OpenOSP

Product Overview

10 April 2001

Document Version 1.5

Data Connection Manual MSM-0005-0105

Notice

Data Connection Limited
100 Church Street
Enfield
Middlesex
EN2 6BQ
England

+44 20 8366 1177
http://www.dataconnection.com

Contents

1 Introduction

This document provides an overview of the OpenOSP open source Open Settlement Protocol (OSP) server protocol stack jointly developed by Cisco Systems and Data Connection Limited (DCL).

The OpenOSP server protocol stack is a full function, secure, highly scaleable C implementation of OSP for the Sparc Solaris platform (although it is engineered to be easily portable to other UNIX variants). It is based on version 2 of the OSP protocol as defined in ETSI document TS 101 321 v2.1.0 (2000-05). The stack supports

- all the server OSP primitives, including usage metering, pricing exchange, call authorization, subscriber authentication, and client capabilities

- all the defined security mechanisms, including SSL/TLS, S/MIME and authorization and authentication tokens

- Cisco extensions to the protocol to enable routing choice by protocol type, and for authorization of prepaid subscribers.

Additionally, the product includes support for the SCEP protocol used by Cisco routers to interact with a certificate authority.

The stack is intended for use both by OEMs with existing open source software (OSS) solutions, and to provide new entrants with a large proportion of the componentry needed to develop a standalone OSP server. For OEMs needing to integrate OSP capabilities into an existing OSS solution, the stack provides APIs to make this integration easy and to maintain existing value-add features. For new entrants, the stack ships with simple implementations of all the components required to build and deploy a real OSP server infrastructure.

The license conditions for the stack are described in detail in the README file associated with this distribution. A copy of the license is also available at http://www.vovida.org, but in summary, the license allows you to develop commercial and non-commercial products using the OpenOSP source code, with minimal restrictions and conditions (which are simply designed to maintain the copyright on the source code and the integrity of the OpenOSP name). In particular, there are no components for which you have to acquire any form of paid license.

Using the OpenOSP distribution as the basis for an OSP server development will minimize your time to market for OSP support, save engineering resource and guarantee interoperability with Cisco's marketing-leading VoIP gateways.

- Time to market and engineering resource saving. Developing a fully functioned and secure OSP server is a complex and significant project, requiring knowledge of several advanced security and cryptography techniques, Sockets programming, HTTP and XML. Basing an OSP server development around the OpenOSP server source code, which handles the majority of these complexities and allows you to focus on the integration and value-add, will save you a minimum of 3 or 4 person years of engineering resource when compared to developing the entire stack in house.

- Interoperability. The OpenOSP server is tested and guaranteed to interoperate with Cisco VoIP gateways.

OpenOSP is a high quality open source product. It emphasizes, to a greater degree than is the norm among open source products, the higher-order features of performance, scaleability and robustness. OpenOSP has been designed and developed specifically for deployment as a robust and scaleable part of a service provider or Inter-Exchange Carrier's infrastructure.

- The stack is engineered to support an unlimited number of clients and concurrent transactions (subject to availability of memory and CPU cycles) and efficiently scale the supported transaction rate as the power and number of CPUs is increased. See section 2.3 for details of the scaleability targets.

- The stack is designed from the outset to be robust, and has thoroughly tested by both Cisco and DCL.

DCL is a UK-based software development company, founded in 1981, with extensive experience developing system level software for many server applications and environments. DCL has worked with many of the industry's leading companies, as a technology provider and development partner, including Cisco, Microsoft, IBM, Hewlett-Packard, Sun, Nortel Networks and Lucent.

The remainder of this document is structured as follows.

- Section 2 gives an overview of OpenOSP.

- Section 3 gives an overview of the internal architecture of OpenOSP.

To ask questions and share ideas about OpenOSP, please contact openosp@vovida.org. For support issues, please refer to the OpenOSP section on http://www.vovida.org.

2 OpenOSP Overview

OpenOSP is a full implementation of an OSP server, based on version 2 of the Open Settlement Protocol including backward compatibility with clients implemented to v1.4.2 of the OSP specification. OpenOSP is written in C for UNIX platforms with POSIX threads, with the development and testing being done on Sparc Solaris. As far as is possible the code does not use any platform-specific functions.

