes as input an XSLT style-sheet for a specific programming language and an XPSL protocol specification and produces source code in the language specified in the XSLT style-sheet. Then, the complier converts the source code into executable code for the protocol.

- **The Interfaces Subsystem:** The interfaces subsystem is responsible for facilitating the communication between the external world and the PCS. Several types of interfaces are needed: user, application, operating system, and network. Breaking down the communication of the PCS with the external world into specific interfaces improves evolution and the scalability of the system. Evolution is improved by the ease of replacing an old version of an interface with a newer one. Scalability is improved by adding new interfaces as needed (e.g., when a new network driver appears). The difference between the user interface and the application interface is that the first is used for the manual access to the system and the latter for automated access. For example, a system administrator who wants to make changes in the system needs a user interface. However, an application that needs to query the system or invoke the execution of a protocol requires an application interface. The operating system interface is responsible for coordinating the execution of protocols with the operating system. The network interface is responsible for remote management of the PCS and for retrieving specifications and components from remote sites.

The Database Subsystem: The database subsystem manages information needed for the operation of the PCS such as: users' accounts, access list, secret keys, certificates, XPSL protocol specifications, components, administrative accounts, trusted providers, and the registry. To save system resource, some of these databases may be maintained externally (e.g., LDAP, UDDI, iPlant, and Active Directories). Thus, the PCS consults these directories and retrieves the information as needed. The choice of an external or internal database is part of the configuration process of the PCS and such information is saved as part of the configuration information. Most of the databases mentioned earlier consist of standard technologies used in many applications. Therefore, I focus only on two databases that are specific to the operation of the PCS: the access list and the registry. The access list defines the access permission that each user has to each object (XPSL specification or component) in the system. Each user has four types of permissions: read (R), write (W), execute (E), and delete (D). A user may also be assigned to a combination of these permissions. Table 2 shows an example of an access list. The '*' in the last row of the table stands for "others". A value of "1" in the table indicates permission and value of "0" indicates denial. Figure 7 shows a UML entity class diagram for the access list. The second database is the registry. This database is used to register available objects(specifications or components), their location, provider information, authentication information, and expiration date. This registry can be updated in two ways: automatic or manual. The automatic update is limited t0 adding new specifications or components through the on-the-fly process of exchanging protocol specifications. Deleting an object from the registry is limited to legitimate users (e.g., Administrator). Legitimate users are allowed to access this registry manually and update it. Table 3 shows an example of the registry database. A registry record consists of an object id, object location, identity of the user who registered the object, identity of the provider who created the object, a signature value (e. g., in binary format), and an expiration date and time for the use of the object. Figure 8 shows a UML entity class diagram for the registry database.

Table 2: Example of an access list

User id	Object id	Permissions			
		R	W	E	D
Mohammed	**DESEncrypt**	**1**	**1**	**1**	**0**
Sultan	**Protocol_X**	**0**	**0**	**1**	**0**
Fahed	**Protocol_Spec_Y**	**1**	**0**	**1**	**1**
*****	*****	**1**	**0**	**1**	**0**

- **Protocol Components:** Every component is an executable program that is designed based on CBSE principles. A component can be designed out of a simple operation (eg., encryption) or it could comprise several components as a compound component (e.g., security envelope). A component may be shared by more than one XPSL protocol specification. For example, an RSA encryption algorithm is a single component. Such a component may be used in a single protocol several times as well as shared by many protocols. On the other hand, a component may also comprise several subcomponents. For instance, a security envelope component consists of a header processing subcomponent, a MAC subcomponent, and an encryption algorithm subcomponent.

