Programmer's Guide

M5T SIP SAFE Version 4.1

Proprietary & Confidential

This page is left intentionally blank.

Publication History

Release	Date	Description
1	08/07/2005	Pre-release, work in progress, most key concepts detailed.
2	20/01/2006	Pre-release, cleaned up TBDs, reviewed.
3	01/02/2006	Pre-release, reworked general presentation
4	22/06/2006	Added Getting Started section for Proxies.
5	06/09/2006	Sip-UA 4.1 release.
6	25/10/2006	Added the IPv6, Local Address Management, Listening Mechanism, Routing Table, Via Address Preference and SIP Proxy Context sections. Merged the v4.0 and v4.1 documents.
7	17/12/2007	Added sequence diagrams.
		Added section “Releasing a SIP Context”
		Added section “Getting started with User Agents”
		Added section “Security considerations”
		Added section “Connection Blacklist service”
8	21/01/2008	Added section “Context lifetime with Stateless Proxy Service”
		Added section “Context lifetime with Transaction Stateful Proxy Service”
		Added section “Context lifetime with Session Stateful Proxy Service”
		Added section “Key concept: Spiralling service”
		Added section “Key concept: Session stateful proxy service forking”
9	28/10/2008	Added section “Key concept: Outbound Connection Service
10	12/12/2008	Added the services ordering section.
11	15/12/2008	Added a new section on persistent connections.
		Added new section on forking

List of Acronyms

ABNF	Augmented Backus-Naur Form
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
BLOB	Binary Large Object
DNS	Domain Name Server
DNS SRV	Domain Name Server Service Record
IETF	Internet Engineering Task Force
IM	Instant Messaging
M5T	M5T (brand name)
NAT	Network Address Translator
PA	Presence Agent
PUA	Presence User Agent
RFC	Request for Comment
SDP	Session Description Protocol
SIP	Session Initiation Protocol
TCP	Transport Control Protocol
TLS	Transport Layer Security
UA	User Agent
UDP	User Datagram Protocol
URI	Uniform Resource Identifier/Universal Resource Identifier
URL	Uniform Resource Locator
UUID	Universal Unique Identifier

Terms and Definitions

Key words 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.

Status

Work in Progress: An incomplete document, designed to guide discussion and generate feedback, which can include several alternative requirements for consideration.
Draft: A document in specification format considered largely complete, but lacking review. Drafts are subject to substantial changes during the review process.
Interim: A document that has undergone rigorous review.
Released: A stable document, reviewed, tested and validated.

List of Figures and Tables

List of Figures

List of Tables

About this document

Welcome to the M5T SIP SAFE Programmer's Guide.

Table 1 - Text and Symbol Conventions

Convention	Description
`Courier`	• Sample code • Manager • Interface
Bold	• Request, Response • Field • Package • Header • Method
Italics	• Scenario • Special information
ATTENTION!	Indicates important information about the current topic.
CAUTION!	Indicates a potentially hazardous situation which, if not avoided, may result in damage to the equipment or loss of data.

Table of Contents

1. Introduction to M5T SIP SAFE

M5T SIP SAFE v4.1 is the latest RFC 3261-based C++ SIP stack generation that can be used to create standard and advanced secure SIP devices, servers and applications. Bearing M5T's security strategy tag "SAFE", it offers built-in security and advanced functionality to help build feature-rich, yet highly secure SIP products.

M5T SIP SAFE offers much more than other SIP stacks. Where other stacks offer parsers, transport and transaction management layers with some helpers for managing dialogs, M5T SIP SAFE offers all this along with numerous SIP-based modular services that can be used within the provided high-level SIP framework.

M5T SIP SAFE is ideally suited for intelligent SIP devices, small or large, such as in the following applications:

- IP Phones
- PC Clients
- CPEs and Gateways
- Mobile, Wireless and Wi-Fi devices
- PDAs
- Systems on a chip
- Applications for collaboration
- Proxy servers
- Redirect servers
- Registrar servers
- Application servers
- Conferencing servers
- ... And any other kind of SIP system.

M5T SIP SAFE is developed over the M5T Framework as with all other products developed at M5T. The M5T Framework is a suite of tools, algorithms and patterns that serves as the foundation for building advanced products. The M5T Framework abstracts the operating system and the network access, as well as simplifies threading for all M5T products. It can also be directly accessed and used as a foundation for portable, multithreaded applications.

This document provides programmers with the necessary information to start using M5T SIP SAFE's API, which is described in the companion document "*M5T SIP SAFE v4.1 - API Reference.pdf*".

1.1 M5T SIP SAFE High-Level Architecture

M5T SIP SAFE is separated into a set of packages, with each package having its own logical boundary or set of responsibilities. Each package has its own directory where source and header files are located, with the directory name being the same as the package's name.

Figure 1 illustrates a package view of M5T SIP SAFE. The dotted arrows represent dependency from one package to another, with the exception of the SipParser package, from which all other packages depend. The following sections will provide a brief overview of each M5T SIP SAFE package.