The final release of the code supports all of the OSP server functionality defined in the final OSP V2.0 specification. The release has been tested to a full product standard, including full functional testing of all supported OSP transactions, security mechanisms and ciphers, stress and robustness testing, and scaleability testing.

The specific level of support in the release is as follows.

- All OSP messages
  - AuthorizationRequest – AuthorizationResponse
  - AuthorizationIndication – AuthorizationConfirmation
  - ReauthorizationRequest – ReauthorizationResponse
  - PricingIndication – PricingConfirmation
  - SubscriberAuthenticationRequest – SubscriberAuthenticationResponse
  - CapabilitiesIndication – CapabilitiesConfirmation
  - UsageIndication (including the enhanced usage reports defined in Annex C of the OSP specification) – UsageConfirmation.
- A rich set of security capabilities including
  - SSL v3 and TLS v1 with support for the following ciphersuites
    - SSL_RSA_WITH_3DES_EDE_CBC_SHA
    - SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
    - SSL_RSA_WITH_NULL_SHA
    - SSL_RSA_WITH_DES_CBC_SHA
    - SSL_EDH_RSA_WITH_DES_CBC_SHA
    - SSL_EDH_RSA_EXPORT_WITH_DES40_CBC_SHA
    - SSL_RSA_WITH_NULL_MD5
    - SSL_RSA_WITH_NULL_SHA
  - S/MIME signatures with support for the SHA-1, DSA, MD5 and RSA algorithms
  - XML authorization tokens with support for PKCS #7 signatures

- the SCEP protocol used by Cisco routers for PKI operations, using direct client LDAP access for certificate query and CRL access.

The OpenOSP distribution is designed to be suitable both for integration with existing OSS components, and as a base for rapid development of a standalone OSP server product. The distribution is composed of

- a core OSP server protocol stack using defined APIs to interface to the other components required for a full OSP server solution

- sample implementations of the other components, suitable for deploying a standalone OSP server. (Note that these components are developed, tested, and supported to the same standard as the core OSP server code.)

- additional utility programs for certificate creation and revocation to allow an OEM's initial development project to be able to proceed without the need for a separate certificate authority (CA).

The following diagram illustrates the core OSP server stack together with the APIs and the sample implementations of the other components.

