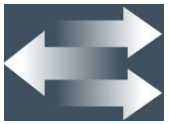
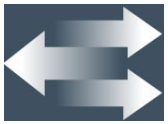
Resiprocate SIP Stack

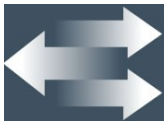


Contents

- Resiprocate Architecture
- Using Resiprocate Stack
- Using Resiprocate DUM
- Resiprocate Code Overview



Architecture Overview of Resiprocate SIP Stack

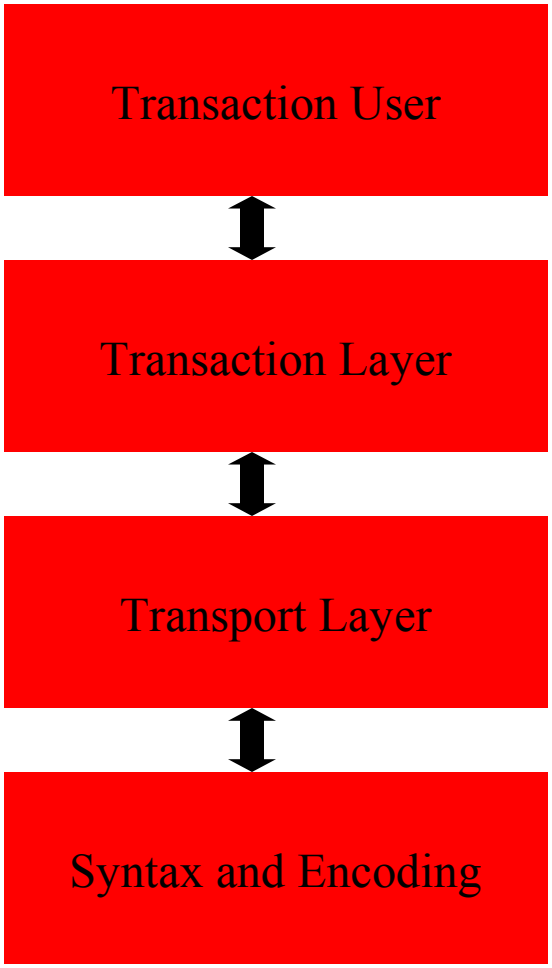


ReSIProcate Stack

- ReSIProcate is an object oriented SIP interface and stack implemented in C++. The ReSIProcate approach emphasizes consistency, type safety, and ease of use.

- The key goals are:
  - 3261 compliance
  - easy to program with
  - efficient (> 1000 transactions per second) (Platform Dependent)
  - excellent security implementation including TLS and S/MIME
  - Win32 and Linux/Unix support
  - usable by proxies, user agents and b2buas
  - object oriented interface

SIP Stack Architecture (RFC 3261)