The set of all needed components does not need to be stored in every PCS system. Communicating parties can download the code of a component from one system to another if the code is needed but missing from one of the systems. Moreover, components that are needed for a short time period or by a certain process can be deleted from the system after their use. Moreover, upgrading a component is easier than upgrading a version of a protocol in the traditional sense. This same concept also applies to the XPSL specification of protocols, which could be downloaded from one user to another or from a trusted center to the user for temporary usage or for an upgrade. To protect these components from malicious alteration they have to be signed by their authors and checked by the PCS system before being added to the system. Trusted component providers may place them in public repositories on the Internet so that users may pull them when needed. Newer versions may also be pushed from trusted providers to subscribed users to keep their systems up to date. Every component has an interface that specifies its parameters and their types. This interface should be published with a description of the component operations so that protocol designers can understand it and use it. Developing interfaces and defining conventions for components names and functions is, however, essential for compatibility and interoperability among components in the framework. This can be achieved in two ways: either by standards development committees or by the fact that a component becomes widely available because a major vendor with global reach supports it. Data objects are also considered components in the system. Such data objects are used during the process of writing of protocol specifications. Such data objects are often used to hold values that are common among several components such as (time, security keys, sequence and random numbers, and header information).

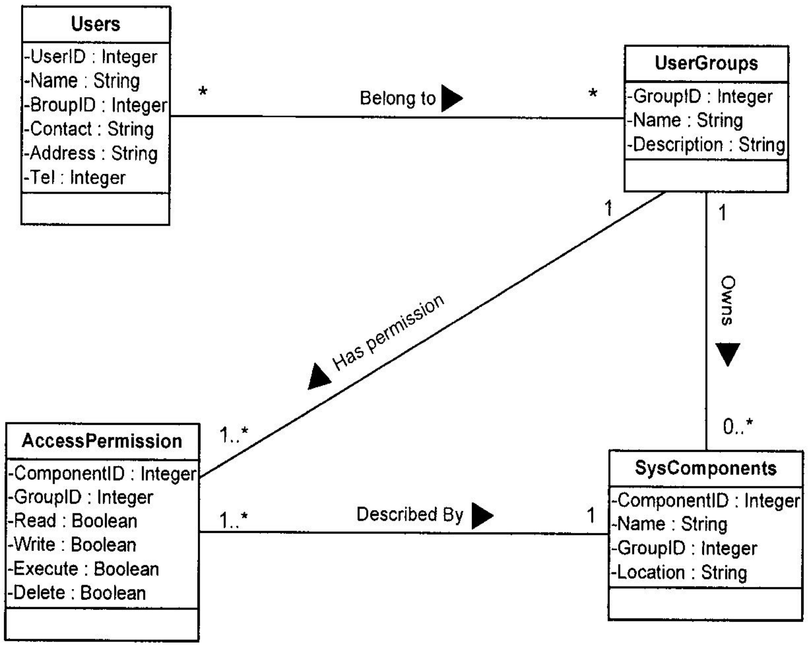


Figure 7: UML entity class diagram for the access list

Table 3: Example of records in the registry database

Object id	Location	Registered by	Provider	Signature	Expiration
Protocol_X	**C:\PCSObjects**	**Admin1**	**Company A**	**010100101**	**9:00-10-6-2008**
Components	**http:\\www.xyz.com\comps**	**Admin1**	**Company B**	**000101010**	**Never**

3.3 Operation of the PCS

The objective of the operation of the PCS is to integrate the layers of the Meta-Protocol framework. Each layer of the framework is implemented through a call for an operation as follow:

1. A call for negotiation operation
2. Calls for retrieving a specification or a component
3. A call for generating protocol code (XSLT processing and compilation)
4. A call for execution.

Figure 9 shows a UML activity diagram for the operation of the PCS. This diagram describes how the four layers of the framework are integrated through the implementation of the above mentioned four calls. A typical scenario for the operation of the PCS starts with a request to run a protocol. The PCS checks locally for the protocol code. If that code is found, the PCS sends that code for execution. If that code is not found, the PCS looks for a specification, transforms the specification into code, and sends it for execution. Otherwise, the PCS uses the location information provided in the request to retrieve the specification. Once the specification is downloaded, the PCS transforms the specification into code and sends the code for execution. The transformation of protocol specifications to code requires the availability of the code of the components mentioned in the specification. If there are missing components, the PCS has to retrieve them from remote locations according to its internal operational policies. The minimum requirements for any user of the framework are to have ISAKMP specification, its components, and an XML transformation stylesheet. ISAKMP is considered the bootstrap of the Meta-Protocol. Communicating parties are required to generate code for ISAKMP to be able to agree upon the subsequent protocol they plan to use. Once that agreement is established, the parties proceed to download the specification of the agreed upon protocol, generate the code, and start communication by running the generated code.