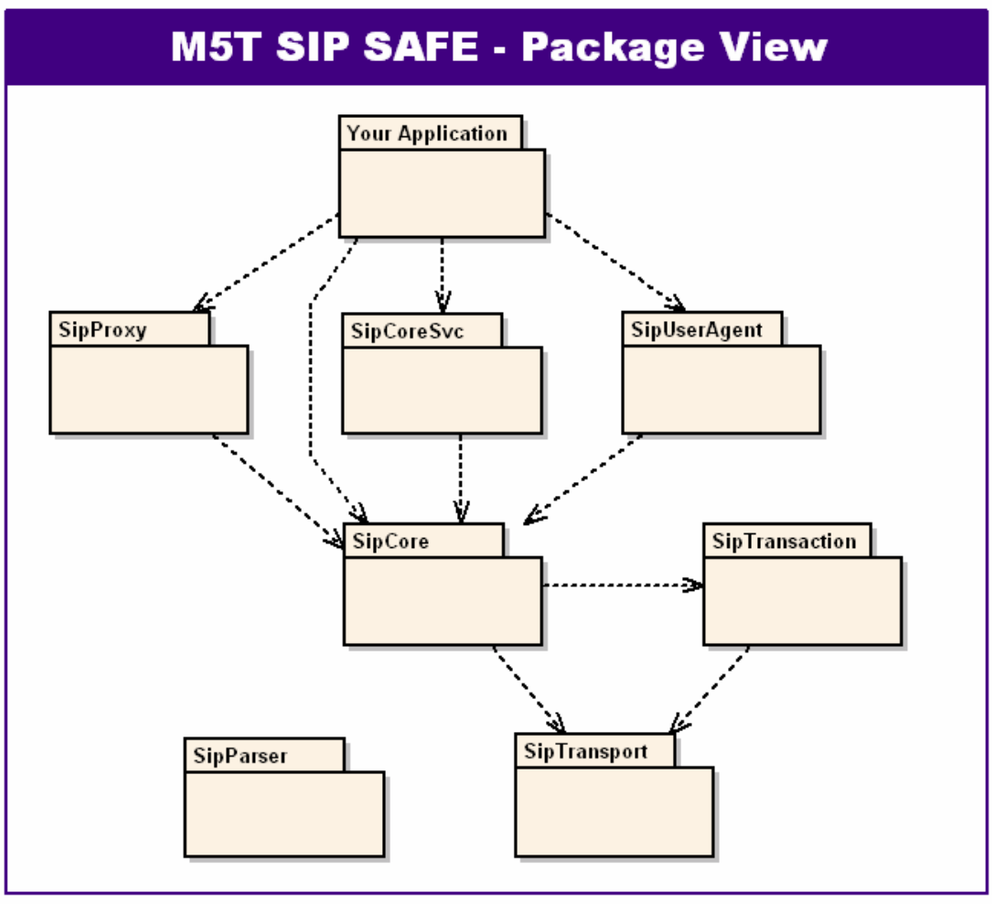


Figure 1: M5T SIP SAFE Package View

Although the M5T Framework is not depicted in Figure 1, it is important to understand that all M5T SIP SAFE packages depend on the M5T Framework. The M5T Framework has its own documentation.

1.1.1 SipParser

At a high-level, the SipParser package defines a text-based, two-pass, on-demand parser, with each class being able to parse, generate and hold SIP data.

It is important to understand the responsibilities of each parser class found in the SipParser package:

Table 2 - Parser Classes

Parser Class	Description
Decoder	Each SipParser class can parse (or decode) a const char* buffer according to the ABNF the class represents. The stack uses it when receiving an incoming SIP packet from the network, but your application can also use it to parse information entered through a user-interface or any other mechanism.
Container	Each SipParser class can act as a container, holding the value that it has just parsed, or it can also hold values that were set through its interface.
Encoder	Each SipParser class can serialize or write the values it contains into a CBlob object. The values are written according to the ABNF the class represents. The stack uses it when generating packets to be sent on the network and your application can also use it to generate user-information or tracing.

The various SipParser classes are used throughout the stack to manipulate SIP information such as URLs, name-addr constructs, and contact headers.

1.1.2 SipTransport

The SipTransport package is responsible for the actual communication with the IP network. It abstracts the type of transport used for sending the packet and manages connections with other peers.

When receiving a SIP packet, the SipTransport uses the SipParser package for decoding the received data into a CSipPacket instance, which can then transit up in the stack for further processing. The same is true when sending a SIP packet; the SipTransport uses the CSipPacket instance to send for generating the proper data to be sent on the network, which represents a well-formatted SIP packet.

The SipTransport manages established connections with other peers, which means that after connecting to a peer and sending a SIP packet, the SipTransport keeps the connection up for future packets that may be sent to the same peer. This connection management mechanism can easily be configured to suit the application's need.

1.1.3 SipTransaction

The SipTransaction package manages stateful SIP transactions. More specifically, it manages the retransmissions and timeouts of requests and responses, and also detects and manages incoming retransmissions.

The SipTransaction uses the SipParser to access the information within a SIP packet, and also uses the SipTransport to send and receive SIP packets.