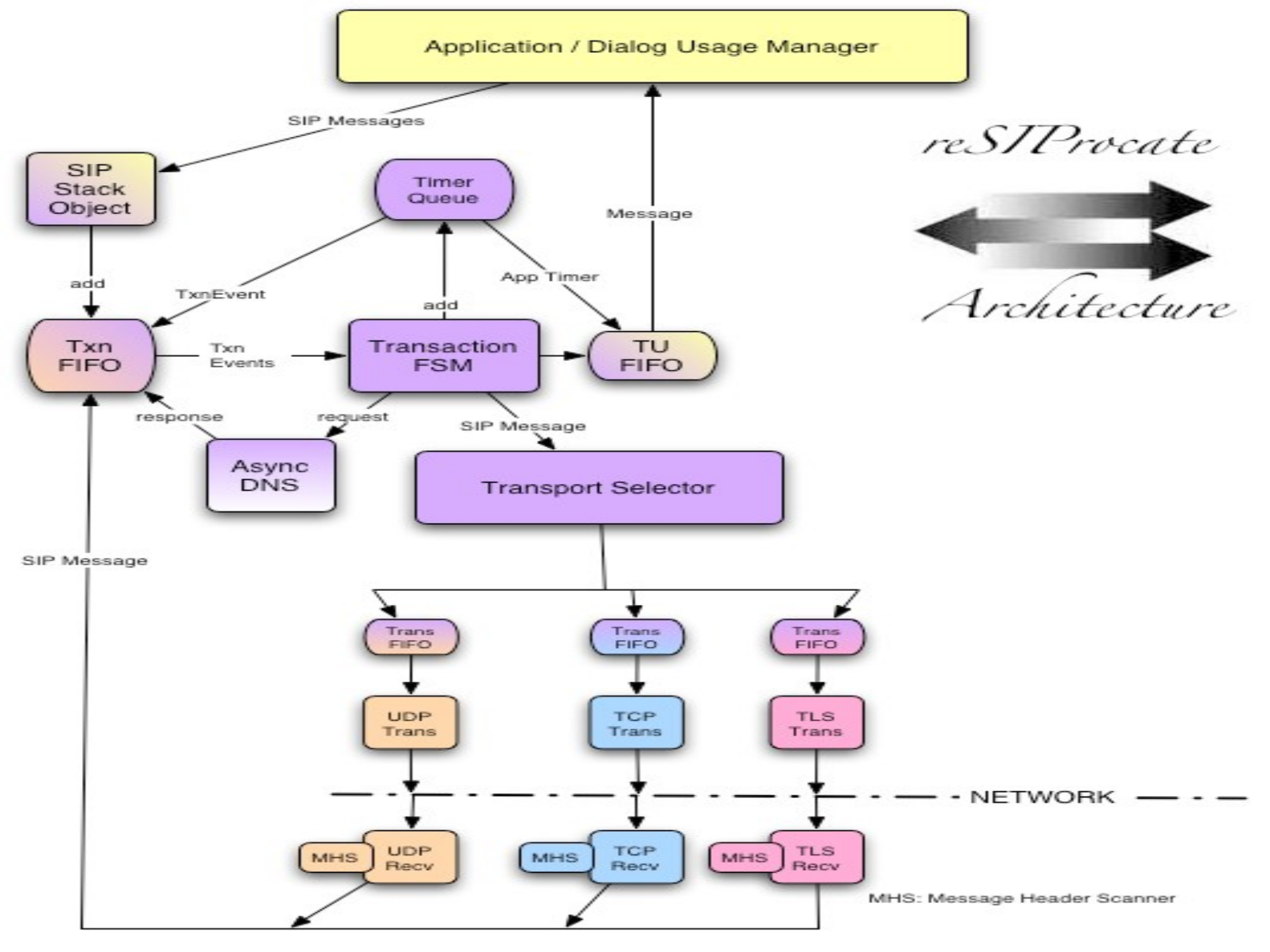


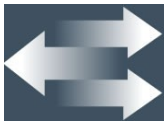
Each of the SIP entities, except the stateless proxy, is a transaction user.

The transaction layer handles application-layer retransmissions, matching of responses to requests, and application-layer timeouts. The transaction layer has a client component (client transaction) and a server component (server transaction),

Defines how a client/server sends/receives requests & receives responses

Augmented Backus-Naur Form grammar (BNF)


Application / Dialog Usage Manager
SIP Messages
SIP Stack Object
Timer Queue
Message
reSIProcate
Architecture
add
TxnEvent
add
App Timer
Txn FIFO
Txn Events
Transaction FSM
TU FIFO
response
request
SIP Message
Async DNS
Transport Selector
SIP Message
Trans FIFO
Trans FIFO
Trans FIFO
UDP Trans
TCP Trans
TLS Trans
NETWORK
MHS
UDP Recv
MHS
TCP Recv
MHS
TLS Recv
MHS: Message Header Scanner

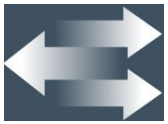


Resiprocate Components

- **Entities**
  - Transaction User
  - Sip Stack Objects
  - Transaction FSM
  - Transport Selector
  - UDP/TCP/TLS Transport
  - Message Header Scanner (MHS)
  - Async DNS Utility

