INTERNET-DRAFT
Intended Status: Informational
Expires: April 11, 2013

R. Fernando
J. Medved
D. Ward
Cisco

A. Atlas
B. Rijsman
Juniper Networks

October 11, 2012

IRS Framework Requirements
draft-rfernando-irs-framework-requirement-00

Abstract

The Interface to Routing System (IRS) allows an application to programmatically query and modify the state of the network. This document defines the requirements for IRS with appropriate reasoning where required.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

Copyright and License Notice

Table of Contents

1 Introduction

Routers, switches and network appliances that form today's network infrastructure maintain state at various layers of detail and function. For example, each router has a Routing Information Base (RIB), and the routing protocols (OSPF, ISIS, BGP, etc.) each maintain protocol state and information about the state of the network.

IRS [IRS-FRMWK] defines a standard interface through well defined APIs to access this information. The information collected by an application could be used to influence the routing system in conjunction with user defined policies in a feedback loop.

IRS enables this feedback loop so that applications can not only collect information but also use them to influence the network. The goal is to facilitate control and diagnosis of the routing infrastructure, as well as enable sophisticated applications to be built on top of today's network infrastructure.

Over time applications would evolve and with it their requirements too. IRS MUST be extensible so that future requirements can be easily factored in. IRS should be modular and extensible. It should be simple to understand and friendly to application developers.

This document describes some of these requirements in detail taking into consideration the use cases described in [2]. Particular attention is paid to API and the application consumption model so that it is developer friendly.

This document's scope is purely to collect and document requirements for the IRS framework. This could serve three purposes:

a. To help the stakeholders (equipment vendors, application programmers or interested IETF participants), to arrive at a common understanding of the important elements of IRS.

b. To provide requirements to the designers of IRS framework on the different aspects of the framework that needs consideration in the design process.

c. To allow the stakeholders to evaluate technology choices that are suitable for IRS, to identify gaps in them and to help evolve them to suite IRS's needs.

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",

"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. IRS Overview

IRS provides a standard interface for applications to read and write state in a network device. Since the application and the network device could reside in different physical nodes, IRS could be viewed as a distributed client-server system.

IRS can also be viewed as a "framework" that helps reduce the "start up" cost in developing network based applications. A framework codifies a set of principles, patterns and software artifacts that allow application developers to quickly develop new applications.

Instead of designing each application from scratch, the IRS framework provides a set of infrastructure that abstracts the application indepedent mechanisms. This approach enhances software agility, reusability and portability.

This document aims at making sure that the requirements of the IRS framework are well articulated by describing its high level objectives, the concepts and components involved, how they are related and what their requirements are.

3. IRS Framework Terminology

Before we delve into the details of the IRS framework, it might help to establish some basic terminology.

Service: For the purposes of IRS, a service refers to a set of related state access functions together with the policies that control its usage. For instance, 'RIB service' could be an example of a service that gives access to state held in a device's RIB.

Server: Is a system that implements one or more services so that other client systems can call them through well defined interfaces. A server can export multiple services. A server is typically a network device.

Client: Is a system that calls a service implemented by a server through the well defined interface. A client can make use of multiple services from many servers. A client is typically a network application.

Participants: The server and client are collectively called the

participants of a service.

Transport: Is any mode of communication on an end-to-end basis between the server and client that allows them to exchange data. In principle, the transport hides the topology and other network properties from the participants of a service.

Messages: Messages are logical chunks of data that are exchanged between service participants.

Message Exchange Pattern: Is a categorization of different ways in which messages could be exchanged between service participants. MEPs specify the sequence, order, direction and cardinality of messages exchanged. Request-response and asynchronous notifications are examples of MEPs. MEPs are also sometimes referred to as the session protocol.

Message Data Model: The schema representing the structure of messages being exchanged between the service participants. The MDMs can specify certain constraints such as the data type, length, format and allowed values of fields in messages.

Message Encoding: The "wire" representation of messages exchanged between service participants.

API Method: Is an application level procedure or a function that is invoked by the client to query or modify the state held in the server.

Service Scope: Is the functional scope of a service. The service scope is established during the service definition phase.

Service Data Model: The schema representing the conceptual structure of the state held in the server for a given service. The SDMs can specify certain constraints such as the data type, length, format and allowed values for fields representing the state. They also describe the relationship between the state.