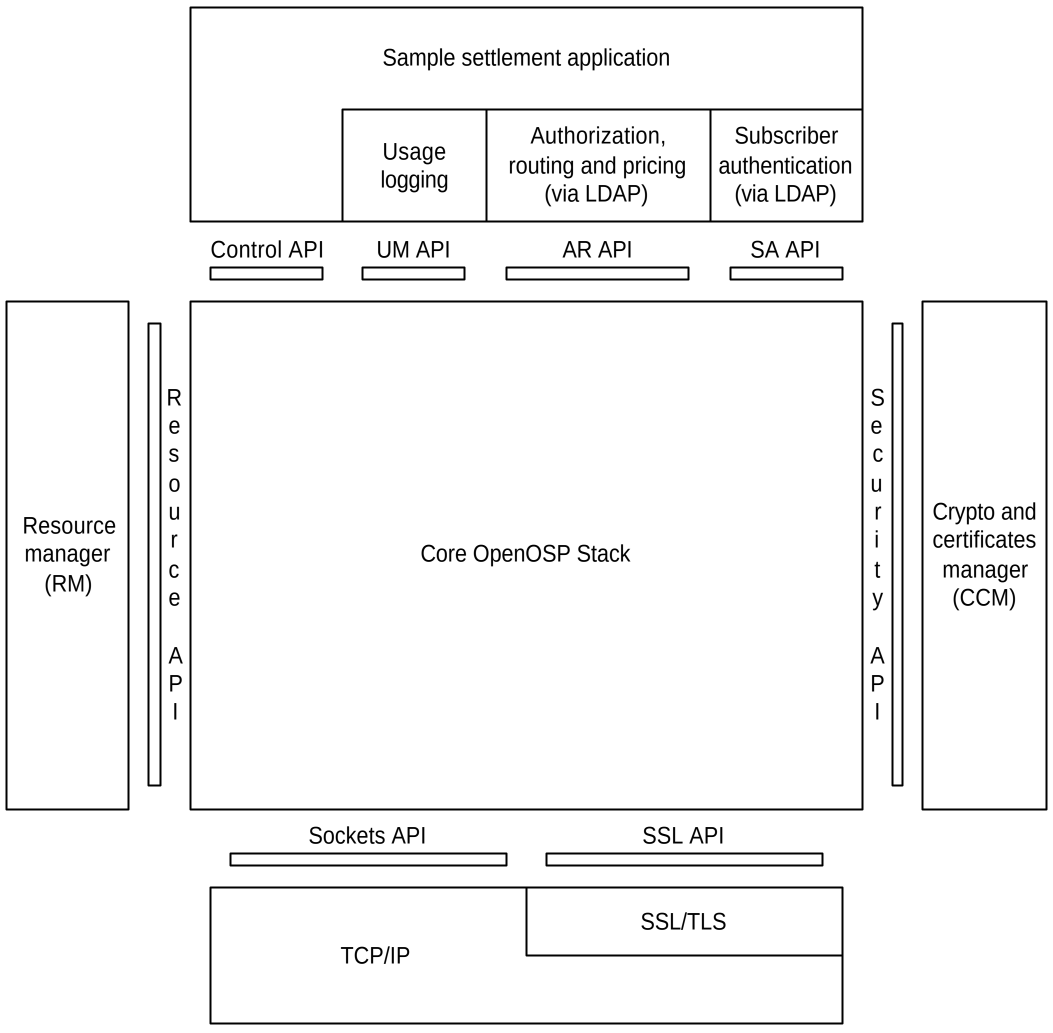
Sample settlement application
Usage logging
Authorization, routing and pricing (via LDAP)
Subscriber authentication (via LDAP)
Control API
UM API
AR API
SA API
Resource manager (RM)
Resource API
Core OpenOSP Stack
Security API
Crypto and certificates manager (CCM)
Sockets API
SSL API
TCP/IP
SSL/TLS

Figure 1 : OpenOSP Server APIs

The following sections describe the system in more detail.

- Section 2.1 gives an overview of the APIs.
- Section 2.2 describes the capabilities of the sample components included in the distribution.
- Section 2.3 describes the scaleability of the system, including how it is engineered for scaleability, performance benchmarks and scaleability characteristics.

Section 3 describes the internal architecture of the OpenOSP server in more detail.

2.1 APIs

2.1.1 Control API

The Control API is provided by the OpenOSP stack to allow the settlement application to

- initialize the stack, start it listening for client connections, and terminate the stack
- obtain performance data about the stack.

The Control API also allows the settlement application to register callbacks which may be used to

- verify a client when it connects
- store received messages for non-repudiation purposes.

If the application registers a client verification callback (and an associated disconnection callback), OpenOSP passes out any certificates received during SSL/TLS negotiation. The application may then return a 'cookie' (that is, some application-specific correlator) which OpenOSP will pass back to the application on any subsequent callbacks, and it may accept or reject the client's certificates. When the client connection is closed, the server calls the application's disconnection callback to allow the application to free any resources associated with the connection.

If no client verification callback is registered, OpenOSP will accept all non-secure connections, and it will accept all secure connections subject to a consistent and within-date certificate chain.

OpenOSP calls the non-repudiation callback with all valid S/MIME-signed OSP requests it receives. This allows the application to store a complete copy of the request.

For further details of the Control API, see the OpenOSP Interface Specification.

2.1.2 Usage Metering API

The Usage Metering (UM) API is a C procedural interface provided by OpenOSP that allows it to pass usage information to the settlement application. The information may be transferred to a permanent storage facility (for later retrieval by an offline billing application) or passed directly to an external usage metering application.

The API is asynchronous; that is, the settlement application may respond to a usage metering callback after the initial callback has returned, and it may do so on a different thread. Alternatively, the application may respond synchronously by calling the response function from within the callback.

For further details of the UM API, see the OpenOSP Interface Specification.