- **FIFO**
  - TU FIFO
  - Txn FIFO
  - Trans FIFO

- **Timer Queue**

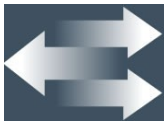


Transaction User

- To use the SIP Stack Transaction User has to be implemented.
- Dialog User Manager (DUM) is a Transaction User.
- DUM provides User Agent Functionality including handling of INVITE and SUBSCRIBE/NOTIFY dialogs, registration and instant messaging
- By using Transaction User Interface we can build applications like JSR 180, Customized B2BUA
- Mainly Provides Interface for Sending and Receiving SIP Messages

SIP Stack Object

- Interface for accessing the SIP Stack functionalities
- Provides APIs for
  - Management of SIP Stack
  - Register/Unregister TUs
  - Adding new Transports
  - Sending/Receiving SIP Messages

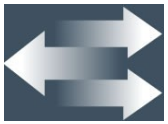


Transaction FSM

- Implements the SIP Stack Transaction State Machine
- Resolves the Domain Names for Routing

Transport Selector

- The Transport Selector is primarily responsible for:
  - Determining the fully-specified destination for an outgoing SipMessage, given a (possibly) partially-specified destination.
  - Deciding which instance of class Transport the outgoing SipMessage is to be sent on.
  - If necessary, filling out any fields in the SipMessage that depend on 1) (ie, the host in the topmost Via, unspecified host parts in Contact header-field-values that refer to this UA, Record-Route header-field-values that depend on which transport the message is being sent on, etc)
  - Serializing the outgoing SipMessage, and passing the serialized form to the Transport instance chosen in second point.



UDP/TCP/TLS Transport

- Sends/Receives the SIP Message to Network

MHS

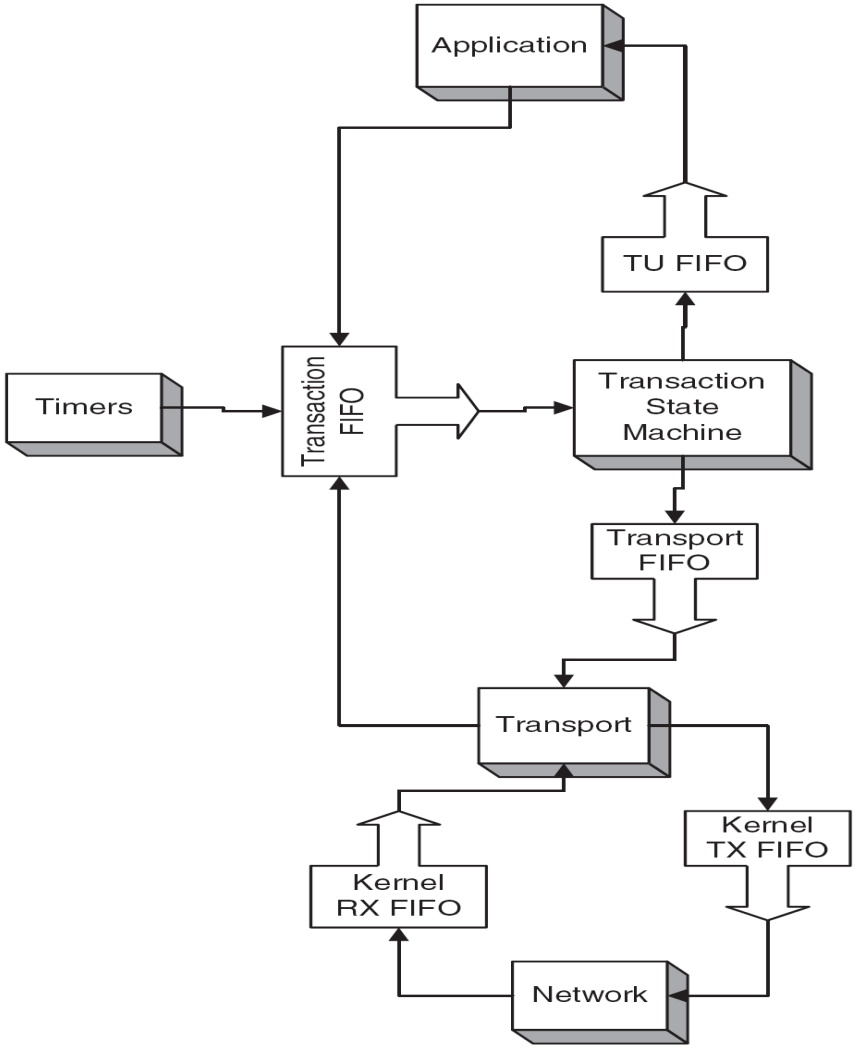
- SipMessage parsing happens in phases. MsgHeaderScanner performs the first phase of parsing. MsgHeaderScanner is called from a transport. Framing of the message is an interaction between the transport and the scanner. The transport feeds the scanner consecutive chunks and the scanner reports when it has framed a message.