Modeling Language: Is a language that defines schema for Message Data Models and Service Data Models.

Namespaces: Allows a method for uniquely identifying and scoping of schemas declared for messages and services. Namespace is an important consideration when defining services and messages.

Service State or State: Is the general data held by the server for a given service.

State Element: A programmable state present in the server. State Element could vary in granularity.

State Identifier: A unique identity for the state element. The identifier is derived from the SDM and uses the same naming convention as the SDM. State Identifier can be viewed as the 'key' for the state.

State Value or 'value: This is a value that is assigned to a particular state identifier (key). The state is referred using the State Identifier or 'key' in operations that sets or transfers the value of the state.

State Owner: Identity of the client that was the source of a state held in the server.

State lifetime: The duration up to which the state is maintained in the server.

Datastore: This is the physical mechanism used to store a service's state.

Capabilities: Capabilities represents the functionality supported by a server including the services supported and exported to clients.

Authentication: Mechanism that allows a server to recognize the identity of a client.

Authorization: Determination of what an authenticated client is allowed to do.

Confidentiality: Specifies if data remains confidential while in transit between service participants.

Policy: For the purposes of this document, a policy is an explicit user configurable modification to the default behavior of the system. The enforcement could be conditional; they could become effective only when certain conditions are met.

As can be seen, there are many aspects to be considered in designing the IRS framework. The next section decribes the broad objectives of the framework and breaks down the concerns so that each's requirements can be individually examined.

4. IRS Framework Design Objectives

The goal is to provide a framework with the infrastructural components needed to develop intelligent applications that control the network. These are some of the core guiding principles and objectives that should be kept in mind when designing that framework.

a. Requirements Driven: The design of the framework should be pragmatic and requirements driven. Having adequate provisions to meet the needs of current applications yet making key aspects extensible to meet future needs should be the goal.

b. Simple to Program: The success of any architectural framework depends on the how simple it is to understand and implement against. When presented with multiple choices to perform a function, choosing one of them instead of supporting all of them might lead to simpler design. In doing so, the design should consider the most important requirements and the most common deployment scenarios.

c. Standards Based: The need for a standards-based approach to network programmability has been recognized by many standardization groups including IETF. All aspects of IRS should be open standards based. However, IRS should specify mechanisms to extend it in vendor specific manner. The aspects of IRS that could be extended should be identified in this document and should be supported by an implementation.

d. Design for Scale and Performance: The design should meet current and future performance and scale needs. It goes without saying that scale and performance should be key criteria for making design choices. There are well understood design patterns that allow us to compose a scalable, high performing system.

e. Extensible: IRS will be deployed in environments whose requirements evolve over time. Hence the system should be designed with provisions that will allow significant enhancements to be added to meet specified future goals and requirements. An extensible and future-proof design will drive better adoption as it is a promise against future technology churn.

f. Promote Reuse: Reuse in this context refers to using existing tools, technologies and mechanisms instead of inventing them from scratch. It also refers to reusing a network device's current set of capabilities that applications could harness without reinventing them from scratch.

g. Promote Portability: Portability refers to the ease with which

software written for one device or environment can be moved to work seamlessly with another device or environment to achieve similar functionality. A fundamental requirement for IRS is to achieve predictive and consistent behavior when applications are migrated from one platform or environment to another.

h. Security: IRS could be deployed in environments where it might be subjected to threats and denial-of-service attacks that might cause intentional damage to the functioning of a network. This could be in the form of loss of service, degradation of performance, loss of confidentiality, etc. Therefore, the security aspects should be carefully thought through when designing IRS.

i. Separation of concerns: The components of the system should be decoupled from each other as much as possible to achieve clear separation of concerns. This modularity would allow for interchangeable design and implementation choices that address the individual components requirements.

j. Robustness: Robustness is the ability of a system to operate in the face of failures and errors. It is also its ability to correctly and predictably recover from such errors and to settle to a known state. Since applications that use the IRS framework are remote and would be controlling the entire network, ensuring fault tolerance is an important consideration.