2.1.3 Authorization and Routing API

The Authorization and Routing (AR) API is a C procedural interface provided by OpenOSP that allows the settlement application to authorize calls and generate routes and authorization tokens. The API also allows the application to update pricing information based on OSP pricing indications. Authorization and pricing are grouped into the same API because both sets of functions need to access the routing and pricing database.

The API is asynchronous; that is, responses may be sent by the settlement application after the original callback has returned, and may be sent on a different thread. Alternatively, the application may respond synchronously by calling the response function from within the callback.

For further details of the AR API, see the OpenOSP Interface Specification.

2.1.4 Subscriber Authentication API

The Subscriber Authentication (SA) API is a C procedural interface provided by OpenOSP which allows the settlement application to authenticate subscribers and generate authentication tokens.

The API is asynchronous; that is, responses may be sent by the settlement application after the original callback has returned, and may be sent on a different thread. Alternatively, the application may respond synchronously by calling the response function from within the callback.

For further details of the SA API, see the OpenOSP Interface Specification.

2.1.5 Security API

The Security API is provided by an OEM-replaceable Crypto and Certificates Manager (CCM) component. This API is used by the core OpenOSP stack and the sample application to

- create and verify signatures
- retrieve the server's certificate authority (CA) certificate.

If the OSP server is also acting as a CA for SCEP clients, then the API will also be used to create certificates.

For further details of the Security API, see the OpenOSP Interface Specification.

2.1.6 Resource Manager API

The Resource Manager (RM) API is provided by an OEM-replaceable Resource Manager component. This API is used by the core OpenOSP stack for memory management and to control thread creation and destruction. It treats memory used for data buffers separately from memory used for control blocks and allows reallocation of data buffer memory.

For further details of the RM API, see the OpenOSP Interface Specification.

2.1.7 Sockets and Secure Sockets Layer (SSL) APIs

OpenOSP uses the sockets and SSL APIs to communicate with OSP clients via TCP and/or SSL or TLS.

The sockets API is the standard BSD sockets interface as implemented on Sparc Solaris.

OpenOSP uses a subset of the API defined by the OpenSSL open source SSL/TLS implementation. For further details of the API functions used, see the OpenOSP Interface Specification.

2.2 Sample application

This section describes the sample settlement application which is included in the OpenOSP distribution, and the associated support functions. This application is fully tested to the same standard as the rest of the stack. The reason why it is considered to be 'sample' is that it implements function that an OEM is likely to want to replace when integrating with existing systems for billing, routing, certificate management, etc. The components of the sample application are described below.

2.2.1 Usage Metering

The usage metering component writes a call detail record (CDR) containing the complete contents of each received UsageIndication to a text file. The file format is similar to that used in Annex F of the v2 OSP specification, but is extended to include all fields in the received message (including the enhanced usage parameters defined in Annex C of the specification). The file format is fully documented in Appendix A.

It is expected that an OEM will replace this with code that either interacts dynamically with a settlement back office system, or which writes OEM-specific format CDRs.

2.2.2 Authorization, Routing, Pricing and Capabilities

This component handles authorization, routing, pricing and capabilities exchanges, using an LDAP directory to store information required by the routing algorithm. The key objects in the directory are

- gateways, which have the following attributes
  - an IP address that is used to identify the gateway
  - one or more partial e164 phone numbers indicating the destination numbers that it can reach – these partial numbers may be of any length e.g. 44 for the UK, 4420 for London, 44208366 for Enfield
  - pricing information (in USD) for call services terminated by the gateway for a given destination number.
- service providers, which are hierarchically above the gateways in the directory – each gateway appears in the directory under the service provider that owns it.

The directory also includes information used in subscriber authentication – see the next section for details.

This component handles each of the API primitives as follows.

- On receipt of a capabilities indication, it locates in the directory the gateway to be updated , using the transport address supplied in the DeviceInfo, and then
  - records in the directory
    - whether the gateway is almost-out-of-resources
    - any changes to the protocols supported at the gateway
  - responds with a single URL for all services offered by the gateway, indicating no signature is required on any primitive
  - does not return a device ID.
- On receipt of a pricing indication, it locates in the directory the gateway to be updated and updates the pricing information.
  - The appropriate gateway is identified as follows.
    - If the indication has a supplied source_info of type transport, then this is taken to mean that the price information is for calls using this as the terminating gateway. These indications are used to update the directory.
    - Otherwise the indication is deemed to be for calls using the source as the originating gateway. Since this is not used in the routing algorithm implemented here, such indications are responded to positively but do not update the directory.
  - If the gateway is not found in the directory, or the pricing information is for a route not currently configured in the directory, or the price is not specified in USD, then the pricing indication does not update the directory and is responded to negatively.
- On receipt of an authorization request, the server employs the following algorithm.