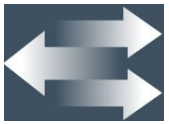
- The chunks containing the memory are handed back to the SipMessage for eventual deallocation. In addition, the scanner lexes the message and identifies the boundaries of header field values.

- Each header field value is parsed on demand. The instance of parser that is created for the header field value is determined by the header type accessed in the message

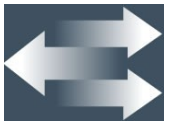
FIFO

- FIFOs also known as queues, are a standard thread synchronization/buffering mechanism. FIFOs are the main way to move information between threads within reSIProcate. Since SIP is a message/event protocol, event FIFOs are a natural and error resistant mechanism for communicating among threads.
- There is a FIFO between every layer of reSIProcate.
- The following are the FIFOs present in Resiprocate
  - between the Transaction State Machine and the application (TU FIFO)
  - between the timers and the transaction state machine (transaction FIFO)
  - between the transaction state machine and the transports
  - between the transport and the network (kernel TX FIFO)
  - between the network and the transports (kernel RX FIFO)
- The actual FIFO is a template class (abstractFifo.hxx). The getNext method will wait until there is something in the FIFO.



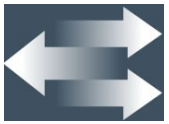


Using Resiprocate Stack



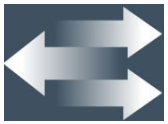
Using Resiprocate Stack

- Transaction User Management
- SIP Stack Management
- SIP Message and Headers



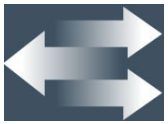
Transaction User Management

- To Create a new Transaction User the following needs to be done
  - Implement TransactionUser Interface
    - name() - New Transaction User Name
  - Register Transaction User with the stack
- To send the SIP Message SIP Stack Object should be used and new Transaction User Reference should be passed.
- The Transaction User Class contains TU FIFO through which we can get the SIP Messages.
- Code Snippet



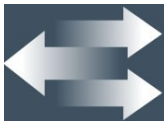
SIP Stack Management

- The SIP Stack can be created as an Object inside the User created class
- Sip Stack Object provides Transaction Level APIs
- The following are the common APIs used
  - sipstack() Creates the SIP Stack
  - addTransport() adds Listening port
  - send() API call is used to send SIP Messages
  - buildFdSet() API builds the fdset of the Transport.
  - process() API call runs the SIP Stack. So it needs to be called periodically.
  - StackThread class can be used for doing the above functionalities.
  - The details of the APIs are present in the following location
  - Shutdown() API Call can be used to stop the stack



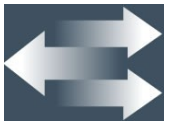
Construction of SIP Message

- The Sip Message may be constructed by using Helper class present in the stack.
- The Helper Class provides static function calls
- The some of common API used
  - makeRequest()
  - makeResponse()
  - makeInvite()
  - makeCancel()
  - makeRegister()
  - makeSubscribe()
  - makeMessage()
  - makePublish()
  - …
- Code Snippet



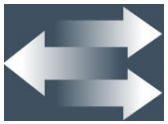
Sip Message and Headers

- A central component of any SIP service is handling of SIP messages and their parts. SIP Messages are handled by SipMessage Class
- A SIP message consists of
  - Headers
  - Request/status line
  - Body.
- If we want to access/modify the headers, request/status line and body we have to use SipMessage class.
- SIP Headers can be grouped into two types
  - Single Instance
  - Multiple Instance
- SIP Headers may also consist of parameters
- Request Line and Status Line depend on the type of Message we have received that is whether Request or Response
- Body is the contents that is carried by SIP Message. For Example SDP.