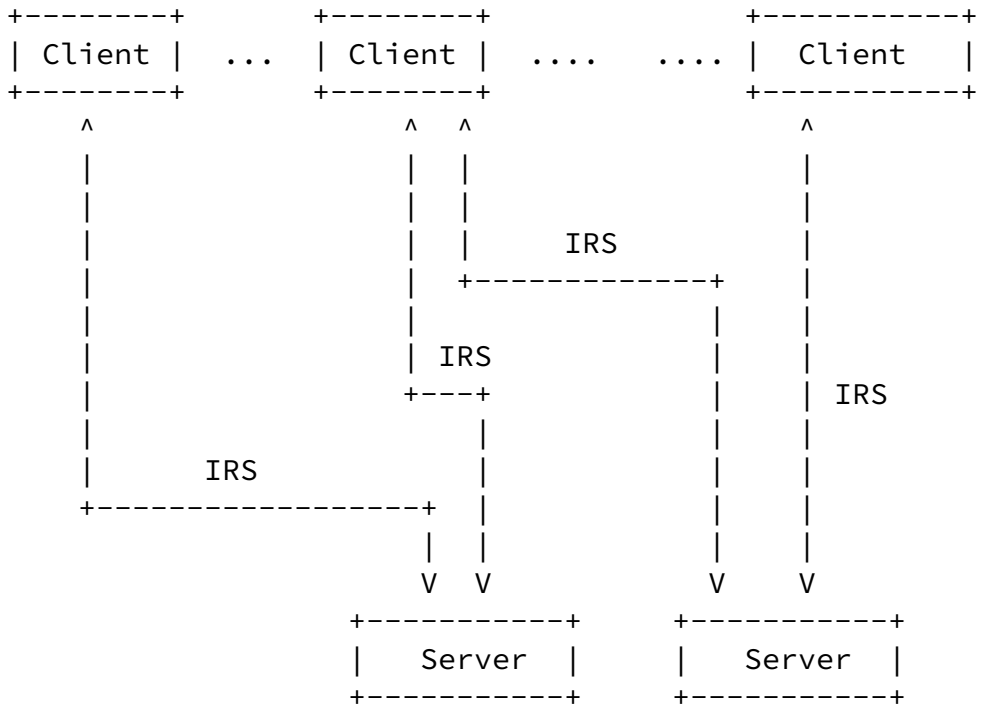
Most of these requirements cut across all the components of the system and hence should be kept in mind while designing each component and the system as a whole.

5. IRS Framework Requirements

This section is divided into multiple sub-sections, each dealing with a specific consideration of IRS framework design. As we list the requirements under each subsection, we'll annotate each requirement with what high level objectives they meet. A reason for creating the requirement is additionally provided where appropriate.

5.1 General Assumptions

This section captures the general, high level assumptions of the IRS framework. Since the design choices for the IRS framework are many, some simplifying assumptions could make the framework requirements more tangible and useful.



G.1 Programmatic access to the state held in a network device is provided to an application by exposing a set of API's from the device to the application. Due to this characteristic, IRS is a client-server protocol/framework. IRS must provide mechanisms for the client to discover services that a server provides.

G.2 The client can use the API's provided by the server to programmatically add, modify, delete and query state held in the server. Additionally clients can register for certain events and be

notified when those events occur.

G.3 The client and the server communicate using a simple transport connection. The client initiates the transport connection to the server. The server does not know the number and timing of the connections from its clients.

G.4 A service provides access to the state held in the server structured according to the SDM of that service. A service allows a client the ability to manipulate the service state.

G.5 The IRS MUST define a data model to describe the SDMs supported in the server and MUST define a data modeling language to formally describe that data model. IRS MUST specify the mapping from the service data model to the message data model and subsequently to the client API.

5.2 Transport Requirements

The transport layer provides connectivity between the client and the server. This section details the transport requirements.

T.1 There should exist a default transport connection between the client and the server for communication. This control connection is point-to-point and should provide in-order and reliable delivery of data in both directions. The simplest IRS setup will only have a single transport session between the participants.

T.2 Depending on the data being exchanged, there could be additional transport connections between the client and server defined in future. The characteristics of these additional transport connections will be dictated by the requirements that create them.

T.3 The transport connection between the client and server should have mechanisms to support authentication, authorization and optionally provide confidentiality of data exchanged between the client and the server. See 'Security Requirements' for more details.

T.4 A client could connect to multiple servers. Similarly, a server could accept connections from multiple clients.

T.5 The exact technology used for the transport layer should be replaceable. There should be a single mandatory transport that should be supported by all participants. This requirement will ensure that there is always an interoperable transport mechanism between any client and any server.