    - Find all entries in the directory that match the requested DestinationInfo (only type=e164 is supported), and order them by the following criteria, in the following order (highest priority first)
        - gateways that are not almost out of resources ahead of those that are
        - ones that the client requested ahead of ones that were not requested
        - price to terminate the call, cheapest first
        - quality of match on destination address i.e. a gateway that advertises ‘44208366’ comes ahead of ‘4420’ for matching on destination number ‘442083661177’.
    - This ordered list is then restricted to those in any supplied list of destination alternates, truncated to the maximum number of destinations that the client has requested, a token generated for each, and the routes returned to the client.

      The tokens are always XML format, and are signed (but not encrypted) using the SHA and DSA algorithms via the signing facilities at the security API.
- On receipt of an authorization indication, the server simply checks that the supplied token
    - is correctly signed
    - contains source, destination and call ID information that matches that supplied on the authorization indication.

  It is expected that an OEM will replace this code with a more sophisticated algorithm.
- On receipt of a reauthorization request, the server
    - checks the validity of the supplied authorization token, as for Authorization Indications
    - generates a new authorization based on the supplied parameters, as for an Authorization Request.

2.2.3 Subscriber Authentication

Subscriber Authentication uses an extension to the directory described previously. For each service provider, there may be a list of subscribers configured. Each subscriber entry is identified by a ‘subscriber ID’ attribute.

On receipt of a SubscriberAuthenticationRequest, the sample application

- extracts the subscriber ID from the first SourceAlternate with type=subscriber
- searches the directory for an entry whose subscriber ID matches the subscriber ID extracted from the request

- returns an authentication token if a match is found, and a failure response otherwise. The tokens are in an XML format similar to that used for call authorization tokens, and are signed (but not encrypted) using the signing facilities at the security API.

It is expected that OEMs will replace this in favor of some alternate scheme for remotely querying the private subscriber records of each service provider.

2.2.4 Security Support

The sample crypto and certificates management library is implemented using routines in the OpenSSL distribution. It manages certificates in an LDAP-accessed directory. As such it may be suitable for large-scale deployment if the schema is deemed appropriate for a particular OEM.

If an OEM wishes to replace the sample certificate management, in order to integrate with some existing PKI system, then it should ensure that any certificates registered through SCEP are put into an LDAP-accessible directory, as well as in whatever underlying storage is used for the PKI. Alternatively, the OEM could implement the alternate (but non-preferred) solution of supporting the GetCert SCEP message. Similarly, the OEM should ensure that there is an LDAP-accessible CRL distribution point, or implement the alternate (non-preferred) GetCRL message.

The sample code does nothing with the client verification APIs – it always accepts client connections.

The sample code writes to file any messages passed across the non-repudiation API.

2.2.5 Resource Manager

The sample resource manager implements the most simple policy.

- When resources are requested, it allows the OS to determine whether the request should be satisified (for example, a request for memory is passed directly to malloc()).

- When resources are released, they are released directly to the OS.

OEMs implementing the OpenOSP server on a system that also has other functions may wish to implement some kind of threshold system to prevent the OSP server from consuming all the system's resources.

2.2.6 Secure Sockets Layer

The SSL API is implemented by OpenSSL. This handles SSL and TLS connections, including renegotiation and session ID re-use, as well as providing other security-related routines for the sample crypto and certificates management library.

2.3 Scaleability

The OpenOSP server stack is designed to scale to handle large VoIP networks. This means that the server must be able to support

- a large number of concurrent connections from OSP clients
- an overall high rate of OSP transactions.