Headers

- Resiprocate Stack provides a uniform way of accessing SIP Headers. To access the headers following procedure has to be followed
  - Header - RFC 3261- **header access token**. For From Header – Token in “From”
  - In Resiporcate to access prefix “h_”. For From Header – h_From
  - For Multiple Instance header “s” will be appended to the RFC name.
  - For Example if RFC Name is Record Route then header access token is h_RecordRoutes
  - header() overloaded method of SipMessage is used to access the header and assign value.
  - remove() overloaded method is used to remove the header
  - exists() overloaded method is used check the existence of the header
  - The Multiple Headers are accessed by stl fashion iteration.
- **Parameters**
  - Parameters are accessed from headers.
  - p_ should be used
  - Example: const Data& tag = msg->header(h_To).param(p_tag); to access To tag
  - param() overloaded method of SipMessage is used to access the header and assign value.
  - remove() overloaded method is used to remove the parameter
  - exists() overloaded method is used check the existence of the parameter

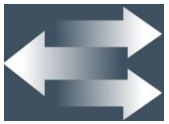


Request/Status Line

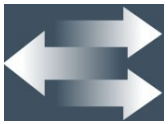
- Special Types of Headers
- Accessed by h_RequestLine and h_StatusLine
- isRequest method and isResponse method
- RequestLine provides APIs like method(), uri(), getSipVersion()
- StatusLine provides responseCode(), statusCode(), getSipVersion()

Sip Message Body

- getContents method returns Contents type
- Cast the Contents type to the required Content type found in Content-Type Header
- For Setting content type use setContents of SipMessage
- We can add new Content types and Parser without any modification to reSIP library.



Using Resiprocate DUM

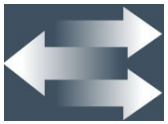


Dialog Usage Manager - DUM

- The Dialog Usage Manager makes writing user agents easy by hiding complex SIP specifics. DUM provides user agent functionality including the handling of INVITE and SUBSCRIBE/NOTIFY dialogs, registration, and instant messaging. With DUM we can create applications like softphones, back-to-back user agents, and load generators.

- DUM implements Transaction User

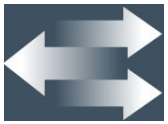
- DUM does the following functions
  - Implements Offer/Answer
  - Manages Profiles
  - Manages AppDialogSetFactory (Equivalent to Call)
  - Manages and stores Handlers, which are a way of referencing usages
  - Manages redirections (RedirectManager)
  - Manages client authentication (ClientAuthManager)
  - Manages server authentication (ServerAuthManager)
  - Interface to add new Transports
  - Manages handles to Usages (HandleManager)
  - Provides interfaces to create new sessions as a UAC (invite, subscription, publication, registration, pager, others)
  - Provides interfaces to find particular usages based on a DialogId.



DUM Application

- **DUM Appliaction**
  - Create Stack
  - Create DUM
  - Add Transports
  - Create Profile
  - Set Profile Options
  - Set Handlers
  - Start Process Loop

  - Application should implement Handlers
  - DUM provides API for constructing SipMessage
  - Handle are used to send/receive SipMessages within a Dialog



Dialog Usages

- When Dialog is created they establish an association between endpoints within the Dialog. This association is known as Dialog Usage (http://www.ietf.org/internet-drafts/draft-ietf-sipping-dialogusage-06.txt)
  - A Dialog initiated by INVITE Request has an INVITE Usage
  - A Dialog initiated by SUBSCRIBE has an SUBSCRIBE Usage