T.6 Clients and servers by default communicate using a point-to-point transport connection.

T.7 Point-to-multipoint transport are mainly used to scale the system by avoiding ingress replication when the same message has to be delivered to multiple receivers. P2MP transport would work hand-in-hand with a P2MP MEP. The subject of P2MP transport and P2MP MEP is for future work.

T.8 Once the transport connection is up, it is desirable to keep it up and use it to perform multiple operations. This requirement ensures that the system scales by amortizing the session setup cost across multiple operations. Session down events do not have an impact on the state maintained by the server.

T.9 After the transport connection comes up, the participants exchange capabilities and other session parameters before exchanging service related messages.

T.10 Messages pertaining to multiple services could be exchanged over a single transport connection.

T.11 The "default" transport connection between the client and server is purely for control plane message exchanges. Data plane packets are not expected to be sent over this "default" connection. When required, data plane 'punt' and 'inject' packets between participants could be designed as a service in itself that sets up a 'punt-inject-transport' that processes the right characteristics.

T.12 For operational reasons, there MUST be a need to identify a transport connection failure. To satisfy this requirement, transport level keep-alives could be used. If the underlying transport connection does not provide a keep-alive mechanism, it should be provided at the IRS protocol level. For example, if TCP is used as a

transport, TCP keep-alives could be used to detect transport session failures.

5.3 Identity Requirements

IRS could be used in a multi-domain distributed environment. Therefore a fool-proof way to ascertain the identity of clients is of utmost importance. Identity provides authenticated access to clients to state held by the server.

I.1 Each client should have a unique identity that can be verified by the server. The authentication could be direct or through an identity broker.

I.2 The server should use the client's identity to track state

provided by the client. State ownership enables the multiple clients to edit their shared state. This is useful during client death or disconnection when state owned by one client might be delegated to another client that shares the same identity.

I.3 The client's identity should be independent of the location or the network address of the physical node in which it is hosted. This allows the client to move between physical nodes. It also allows a standby client to take over when the primary fails and allows shared state editing by multiple clients as discussed in I.2.

I.4 A client that reboots or reconnects after a disconnection MUST have the same identity if it wishes to continue to operate on the state that it previously injected.

I.5 A clients ability to operate on a state held by the server is expressed at the granularity of a service. A service could be read-only or read-write by a client possessing a particular identity.

I.6 A policy on the server could dictate the services that could be exposed to clients. Upon identity verification, the authorized services are exported to the client by capability announcement.

I.7 A client can edit (write, delete) only the state that was

injected by it or other clients with the same shared identity. Therefore, two conditions must be met for a client to edit a state through a session. First, the client should receive capability from the server that it has 'edit' permissions for the service in question, and, secondly, the state that it edits should be its own state.

I.8 When there is a single client and it dies, operational provisions should be made to garbage collect its state by a client that shares the original clients identity.

I.9 The server retains the client's identity till all of its state is purged from the server.

5.4 Message Encoding Requirements

Clients and servers communicate by exchanging messages between them. Message encoding is the process of converting information content in a message to a form that can be transferred between them.

ME.1 Every message between the client and the server is encoded in a

transport independent frame format.

ME.2 Each message is serialized on the senders side and de-serialized on the receivers side. The technology used for encoding and decoding messages could be negotiated between the client and the server.

ME.3 A mandatory default encoding standard should be specified and implemented by all IRS participants. This ensures that there is an interoperable default encoding mechanism between any client and any server.

ME.4 The mandatory encoding technology chosen should be well supported by a developer community and should be standards based. Availability of tools and language bindings should be one of the criteria in selecting the mandatory encoding technology.

ME.5 If multiple message encoding is supported in the framework, the

encoding used for the current session should be configured using a policy on the server side and negotiated using capabilities. Note that currently there is no requirement to support multiple encoding schemes.

ME.6 The message encoding standard should be language and platform neutral. It should provide tools to express fields in messages platform independent IDL based language.

ME.7 The encoding/decoding mechanism should be fast and efficient. It should allow for operation on legacy equipment.

ME.8 The encoding scheme should allow for optional fields and backward compatibility. It should be independent of the transport and the message exchange pattern used.

ME.9 Human readability of messages exchanged on the wire might be a goal but it is secondary to efficiency needs.