The stack is therefore engineered to be as efficient as possible in its processing of transactions and such that the capacity of the system will scale up automatically as the power of the CPU, the number of CPUs and the amount of memory available are increased. In particular, the stack

- has no hard limits on the number of concurrent connections from OSP clients, the number of concurrent transactions, or the size of the routing or subscriber databases
- uses scaleable algorithms and data structures for processing related to OSP clients, transactions, and the routing and subscriber databases
- dynamically allocates and deallocates memory required to handle client connections and process transactions
- uses multiple threads to concurrently process transactions from multiple clients
- can be easily adapted to take advantage of hardware encryption technology when available.

The system is able to process approximately 5,500,000 transactions per hour (over 1500 per second) on a uni-processor 450MHz Ultra-Sparc II system. A transaction is defined as follows.

- Receipt of an AuthorizationRequest and responding with AuthorizationResponse or receipt of a UsageIndication and responding with UsageConfirmation. It is assumed that these messages are comparable in size to the samples in the v2.1.0 OSP specification.
- Code in the usage metering and authorization components of the sample application is not included, as this may vary widely from one OEM to another.
- SSL is enabled, and is using CBC DES encryption and SHA-1 message digests (or similar performance algorithms).
- S/MIME signing is disabled (this is the current normal practice).
- It is also assumed that all other OSP operations are rare compared to the authorization and usage reporting functions (this includes establishing and releasing socket connections and creating certificates for clients).

However, in a final system, the cost of handling each call is likely to be dominated by the cost of the settlement application signing tokens (if this is required). For a software-only system, this cost is significant enough (~40ms on the specified processor) to reduce the overall call rate to around 20-25 calls per second. This compares to around 500 calls per second above (assuming there are typically 3 OSP transactions per call).

Therefore it is recommended that hardware acceleration is employed for token signing. For example, the CryptoSwift product range from Rainbow Technologies (http://isg.rainbow.com/products/cryptoswift.html) includes cards that will support over 1000 signing operations per second, with minimal load on the main CPU. A card supporting 500 signing operations per second would be able to keep up with the processor discussed above.

The authorization component of the sample settlement application is written to allow easy integration with hardware acceleration cards.

2.4 Acknowledgement

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (**http://www.openssl.org/**).

3 System Architecture

The following diagram is an outline of the overall system structure. Each block represents a component of the system.