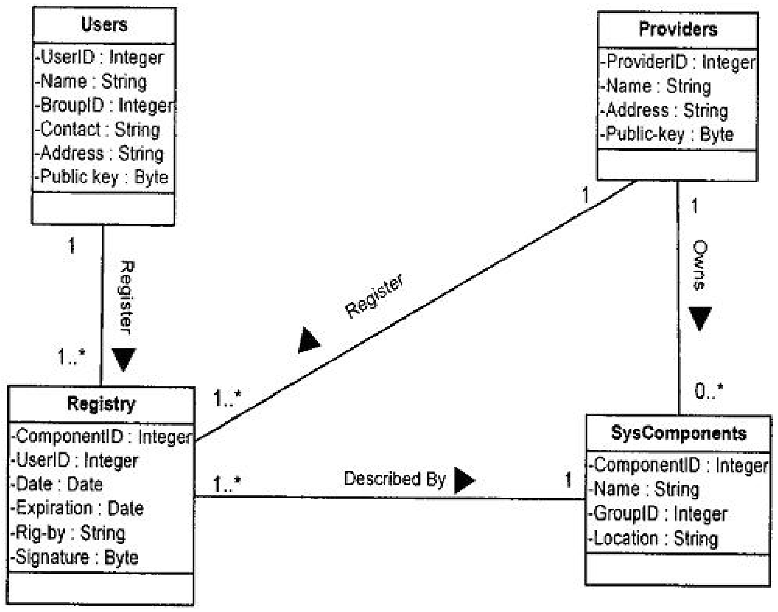


Figure 8: UML entity class diagram for the registry database

3.4 Performance

However, performance is also a concern. While actual performance depends on the implementation of the framework, some architectural issues may be relevant to performance. In general, I believe that the development process of components and protocols should generate optimized code that has reasonable performance compared to traditional protocols. For example, if a specification specifies the services of an SSL protocol, the Meta-Protocol should be able to generate an implementation of SSL with similar performance characteristics as an implementation generated by traditional means.

The code generated by the PCS could be cached in memory for reuse by similar calls to avoid the overhead of regenerating it. However, if the specification requested services that could not be achieved by any other traditional security protocol, I have achieved something that was not previously possible and the performance may or may not be degraded. In this case, the performance degradation may be justifiable because it is for a new service that was not previously possible.