5.5 Message Exchange Pattern Requirements

Message exchange patterns form the basis for all service level activities. MEPs create a pattern of message exchanges that any task can be mapped to whether initiated by a client or the server. This section provides the requirements for MEPS.

MEP.1 IRS defines three types of messages between the client and the server. First, capabilities need to be exchanged on session

establishment. Second, API commands send down from client to server to add, delete, modify and query state. And third, asynchronous notifications from server to client when interesting state changes occur.

MEP.2 The above message exchanges can be satisfied by two message exchange patterns. Capabilities and asynchronous notifications can be satisfied by one-way unsolicited fire and forget message. API commands can be satisfied using a request-response message exchange. The base IRS framework should thus support at least these two MEPs.

MEP.3 For a request-response MEP, the server should acknowledge every request message from the client with a response message.

MEP.4 The response message in a request-response MEP should indicate that the server has received the message, done some basic sanity checking on its contents and has accepted the message. The arrival of a response does not mean all post processing of the message has completed.

MEP.5 The response message should indicate an error and carry error information if there was a failure to process the request. The error code should be accompanied by a descriptive reason for the failure.

MEP.6 Error codes should indicate to the client which layer generated that error (transport, message parsing, schema validation, application level failure, etc). IRS framework should specify a standard set of error codes.

MEP.7 The request-response messages should be asynchronous. That is, the client should not stop-and-wait for one message to be acknowledged before it transmits the next request.

MEP.8 To satisfy MEP.5, there needs to be a mechanism such as a message-id, carried in the response that helps the sender correlate the response message to its original request.

MEP.9 The response messages need not arrive in the order in which the request was transmitted.

MEP.10 The request message should carry an application cookie that should be returned back to it in the corresponding response.

MEP.11 Besides the request-response MEP, there is a need for a fire and forget MEP. Asynchronous notifications from the server to the client could be carried using this MEP. Fire and forget MEPs can be used in both client-to-server and server-to-client directions.

MEP.12 The fire-and-forget MEP does not carry a message-id but it should carry a cookie that can be set by the sender and processed by the receiver. The cookie could help the receiver of the message to use the message for its intended purpose.

5.6 API Method Requirements

API methods specify the exact operation that one participant intends to perform. This section outlines the requirements for API methods.

A.1 The IRS framework should provide for a simple set of API methods, invoked from the client to the server. These methods should allow to add, modify, query and delete of state that the server maintains.

A.2 The IRS framework should provide for two methods, subscribe and unsubscribe, that the client can use to express its interest in specific state changes in the server.

A.3 The API methods discussed in A.1 and A.2 should be transported in a request-response MEP from the client to the server.

A.4 The API framework should provide for a single notify method from the server to the client when interested state changes occur. The notification method should be transported in a fire-and-forget MEP from the server to the client.

A.5 The framework should define a set of base API methods for manipulating state. These should be generic and should not service specific.

A.6 All API methods that affect the state in the server should be idempotent. That is, the final state on the server should be independent of the number of times a state change method with the same parameters was invoked by the client.

A.7 All API methods should support a batched mode for efficiency purposes. In this mode multiple state entries are transmitted in a single message with a single operation such as add, delete, etc. For methods described in A.1 and A.2 which elicit a response, the failure mechanism that is specific to a subset of state in the batch should be devised. Notify method should also support a batched mode.

A.8 Since the API methods are primarily oriented towards state transfer between the client and server, there should be a identifier (or a key) to uniquely identify the state being addressed.

A.9 API methods that refer to value of a particular state should carry the state identifier (key) as well as the its value. For instance, during a state add operation, both the identifier (key) and the value should be passed down from the client to the server.

A.10 Besides the basic API methods that are common to all services, a server could support proprietary methods or service specific methods. The framework should devise a mechanism to express these methods and their semantics through a modelling language or otherwise. The ability to support additional API methods should be conveyed to the client through the capability message.

A.11 Transactions allow a set of operations to be completed atom(all or nothing) and that the end result is consistent. This might be a requirement for some network applications and the framework designers should keep this requirement in mind during the design phase.

5.7 Service and SDM Requirements

S.1 Each service is associated with a service data model that defines the type and structure of the state pertaining to that service. IRS should provide mechanisms to manage the state held in the server in accordance to the SDM.