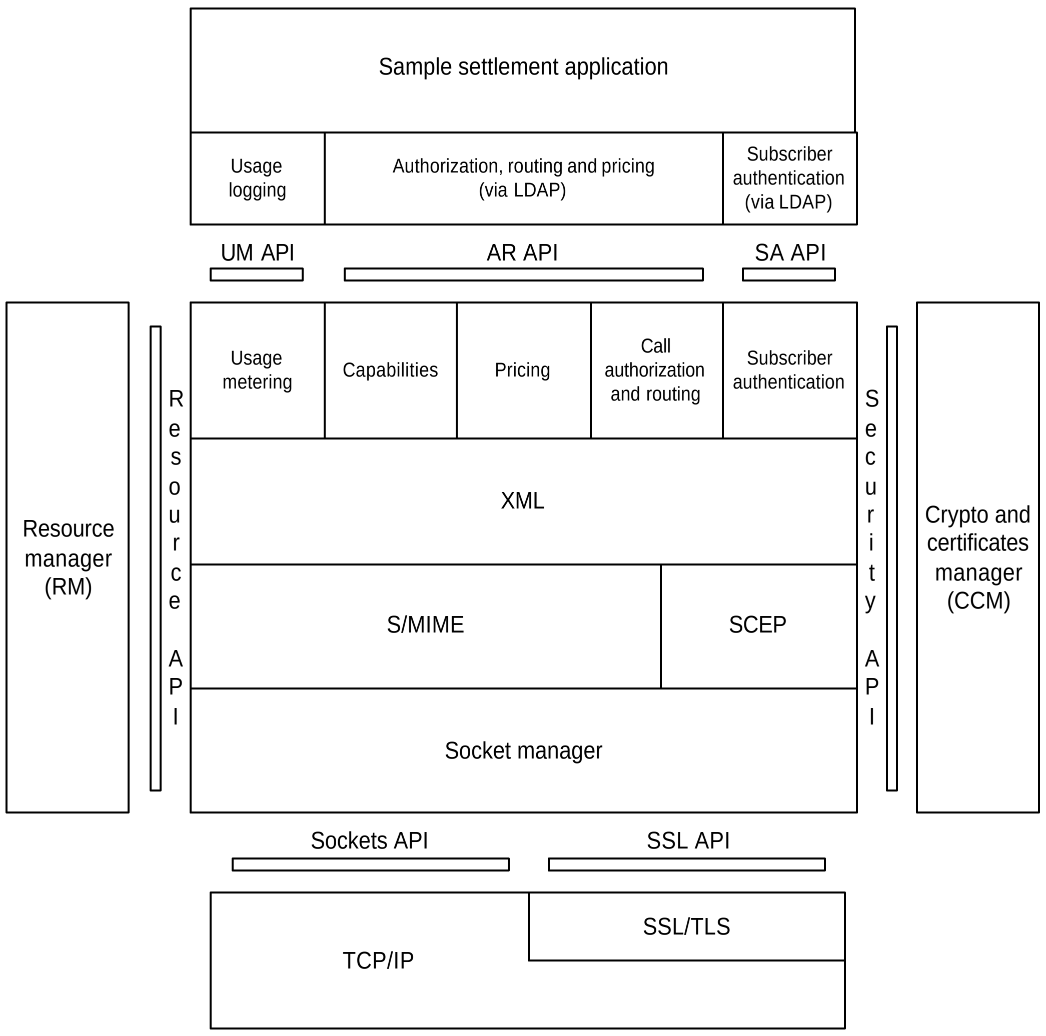
Sample settlement application
Usage logging
Authorization, routing and pricing (via LDAP)
Subscriber authentication (via LDAP)
UM API
AR API
SA API
Resource manager (RM)
Resource API
Usage metering
Capabilities
Pricing
Call authorization and routing
Subscriber authentication
XML
S/MIME
SCEP
Socket manager
Security API
Crypto and certificates manager (CCM)
Sockets API
SSL API
TCP/IP
SSL/TLS

Figure 2 : OpenOSP Architecture

The components of the system are as follows.

- Socket manager. This component drives the sockets and SSL interfaces and implements the limited subset of HTTP required for OSP. By default, it accepts calls on port 80 for non-secure connections (in which case SSL will not be used) and port 443 for secure connections. This code also includes the overall scheduling code for the OSP server, since it is here that the server waits for incoming work.

- S/MIME. This component verifies S/MIME-signed messages received from OSP clients and signs outgoing responses if required. It supports the SHA-1, DSA, MD5 and RSA algorithms mandated and recommended by the OSP specification. It uses the signing and verification functions available at the security API.

- SCEP. This component implements the SCEP protocol used by Cisco clients that use their OSP server to get access to PKI facilities.

- XML. This component implements the XML parsing required to decompose OSP messages. It uses the expat open source XML parsing library (http://www.jclark.com/xml/expat.html). The Binary XML content format (defined in Annex H of the OSP specification) is not supported.

- Capabilities. This component processes OSP capabilities negotiation requests from the client.

- Usage metering. This component processes usage metering commands from OSP clients. It uses the S/MIME component to authenticate received commands and to sign responses. It passes the usage information across the Usage Metering API.

- Pricing. This component processes pricing information commands from OSP clients. As with the usage metering application, it uses the S/MIME component to authenticate received commands and sign responses.

- Call authorization and routing. This component processes call authorization/reauthorization requests and indications from clients. It uses the S/MIME component to authenticate received requests and indications and sign responses and confirmations.

- Subscriber authentication. This component processes subscriber authentication requests from an OSP client. It uses the S/MIME component to authenticate received requests and sign responses.

- The sample settlement application is described in section 2.2.

The operation of all components of the system is controlled through a plain text configuration file.

The system traces all signifcant events and any errors to aid monitoring and diagnosis of any problems. A debugging target may also be built which produces extremely detailed trace at all stages of processing.

Appendix A: Usage record format

The sample application's usage record file contains one record per line, with the fields that make each record occupying fixed width columns.

