Security Enhancements for a User-Controlled Lightpath Provisioning System

Jing Wu, Michel Savoie, Hanxi Zhang, Scott Campbell
Communications Research Centre Canada
3701 Carling Avenue, Ottawa, Ontario, Canada, K2H 8S2
{jing.wu, michel.savoie, hanxi.zhang, scott.campbell}@crc.ca

ABSTRACT - User owned and managed optical networks offer new benefits compared to carrier networks. There are basically two types of user owned and managed optical networks: metro dark fibre networks and long-haul wavelength networks. A user-controlled lightpath provisioning system is designed to address the network management challenges, where only the customer has complete visibility of its own network and no provider can see all the network elements. The prototype management software has a service-oriented architecture and uses the Jini and JavaSpaces technologies. Within one management system for a federation, there are six key components: a Jini Lookup Service, an instance of JavaSpaces for storage of Light Path Objects (LPOs), a Jini Service Access Point (SAP), an LPO service, an instance of switch communication service for each switch in the transport layer and a Grid SAP. Since the new management system is a distributed system and the new management system may be deployed over a public Internet infrastructure, secure access to the management modules is required. The application of existing system security technologies to the new management system is analyzed. To securely transfer objects across a network, SSL is used to encrypt RMI data streams and thus data streams between Jini services. To securely execute a dynamically downloaded Java class, Jini adopts the Java security model. To securely use a dynamically downloaded proxy to communicate to a remote service, Jini Extensible Remote Invocation is implemented to support security features such as invocation constraints, remote method control, and the trust verification model.

KEYWORDS - Network Management Software, Distributed Software Security, User-Controlled Lightpath Provisioning

I. INTRODUCTION

There are basically two types of user owned and managed optical networks: metro dark fibre networks and long-haul wavelength networks [12, 9]. Schools, hospitals and government departments are acquiring their own dark fibres in metropolitan areas. They participate in so-called “condominium” dark fibre networks to better manage their connectivity and bandwidth. They light up the fibres with their own equipment and interconnect their fibres to other institutions, commercial service providers or Internet Exchanges. In the long-haul area, many providers are selling or leasing point-to-point wavelength channels. Some providers are offering “condominium” wavelength solutions, where a number of users share the capital costs of deploying long-haul optical networks. In return, each user in the condominium consortium owns a set of wavelength channels.

User owned and managed optical networks offer new benefits compared to carrier networks. In user-owned optical networks, the cost of bandwidth is substantially reduced, as it now largely becomes a capital cost rather than an ongoing service charge. The user is able to optimize the overall resource utilization. The user purchases dark fibres and/or wavelength channels from a number of independent suppliers, and participates in condominium wavelength networks. Therefore, the user has more flexibility in network planning and deployment and is able to negotiate better deals from different suppliers. The user may fine-tune the usage of each resource. User-managed networks reduce Internet costs via remote peering and transit. Since the user directly owns and manages an optical network, the bandwidth and quality of service are guaranteed. User-controlled lightpath provisioning is a traffic engineering mechanism for inter-domain applications. The primary application for this technology is high-end scientific research in high-energy physics, astronomy, bio-informatics, etc.

User-controlled optical networks face new technical challenges in resource management, operation co-ordination, and information storage management. Managing networks with resources from different suppliers is a new issue that has not been fully addressed by existing network management techniques [12]. Now, only the customer has complete visibility of its own network and no provider can see all the network elements. This is in contrast to the traditional centrally managed networking technologies, e.g. Generalized Multi-Protocol Label Switching (GMPLS) and Automatic Switched Optical/Transport Network (ASON/ASTN), which assume that the provider manages network elements and a common management system is used. Rather than any provider, the user is in a better position to decide the optimal solution for protection and restoration. The protection and restoration need to be coordinated among multiple providers. The collaboration among multiple independent users is critical for end-to-end connection provisioning. User-managed networks adopt the peer-to-peer architecture, in which users peer with each other. Each user not only receives transport services from other users but also contributes new transport services. During the establishment of an end-to-end connection, each connection segment is set up on a peer-to-peer basis. Central guiding intelligence and arbitration of conflicts may be necessary, but day-to-day management and per connection control are decentralized. How to search and take control of partner’s shared resources has to be addressed. Policy enforcement, authorization and authentication have to be applied.