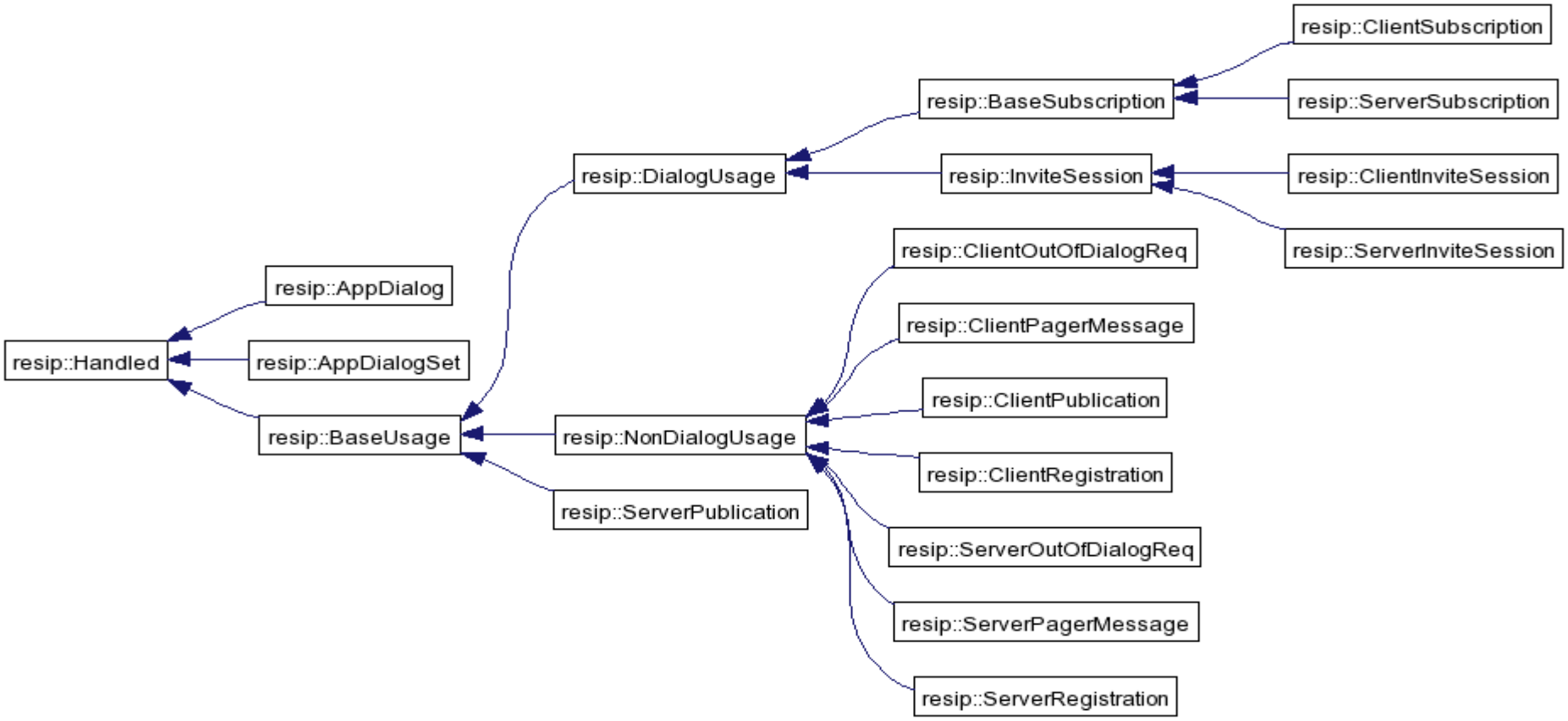
- In Resiprocate Usage concept is used. There are two types of Usages
  1. DialogUsage
  2. NonDialogUsage

  The Usage provides the APIs that are required to send/receive Requests/Responses inside a SIP Dialog.

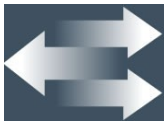
- Some Example of Usages are
  1. ClientRegistration
  2. InviteSession
  3. …

- Some of the API examples in InviteSession are as follows
  1. info()
  2. Refer()
  3. …

Dialog Usage Inheritance



- How to use Dialog Usage?