S.2 The data model should have the ability to express one-to-one, one-to-many and hierarchical relationships between entities.

S.3 The base IRS API methods should allow a client to add, modify, query and delete state information.

S.4 Neither the transport or the MEP should have any bearing on the structure of the state being transferred. Each service module in the server would be responsible for interpreting the structure of the state being transferred corresponding to the SDM.

S.5 A client, after proper identification, could operate on multiple 'services' that are exported to it. A client could have read-only or read-write access to a service. This is expressed by exchanging capability information with the client.

S.6 The arrangement and structure of state (SDM) should be expressed in a network friendly data modelling language.

S.7 Service data model once defined should be able to be extended. Service data models should be able to express mandatory and optional

elements. If should also have the ability to express exceptions

for unsupported elements in the model. These are requirements for the modelling language.

S.8 For every service that it wishes to expose to a client, the server should send capabilities that indicate the service data model, any exceptions to it and the optional features of the data model that it supports.

S.9 A service data model could be dependent on another SDM and should have the ability to refer to state elements in another service data model.

S.10 A state element expressed in a data model could be writeable by a client or purely readable. Readable state elements are populated and managed by the server and clients don't have the ability to write their value. Routing next-hops added by a client is an example of read-write state. Statistics associated with that next-hop is an example of read-only state. The modelling language should have the ability to express this constraint.

S.11 Query and notification API should be able to carry both read-only as well as read-write state.

S.12 Besides specifying a SDM, a service should also specify the interesting state changes that clients can subscribe to for notifications.

S.13 A client which is authenticated to access a service (either read-only or read-write) can subscribe to state change events.

S.14 A subscribe method should optionally have a filter associated. This increases the efficiency by filtering out events that the client is not interested in. The notification filter should have the ability to express state identifiers and wildcards for values.

S.15 The base API operations should be generic and allow a client to operate on multiple services with the same set of methods. Each service dictates its one schema or SDM.

S.16 IRS protocol should allow a server to export standard services as well as vendor proprietary services. A namespace scheme should be devised to recognize standard and proprietary services.

S.17 The server should indicate to the client the availability of infrastructure to manage the state that it maintains. This includes but not limited to the availability of persistent store, the availability of timer to clean up state after a specified

timeout, the ability to clean up state on the occurrence of an event, etc. Equipped with this information, the client is responsible for the lifetime of the state.

S.18 Each state should have a set of meta data associated with it. This includes the state's owner, the state's lifetime attributes, a creation and modification timestamp, etc. This information would aid in the debugging of the system. An authenticated client that is exposed to a service should also have access to the meta data associated with that service's state.

5.7 Security Requirements

Security requirements should be thought through up front to avoid expensive rework to the framework. Adding security requirements once the system is designed could be an expensive and painful process. This section calls out some security concerns to be kept in mind while designing the framework.

SEC.1 Every client should be authenticated and associated with an identity. A secure mechanism to uniquely identify a client such as certificates should be adopted.

SEC.2 Every client should have an authorized role whereby only certain state can be accessed and only certain operations can be performed by that client. To keep the model simple and applications portable, authorization should be at a per service level and not on individual state element level.

SEC.3 The framework should provide for information confidentiality and information integrity as options.

SEC.4 Every state maintained by the server should be tagged with the client's identity as well as meta-data to indicate last access and last modifications time-stamps. This ensures accountability and helps auditing the system.

SEC.5 The framework designers are strongly encouraged to provide mechanisms to "hook" into third-party security infrastructure to achieve these security goals whenever possible. This keeps applications programmers free of security concerns and yet provides a flexible, configurable and well integrated security model.

5.8 Performance and Scale Requirements

Performance requirements are usually weaved in with the functional requirements of a system. They feature in every decision made to fulfill the systems requirements. Performance and scale are a complex function of many things. Hence performance requirements cannot be precisely quantified by a single number. This section lays out some common sense guidelines that should be kept in mind while designing the system from a scale and performance standpoint.

PS.1 The request-response MEP should be asynchronous. This ensures that a system is not stuck waiting for a response and makes the entire system more responsive and increases concurrency between operations.

PS.2 When applicable, messages should carry application level cookies that enable an application to quickly lookup the context necessary to process a message. The management of the cookie is the applications responsibility.

PS.3 The framework should allow for bulk operations which amortizes the communication and messaging costs.