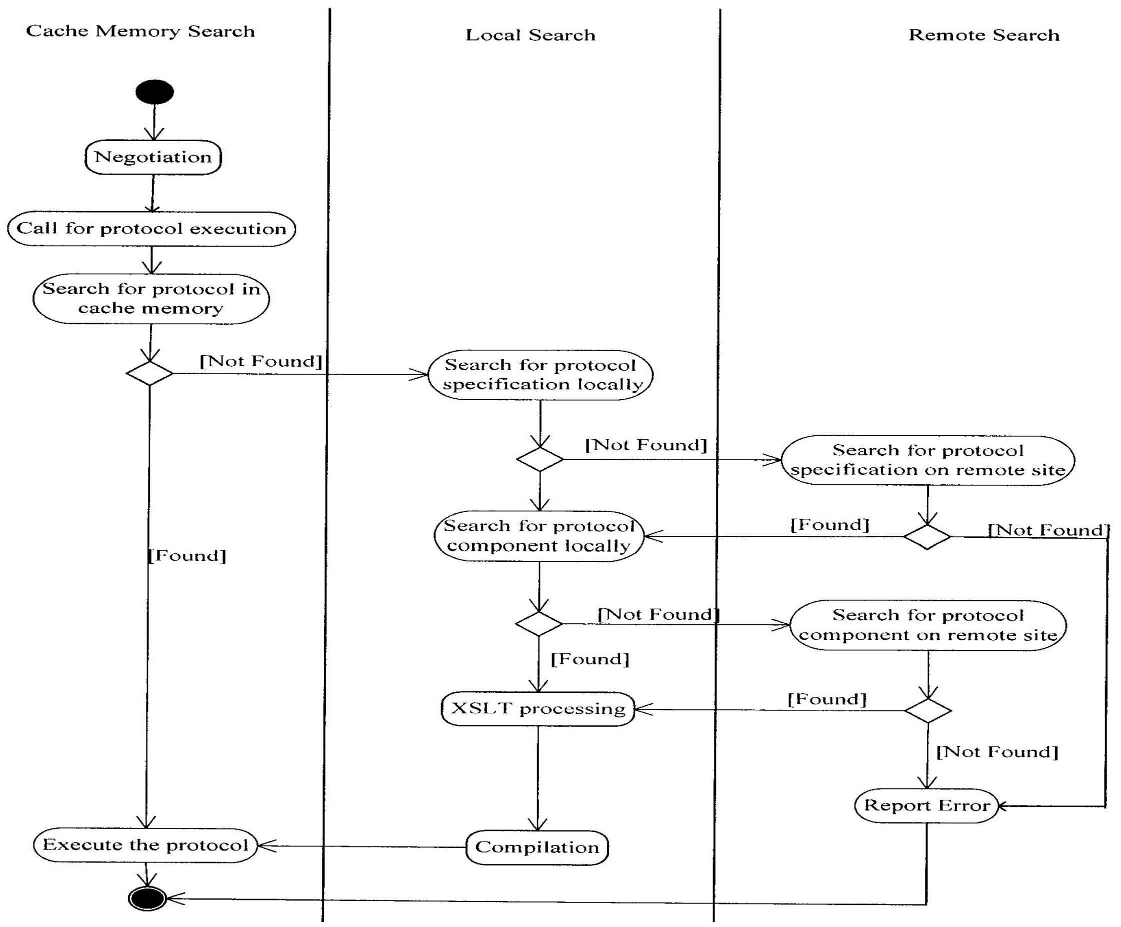

Cache Memory Search
Local Search
Remote Search
Negotiation
Call for protocol execution
Search for protocol in cache memory
[Not Found]
[Found]
Search for protocol specification locally
[Not Found]
Search for protocol specification on remote site
[Found]
[Not Found]
Search for protocol component locally
[Not Found]
[Found]
Search for protocol component on remote site
XSLT processing
[Found]
[Not Found]
Report Error
Execute the protocol
Compilation

Figure 9: UML activity diagram for the operation of the PCS

4. Conclusion and Future Work

The traditional approach for designing security protocols assumes an environment that does not change over time. This is not a valid assumption because no single solution will be able to accommodate the rapid evolution in communication protocols in general and specifically in security protocols for Internet applications. It is difficult to make changes to a monolithic implementation of a protocol. Users often do not have access to source code and even in the cases of open source; it is not trivial to change large pieces of code. In contrast, the Meta-Protocol approach, which proposed in this paper, assumes a dynamic environment and provides enough flexibility so that applications can select the specification that fits their exact needs. Users can also transmit their preferred protocol specification to their correspondent's parties to generate the code. The framework presented here provides a methodology to express protocol specifications in XPSL and a mechanism to exchange the specification among interested parties for automatic code generation and execution. In the negotiation layer, communicating parties negotiate the specification of the protocol they want to use. In subsequent layers, the specification is automatically transformed into code and loaded for execution as needed. The framework is supported by a system architecture that provides interfaces and databases needed for the operation of the system. In the future work, a component should not have several names on different machines because this creates confusion and may prevent PCS systems from locating the correct component. Moreover, duplicates can occur because the PCS may contain several instances of the same components but with different names. This issue is one of the most challenging issues in software engineering and in CBSE. No easy solution is available for these problems except through standardization (e.g., IANA).