The research on building a user-controlled network was initiated by CANARIE, Inc., Canada’s advanced Internet development organization. The Internet inter-domain routing protocol, Border Gateway Protocol (BGP), was extended to facilitate lightpath routing. OBGP takes the BGP routing table at the source node and queries along Autonomous System (AS)
paths to check if there are available lightpaths [1]. Later on, it was realized that network management systems offer more flexibility in providing new functions and features than a distributed control plane. In 2003, CANARIE launched a directed research program to design and prototype network management systems enabling user-controlled lightpath provisioning [8]. A Web-service based management system was designed primarily focusing on the partitioning and concatenation of inter-domain channel resources and transferring ownership among users [2], assuming a central database provides a global view of the inter-domain topology and resource availability on all inter-domain links. A policy based admission control was proposed in another prototype system [10] to regulate the user’s request to the connection provisioning. The nature of inter-domain management was addressed by using a global directory to provide links to the management modules for every domain.

We are motivated to design and prototype a management system directly confronting the challenges of user-centered network management. To provision a connection covering multiple domains, the management modules for the involved domains need to collaborate in a peer-to-peer manner. It is critical to avoid any central resource database or directory. Since optical networks are dominant transport networks, we prototyped our system for optical networks such as wavelength-routed Wavelength Division Multiplexing (WDM) networks, Synchronous Optical Networks (SONET) or Synchronous Digital Hierarchy (SDH) networks. Therefore, we call our system a User-Controlled Lightpath Provisioning (UCLP) system. However, the design is generally applicable to any type of inter-domain circuit-switched or bandwidth guaranteed packet-switched connection provisioning. Recently, our research partner at Technical University of Catalonia, Spain, extended the UCLP system to manage Multi-Protocol Label Switching tunnels.

Since the UCLP system has a distributed architecture, the system reliability and security are of particular significance. The whole system is required to operate even if service modules are temporarily unavailable. For example, when a communication failure between two management modules occurs, the management system should not run into an unstable state. The management function should be available after the failed communication is recovered. The access of the management modules should be protected by security features.

In this paper, we discuss the security enhancements for the UCLP system. In Section 2, we give an overview of the Jini technology, which is used to build our system. Then the UCLP system architecture is presented in Section 3. In Section 4, we describe the techniques used to improve the system security. We conclude the paper in Section 5.

II. JINI TECHNOLOGY

Building on top of the Java technology, Jini provides an infrastructure for defining, advertising and searching services in a network [7, 3]. The Java Virtual Machine (JVM) provides a hardware-independent software execution platform. The Jini infrastructure spans multiple JVMs that may run on different devices or computers connected via a network. Unless we want to emphasize a module is a user or provider of a function, we generally called a Jini module as a Jini service, because a module may provide services to other modules and meanwhile the same module may be a client of other modules utilizing other modules’ services. Since Jini services may be spread throughout a network, Jini technology needs to define, advertise and search services and then enable services to communicate to each other over a network.

A. Service Lookup and Discovery

The Jini Lookup Service (JLS) is central to the Jini technology. The primary function of the JLS is to let a service know the existence of other services. On the other hand, a service announces its presence by registering to a JLS. Fault-tolerance may be provided by using multiple JLSes for the same group of Jini services. A JLS itself is a Jini service and can be registered to other JLSes and therefore can be searched for.

A Jini service locates a JLS by using service discovery. Two types of service discovery are supported: multicast and unicast service discovery. In a multicast service discovery, a Jini service sends out a multicast message and then listens to possible replies from active JLSes. Multicast service discovery does not require a Jini service know the location of a JLS. But, it depends on the IP multicast. A JLS may not be located within the IP multicast range of the Jini service. In a unicast service discovery, a Jini service specifies the Universal Resource Locator (URL) of a JLS. In addition to the two service discovery mechanisms, a JLS may actively announce its presence so that other services within the same multicast domain are notified.