PS.4 Provide for a binary encoding option for messages between the participants.

PS.5 Provide for a non-encrypted transport between the service participants.

PS.6 Provide for message prioritization.

PS.7 Multiple operations could be completed with one transport session.

PS.8 Keep the server as stateless with respect to the number and location of each client.

PS.9 For notifications, support filtered subscription.

PS.10 If a client requires to re-synchronize state with the server, device a mechanism to do this efficiently without transferring all the state between them.

PS.11 Allow clients that perform infrequent operations to disconnect their transport connection without cleaning up their state.

PS.12 Create the basic necessary mechanisms in the framework and build everything else as a service if possible.

5.9 Availability Requirements

The ability of the system to withstand operational failures and function in a predictable manner is called availability. A few guidelines that are important are,

A.1 Provide a 'superuser' identity that is capable of changing security policies, clearing state and perform other actions that override client initiated actions in the system.

A.2 Handle session disconnection and client deaths gracefully. These should have the least impact on the system.

A.3 Log client connections and disconnections and provide this as a well known service to authenticated users.

A.4 Notify clients of message processing and other errors through error codes in messages.

A.5 Have a mechanism to gracefully terminate the session between the client and the server.

A.6 Provide a mechanism for authenticated clients to query the load attributes of the system, both instantaneous and running average. Provide this as a service.

5.10 Application Programmability Requirements

The framework should pay particular attention the the requirements of application programmers. A well written framework should improve the productivity of programmers and shorten the time to make an application. This section has some issues to consider when devising the framework from an applications standpoint.

AP.1 A client programming framework should allow applications writers to focus on the app functionality rather than mechanisms required to communicate with the server.

AP.2 The application once written to certain requirements should be portable to other identical environments. The framework should not have fine grained data access controls as this would lead to a poorly written application with portability issues.

AP.3 The framework should be devised in a manner that it is possible to automate code generation and constraint checking in popular programming languages. Generated code can then be used readily by

application programmers instead of dealing with the nitty-gritties of the system.

AP.4 Define a common repository for SDMs from which clients can obtain the SDMs they are interested in and automatically generate most of the boilerplate code.

AP.5 Provisions should be made for debugging & troubleshooting tools that includes message trace, call traces, access to relevant server traces and logs, packet decode tools to trace & decode messages on the wire, consistency checkers of state inserted into a server.

AP.6 The toolset should have a general portion (for common functions, such as session management) and SDM-specific portions (for example, a flag to control generation of debug code in code generated for a particular SDM).

AP.7 The framework should define SDMs and MDMs in a language neutral format so as to enable code generation in multiple programming languages.

5.11 Operational Requirements

O.1 There is a need to identify operational performance parameters of the system and provide mechanisms to retrieve them from a running system.

O.2 Provide a way to upgrade a service independently of the other services. This modularity allows uninterrupted operation of the all but one service which is being upgraded.

O.3 Provide a detailed workflow for bringing about a new service. This workflow will start with the need to introduce a new service and address the following: How SDMs defined? Where are they standardized? How are new entities (MEPs, transport, encoding) introduced? What are the tools and workflow involved to develop and operationalize a service. The intent is to introduce a level of understanding about stakeholders responsibilities.

O.4 Provide mechanisms and methodologies to test a new service before deployment.

6 Security Considerations

See "Security Requirements", section 5.7 above.

7 Acknowledgements

Thanks to the following people for reviewing and providing feedback: Alexander Clemm, John McDowell.

8. References

8.1 Normative References

[IRS-FRMWK] A. Atlas, T. Nadeau, D. Ward, "Interface to the Routing System Framework", draft-ward-irs-framework-00

Authors' Addresses

Rex Fernando, Ed.
170 W Tasman Dr,
San Jose, CA 95134

EMail: rex@cisco.com

Jan Medved
Cisco Systems
170 W Tasman Dr,
San Jose, CA 95134

Email: jmedved@cisco.com

David Ward
Cisco Systems
170 W Tasman Dr,
San Jose, CA 95134

Email: wardd@cisco.com

Alia Atlas
Juniper Networks
10 Technology park Drive
Westford, MA 01886

Email: akatlas@juniper.net

Bruno Rijsman
Juniper Networks
10 Technology Park Drive
Westford, MA 01886

Email: brijsman@juniper.net