This format is similar to that produced by the Java code in Annex F of [OSP], but necessarily differs because

- that format includes several fields that are irrelevant to the OSP protocol
- that format omits fields for many of the items of usage information that the sample application should record.

Field widths, contents and formats are specified by the following table.

Field width	Field contents	Field format
20	Timestamp	XML data
3	Role	"SRC", "DST", "OTH"
14	TransactionID	XML data
14	CallID	"C:" (for CDATA encoding) or "B:" (for BASE64 encoding) followed by XML data, with non-printing characters replaced by %XX (as in URLs).
4	SourceInfo type	Four letter type abbreviation
16	SourceInfo data	XML data
4	SourceAlternate type	Four letter type abbreviation
16	SourceAlternate data	XML data
4	DestinationInfo type	Four letter type abbreviation
16	DestinationInfo data	XML data
4	DestinationAlternate type	Four letter type abbreviation
16	DestinationAlternate data	XML data
4	Amount	XML data
4	Increment	XML data
4	Unit	XML data

20	StartTime	XML data
20	EndTime	XML data
4	TerminationCause TCCode	XML data
16	TerminationCause Description	XML data
4	LossSent Packets	XML data
4	LossSent Fraction	XML data
4	LossReceived Packets	XML data
4	LossReceived Fraction	XML data
4	OneWayDelay Minimum	XML data
4	OneWayDelay Mean	XML data
4	OneWayDelay Variance	XML data
4	OneWayDelay Samples	XML data
4	RoundTripDelay Minimum	XML data
4	RoundTripDelay Mean	XML data
4	RoundTripDelay Variance	XML data
4	RoundTripDelay Samples	XML data
15	Client's IP address	Dotted decimal format

Notes.

1. Each line is terminated by a single linefeed character (ASCII code 10).

2. Fields are separated by a single space. Hence, for example, the role field begins in column 22.

3. Erroneous or absent values are indicated by filling the field with dashes.

4. Truncated values are indicated by replacing the last three field columns with full stops.

5. The CallID, SourceInfo, SourceAlternate, DestinationInfo and DestinationAlternate parameters are assumed to consist of a single list element. Further list elements are ignored.

6. If the UsageIndication contains more than one UsageDetail element, the sample application generates a separate usage record for each UsageDetail. The usage records thus generated have the same values in their non-UsageDetail fields.

References

The following references provide further information on relevant subjects:

OSP	Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON); Open Settlement Protocol (OSP) for Inter-Domain pricing, authorization and usage exchange. ETSI TS 101 321 V2.1.0 (2000-05).
SCEP	Cisco Systems' Simple Certificate Enrollment Protocol (SCEP) Cisco Systems, August 2000 http://search.ietf.org/internet-drafts/draft-nourse-scep-03.txt
INT	OpenOSP Interface Specification, version 1.0 Data Connection Limited, August 2000 MOM-0101-0100
SSL	The SSL Protocol Version 3.0 Netscape Communications Corporation, November 1996 http://oem.netscape.com/eng/ssl3/draft302.txt
TLS	RFC 2246: The TLS Protocol Version 1.0 T. Dierks and C. Allen, January 1999 http://www.ietf.org/rfc/rfc2246.txt
HTTP/1.0	RFC 1945: Hypertext Transfer Protocol – HTTP/1.0 T. Berners Lee, R. Fielding and H. Frystyk, May 1996 http://www.ietf.org/rfc/rfc1945.txt
HTTP/1.1	RFC 2616: Hypertext Transfer Protocol – HTTP/1.1 R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, June 1999 http://www.ietf.org/rfc/rfc2616.txt
S/MIME	RFC 2311: S/MIME Version 2 Message Specification S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade and L. Repka, March 1998 http://www.ietf.org/rfc/rfc2311.txt
XML	Extensible Markup Language (XML) 1.0 W3C, February 1998 http://www.w3.org/TR/REC-xml
PKCS #7	PKCS #7: Cryptographic Message Syntax Standard (version 1.5) RSA Laboratories, November 1993 http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/index.html
PKCS #10	PKCS #10: Certification Request Syntax Standard (version 1.0) RSA Laboratories, November 1993 http://www.rsasecurity.com/rsalabs/pkcs/pkcs-10/index.html