A Jini service registers its proxy to JLSes, so that available Jini services may be searched for and the registered proxies may be downloaded to where the Jini services are to be called. A proxy of a Jini service is a stub that contains an interface to the Jini service. Several proxy downloads or uploads are involved in a Jini system. First, after a Jini service discovers a JLS, a proxy for the JLS (called a registrar) is downloaded to the Jini service. Second, the Jini service uses the registrar to register to the JLS by uploading the Jini service’s proxy. Third, when other services search for a Jini service in the JLS by describing an interface type (written in the Java Programming Language) and possibly, other attributes. The Jini service’s proxy may be downloaded to other clients when a match is found. Other services use the downloaded proxy to invoke the Jini service.

B. Remote Method Invocation and Dynamic Code Downloading

The Java Remote Method Invocation (RMI) supports distributed Jini services to communicate to each other [3]. Jini services that are implemented as Java objects may be distributed on different JVMs and be hosted by different networked machines. RMI supports a Java object to utilize another remote object’s service. The two objects running on different JVMs are remote to each other. The service is implemented as methods in the object. The service user and provider have a common understanding about what functions the service offers and what are the input parameters and return values by using the same group of Java interfaces to describe the service. Two objects,
The stub and skeleton use the serialization mechanism to convert the content of the corresponding object into a byte stream that can be sent over a network. The input parameters and return values may be of primitive types (such as integer, character, boolean, etc.), or objects. Primitive-type input parameters and return values are marked by the source end, sent over a network, and recognized by the destination. Representing the content of a particular instance of a class needs special treatments, because there is no standard approach to mark and recognize objects as for primitive types. The serialization mechanism is used to package up the member data within an object. A serialized object may be written to a file, or stored as an intermediate object of a special Java class type, called marshalled object, before it is recovered (i.e., unmarshalled) to a functional object that can run on a JVM. The JLS stores and forwards the Jini service proxy as a marshalled object.

The class file corresponding to a serialized object is dynamically downloaded into the JVM where the object is to be reconstructed. The class file is required to create a functional object, because the serialization only packages up the states of all the member variables within the object, not the bytecodes for the object itself. To interpret and translate the received serialized object, the class file for the object needs to be downloaded into the JVM where the reconstruction of the object is to take place. The place from where the class file is downloaded is called the codebase of the class and is packaged into the serialized object. Tagging the codebase to the serialized data stream is called annotating the stream with the codebase, because the stream depends on the codebase class to be interpreted. After receiving the byte stream representing the serialized object, the receiver discovers the codebase of the class file, downloads the class file, and then reconstructs an object identical to the object that is serialized at the sender. The codebase is specified as a URL. For example, when a client downloads a Jini service proxy from a lookup service, the client also needs to download the stub class file from the Jini service proxy’s codebase, in most cases, from a directory or a Java ARchive (JAR) file that the service made available for downloading.

C. Jini Federation

Jini services for the same management domain create a Jini federation, in which a Jini service may access other Jini services in the same Jini community via service proxies dynamically downloaded from a local JLS. Jini services in the same federation collectively offer a group of functions under the same administration. Therefore, basic notions of trust, identification and policy are presumably agreed upon.

The integration of a JLS with other types of naming or directory services provides a means of hierarchical lookup. One option is to include other lookup services as service objects in a JLS. The other option is to place a reference to a JLS in other naming or directory services, providing a means for clients of those services to access a Jini system.

III. A MANAGEMENT TOOL DESIGN BASED ON JINI AND JAVA SPACES

A. System Architecture

Figure 2 shows the distributed deployment of two copies of the UCLP system, one for each independently managed domain (i.e., a federation). Within one management system for a federation, there are six key components [13]: a Jini Lookup Service (JLS), an instance of JavaSpaces for storage of Light Path Objects (LPOs), a Jini Service Access Point (SAP), an LPO service, an instance of switch communication service for each switch in the transport layer and a Grid SAP.