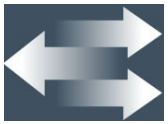
DUM Handles

- Handles are used to access the Usage Object.
- Usages are derived from Handled Class. Handles can point to objects which subclass Handled.
  - Reasons for Accessing Usages through Handles are
  - Usages might get deleted by the time application uses it
  - Once created Handle will continue to exists even if Handled Object gets deleted. So it can throw exceptions
- Smart Pointer
  - Keeps track of referenced Object
  - Throws exception if not found

  Handle point to InviteSessions, ClientRegistration, ClientSubscription …
- Code Example

DUM HandleManager

- HandleManager keeps track of Handles.
- The Reference of Handles are stored in HandleManager
- DialogUsageManager is subclass of HandleManager
- This the way the DialogUsageManager and Handles (Usages) are linked.
- DialogUsageManager keeps track of Usages



Handlers

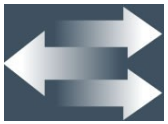
- Handlers are essentially Callbacks needs to be implemented by the application.
- The Callbacks are called from SipStack when a Response or New Requests in the context of the Usages are received.
- The Handlers usually return
  - Handles to Usages
  - The SipMessage received
- Example of a Handler is InviteSessionHandler. Some of the Callback APIs are
  - virtual void onNewSession(ClientInviteSessionHandle, InviteSession::OfferAnswerType oat, const SipMessage& msg)=0;
  - virtual void onNewSession(ServerInviteSessionHandle, InviteSession::OfferAnswerType oat, const SipMessage& msg)=0;
- Application should implement the InviteSessionHandler class and the APIs
- setHandler method in DialogUsageManager should be used to set the Handlers.

Profiles

- Profiles are used to set the properties of User Agent like Outbound Proxy, User Name, User Authentication parameters, etc
- The Profile Hierarchy as shown

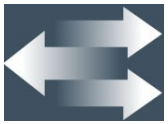
- Profile is base class. Its as various settable properties such as RegistrationTime, MaxRegistrationTime, SubscriptionTime, outBoundProxy, etc.
- Three types of APIs
  - **setXXXX**
  - **getXXXX**
  - **unsetXXX**

- If a property is not set then the default value in BaseProfile is taken.
- The User Profiles handles the User related configurations suncg as AOR, Credentials etc
- MasterProfile handles SIP Capability related configuratiions such as method types, methods, options, mime types etc
- There are two models of Profile settings
  - Single Profile
  - Mutli Profile



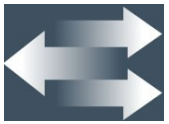
Dialogs

- **DialogSet**
  - Container class holding a set of dialogs initiated from a common requests
  - Share the same Call-ID and the same from tag in the request that generated the dialog
- **Dialog**
  - Container class holding SIP RFC dialog details
- **AppDialog**
  - An Application can associate user data with Dialog/DialogSet AppDialog, AppDialogSet and AppDialogSetFactory classes
  - This type of association may be required when there are multiple dialogs the Application has to track.

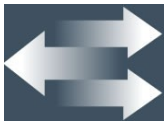


Logger Module

- The Logger Module can be used for debugging purpose
- Log::initialize() call used to initialize the Log Module
- The following options can be used to print the Logs
  - Cout – Prints on Standard Output
  - Syslog – Prints to Syslog
  - File – Prints to User Given File or resiprocate.log
  - Cerr – Prints to Standard Error Output
- The Log Levels that can be set are as follows
  - CRIT
  - Err
  - Warning
  - Info
  - Debug
  - Stack
  - StdErr

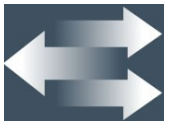


Code Structure

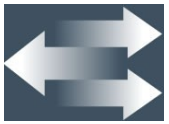


Code Structure

- **rutil**
  - Various protocol-independent utility classes, used by all the other modules.
- **Stack**
  - -Core SIP stack functionality, including message parsing, message synthesis, and transaction handling.
- **Dum**
  - User Agent functionality, including handling of INVITE and SUBSCRIBE/NOTIFY dialogs, registration, and instant messaging.
- **repro**
  - Flexible SIP proxy framework.
- API Definitions can be found at **http://www.estacado.net/resip-dox/**



Q & A



Backup Slides

ReSIProcate Stack Architecture