References

1. Hutchinson, N. and Peterson, L. Design of the x-kernel. ACM Symposium proceedings on Communications Architectures and Protocols, 1988, pp 65-75.
2. Hutchinson, N., and Peterson, L., The x-kernel: an architecture for implementing network protocols, [EEE Tr. Software Engineering, Vol. 17, Issue 1, 1991, pp 64-76.
3. The Cactus Project, University of Arizona, Computer Science Department, http://www.cs.arizona.edu/cactus/
4. Hiltunen, M. Schlichting, R. and Wong, G. Implementing Integrated Fine-Grain Customizable QoS using Cactus The 29th Annual International Symposium on Fault-Tolerant Computing (Fast Abstract), Madison, WI, June 1999, 59--60.
5. Hiltunen, M. and Schlichting, R. The Cactus Approach to Building Configurable Middleware Services, Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), Nuremberg, Germany, October 2000.
6. Hayden, M. The Ensemble System. PhD thesis, Cornell University, Jan. 1998.
7. Morris, R., Kohler, E., J annotti, J., and Kasshoek, M., The Click router, Proc. 17th ACM Symposium on Operating Systems Principles, Dec 1999, pp. 217- 231.
8. Bhatti, N., and Schlichting, R., A System for Constructing Configurable High- Level Protocols, Proc. Conf. Application, Technologies, Architectures, and Protocols for Computer Communications, 1995, pp. 138-150.
9. Malley, S., and Peterson, L., A Dynamic Network Architecture, ACM Tr. Computer Systems, Vol. 10, No. 2, May 1992, pp. 110-143.
10. Tschudin, C., Flexible Protocol Stacks, Proceeding of the ACM conference on Communications architecture & protocols, 1991, pp. 197-205.
11. Rensesse, R., Briman, K., Friedman, R., Hayden, M., and Karr, D., A Framework for Protocol Composition in Horus, Proc. Annual ACM Symposium on Principles of Distributed computing, 1995, pp. 80-89.
12. Hiltunen, M. and Schlichting, R. A Configurable Membership services, IEEE transactions on Computers, Vol. 47, Issue 5, May 1998, pp. 537-586.
13. Huni, H., Johnson, R., and Engel, R., A framework for Network Protocol Software, Proc. OOPSLA95 ACM SIGPLAN Notices, 1995, pp.1-14.
14. Wong, G., Hiltunen, M., and Schlichting, R., A configurable and Extensible Transport Protocol, Proc. IEEE INFOCOM 20th Annual Joint Conf. IEEE Computer and Communications Societies, Vol. 1, 2001, pp. 319-328.
15. Stallings, W., Network Security Essentials: Applications and Standards, Prentice-Hall Inc, New Jersey, 2000.
16. Internet Security Association and Key Management Protocol (ISAKMP), http://www.ietf.org/rfc/rfc2408.txt.
17. Abdullah, I., and Menasce, D., Protocol Specification and Automatic Implementation Using XML and CBSE, Proc. of the IASTED International conference on Communications, Internet and Information Technology (CIIT2003), Scottsdale, Arizona, USA, Nov 03. pp. 191-196.
18. Abdullah, I., and Menasce, D., Unified Architecture for the Implementation of Security Protocols, Proc. of the 16th International conference on Computer Applications in Industry and Engineering (CAINE03), Las Vegas, Nevada, USA, November 03. pp. 95-100.
19. Glass R., Some Thoughts on Automatic Code Generation, ACM SIGMIS Database, Vol. 27, Issue 2, spring 1996, pp. 16-18.
20. Abdullah, I., Sibley, E., A Step toward Building Dynamic Security Infrastructure, Proc. of the Fifteenth International Conference on Software Engineering and Knowledge Engineering- SEKE03, San Francisco Bay, CA, USA, 1 to 3 of July 2003, PP. 483-488.