B. Lightpath Management Services

Figure 3 shows the system architecture in more detail and indicates the main service methods provided by the component interface. The management services provided by our system are classified into two groups: those only available to administrative users and those available to all users. The latter include in particular “ConnectionRequest” by which a user can request the establishment of an end-to-end connection from a given entry port of a given switch to a given exit port on another given switch, possibly belonging to a different federation. One of the functions reserved to administrative customers is the addition of new physical links to the available optical network.

In our design, LPOs are objects stored in JavaSpaces. An LPO is an abstraction of a lightspan. It is associated with a set of attributes and methods that enable possible peering to other LPOs at a switch to create an end-to-end connection or a longer lightspan. Supported customer operations on LPOs include: concatenating two LPOs, partitioning one LPO into many LPOs sharing common start and end points but with smaller bandwidth allocations, and reserving/using/releasing LPOs. The administrative operations include adding new LPOs and deleting LPOs corresponding to changes in the physical layer and the allocation of new resources.
For the execution of the ConnectionRequest, the Jini SAP uses the internal methods FindSwitchPath and FindLPOs. The latter searches through the pertinent JavaSpaces to look for LPOs with attributes suitable for the end-to-end connection to be built. It also uses the functions provided by the methods of the LPO service.

IV. SECURITY ENHANCEMENTS

A. Secure Socket Layer (SSL)

To securely transfer objects across a network, SSL is used to encrypt RMI data streams and thus data streams between Jini services. SSL provides data stream security at the session layer on top of TCP/IP. SSL offers four functions: server authentication, data encryption, message integrity and optional client authentication [6]. Compared to encrypting IP flows using the IPsec protocol or encrypting RMI data streams using public and private keys, SSL demands less computation by using the same session key to encrypt and decrypt RMI data streams. In the session establishment phase, a session key is derived from a randomly generated number. The secure distribution of the session key between the authenticated server and client is achieved by the client’s encryption of the session key using the server’s public key and the server’s decryption of the session key using the server’s private key. Practically, only with the server’s private key, can the encrypted session key be decrypted, and only the server knows its own private key, so nobody else can intercept the session key. The encrypted RMI data streams can only be decrypted by using the right session key, which is solely known to the involved client and server.

The server and client authentication in SSL has limitations in authenticating dynamically discovered services. In SSL, the server authentication assumes the client knows the identity of the server; the client authentication assumes the server maintains a list of authorized clients who are allowed to access the service. The server authentication only proves to the client that the server is the real one that the client intends to connect to. It is the client’s responsibility to make sure which server to trust and then to connect to. Similarly, the client authentication only verifies that the client is whom the client claims to be. The server needs to know which clients are allowed to use its service before any clients connect to the server. However, in Jini, clients and servers dynamically discover each other. It is not required that they know each other’s identity in advance. A
trade-off needs to be made between the effectiveness of SSL security and the dynamics and spontaneity of the Jini networking environment.

There is a security gap between the data stream security provided by the SSL and the system security observed by a Jini client, because the client’s communication to a remote service uses the RMI mechanism, which uses dynamically downloaded service proxies (i.e., service stubs) at the client [5]. The client needs a mechanism to verify the trust associated with the downloaded proxy. By attaching an authentication authority’s signature to the proxy and using a public key based decryption, the client may build a trust for the signed proxy. A different approach is proposed in Jini version 2.0 to verify proxy trusts without using public/private key based infrastructures. This will be discussed shortly in section 4.3.

B. Java security model

To securely execute a dynamically downloaded Java class, Jini adopts the Java security model. A security manager provides authorization to dynamically download classes by specifying which dangerous operations are allowed for each class. The security manager offers flexible, fine-grained security policy control. Based on the codebase and the optional signatures associated with the class file, the security manager looks up the security policy file whether a permission on a specific operation is granted. Examples of such operations include executing other application programs, shutting down the JVM, accessing other application processes, accessing system resources (such as print queues, event queues, system properties, and windows), file system and network operations, etc. By attaching a third-party’s signature to the bytecode, a bytecode is signed to guarantee the bytecode’s origination and the integrity of the bytecode. This bytecode signing procedure is similar to the certificate signing in the SSL, and also public-key based decryption is used. Multiple third-parties may sign the same bytecode, resulting in potential matching multiple policy entries in the security policy. In such situations, permissions are granted in an additive fashion.

Jini adds customerized permissions [7]. One such extension is the discovery permission. A Jini client or service may be granted permissions to join a specific group of JLSes. This restricts the JLSes that a Jini client or service can discover in a multicast discovery. The risk is reduced for a Jini client or service running into an unknown or undesired JLS. In the UCLP system, the domain that a Jini service (including a JLS) serves for is defined. Therefore, the multicast discovery permission can be properly set up. The location of the JLSes remains floating, so it is flexible to use different JLSes (e.g., backup JLSes) at various locations.

The Java security model is static and operates at the per class granularity. The security permissions are pre-configured in policy files. Pre-configured policies cannot be modified during the execution of an object. Although the bytecode verifier
examines downloaded bytecodes based on basic Java design rules, it is advantageous to delay granting security permissions until an object is going to be executed. Before an advanced proxy verification finishes, granting security permissions is risky.

C. Jini Extensible Remote Invocation (JERI) security features

To securely use a dynamically downloaded proxy to communicate to a remote service, JERI is implemented in Jini version 2.0 [11]. JERI supports security features such as invocation constraints, remote method control, and the trust verification model. JERI is an extension of the Java RMI model.

Proxy trust is a new security issue in Jini, because a Jini client relies on a dynamically downloaded proxy to communicate to a Jini service. JERI makes a good trade-off between the flexibility and the security of the Jini architecture by limiting pre-configurations required for security. Two assumptions are made for a client to verify a downloaded service proxy: i) the client knows the server’s identity; and ii) the client has a locally pre-installed small bootstrap proxy to securely connect to a server. The flexibility of using dynamically downloaded service proxies is maintained by not limiting where the proxies are downloaded from and who signs the proxies. JERI avoids using public-key based proxy authentication because of the complexity associated with the authentication authority management, public key distribution to all potential clients, and signing large number of proxies that may change over time.

JERI supports dynamic policy for a downloaded Jini proxy in addition to existing static permissions in the Java security model. Dynamic policy allows a Jini client to delay the permission granting until after the client has fully verified the proxy trust. A client may specify constraints on the behaviour of remote invocations through a downloaded proxy. A server may specify its constraints on the service based on the incoming remote requests. The constraints are assigned to individual remote invocations as opposed to the entire proxy as for static permissions in the Java security model. When a JERI proxy wants to invoke a remote method, the client’s and the server’s constraints are combined for that particular method, taking into consideration the execution context related constraints such as time related constraints. For example, a client may require a server authentication to execute a remote method. A client may require the server to be authenticated as any subset of a list of names. Independently, a server may require a client authentication as well. Both a client and a server may choose to require encryption on the data streams between them, verifying the data is unaltered during transmission, etc.

V. CONCLUSIONS

The service-oriented architecture is used in the management system for user-controlled lightpath provisioning. The architecture is proven to be modular and easy to maintain. Jini and JavaSpaces are suitable for the management system thanks to their rich functions such as supporting leasing, transaction and event notification. We demonstrated our prototype management system in several events showing its capability in setting up lightpaths directly controlled by users.

Since the management system transfers information over public Internet infrastructures and uses dynamically downloaded management modules or their proxies, system security is required. Secure socket layer is used to securely transfer information over public networks. Java security functions, such as granting execution permissions, help to securely execute a dynamically downloaded Java class. Jini extensible remote invocation offers new security features that improve our system security. These security features are analyzed in the context of our management system.

VI. ACKNOWLEDGEMENTS

The research is partially funded by CANARIE’s directed research program on UCLP. We thank Bill St. Arnaud at CANARIE, Inc. for his leadership and innovative vision for the UCLP research. We thank Prof. Gregor von Bochmann at the University of Ottawa for his contributions in the discussions and system design and the team he led, including Jun Chen, Wei Zhang and Ling Zou for their contributions to the implementation of a prototype system. We thank Mathieu Lemay (Inocybe, Québec); Sergi Figuerola, Eduard Grasa, Joaquim Recio and Albert López (i2CAT, Spain) for their participation in the discussions.

VII. REFERENCES