1.1.4 SipCore

The SipCore package is an abstract package that mainly defines the plug-in mechanism used by the high-level services. It uses:

- the SipParser for setting and accessing information within SIP packets;
- the SipTransaction package when a service wants to create and send a packet within a transaction;
- the SipTransport when a service must send a packet without creating a transaction.

The SipCore package defines the event management mechanism, which in turn defines the events reported to the application and their order.

The application interacts directly with the SipCore through a set of interfaces that are received from the services used by the application.

1.1.5 SipCoreSvc

The SipCoreSvc package holds modular services the application can use when performing various actions that can impact the processing of SIP packets. These services are not tied to the proxy or UA functionality and thus can be used with both the UA and proxy services.

1.1.6 SipUserAgent

The SipUserAgent package holds all the modular user agent related services. Although called user agent services, these services are useful for all devices that must act as a user agent, meaning devices that have to generate their own requests and responses, instead of forwarding them.

These services can be used to build devices like IP Phones, B2B-UAs, conferencing servers, application servers, etc.

The SipUserAgent package uses the SipCore package to access the service plug-in architecture, and the SipParser to set and access the information found within a SIP packet.

1.1.7 SipProxy

The SipProxy package holds all the modular SIP proxy related services. These services mainly allow applications to forward SIP packets, either statelessly or statefully.

The SipProxy package uses the SipCore package to access the service plug-in architecture and the SipParser to set and access the information found within a SIP packet.

1.1.8 SdpParser

The SdpParser package is not depicted in Figure 1, as M5T SIP SAFE is completely dissociated from it. It is provided as a helper package for applications that must handle SDP packets. Since the stack is dissociated from the SDP packet management, applications are free to use their own SDP parsers or negotiate their media through another mechanism.

This package implements a simple one pass SDP parser, generator and container. It is up to the application to use the SdpParser for parsing SDP payloads.

1.1.9 SDP Capabilities Manager

The SDP capabilities manager is not depicted in Figure 1, as M5T SIP SAFE is completely dissociated from it. As with the SdpParser package, it is provided as a helper package for applications that must handle SDP negotiation as per RFC 3264. This package implements the basic mechanisms for performing the offer/answer mechanism required when negotiating SDP sessions between two peers.

This package depends on the SdpParser package for accessing the SDP information held by the SDP packets. It is up to the application to use the Capabilities manager once it has parsed an SDP packet, for managing the offer/answer mechanism.

2. Key M5T SIP SAFE Concepts

This section describes the key concepts required to use M5T SIP SAFE. It mainly focuses on the public interfaces of the SipParser and SipCore packages, while providing the necessary stack “internal” information to allow programmers to quickly start using the stack. Once these concepts are understood, they can be easily applied to understand how the user agent and proxy services work.

This documentation refers to the source code external to M5T SIP SAFE that is directly interacting with it as “the application”. This can be seen as the direct stack user.

2.1 Key Concept: Stack / Application Interfaces

This section describes the basic interfaces the application must use or implement to build a complete application with M5T SIP SAFE. This section however does not discuss the interaction between the application and the services, which is introduced in section 2.3: Key Concept: Pluggable Service Architecture.

Figure 2 introduces a few classes and interfaces the application must use or implement.

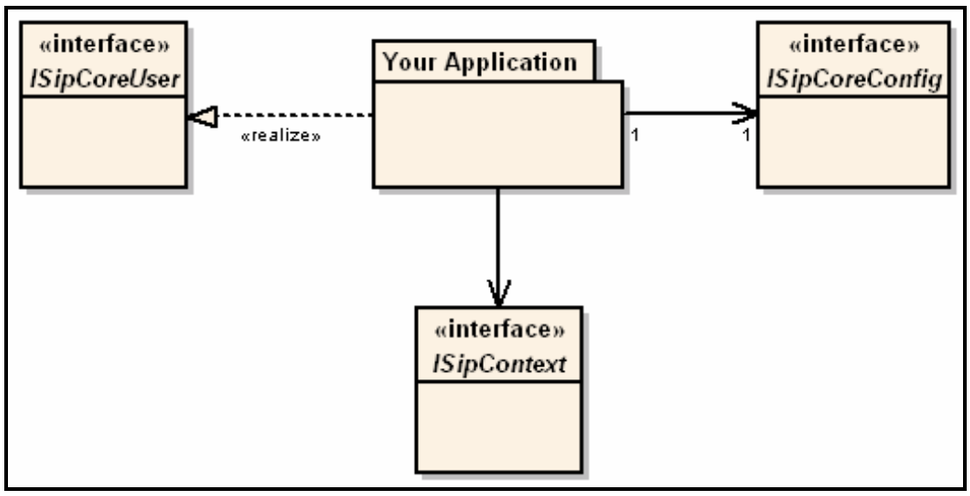


Figure 2: Application / Stack Interfaces

Table 3 - Classes / Interfaces

Class/Interface	Description
ISipCoreUser	The user of the SipCore package must implement this interface. This is the application. The stack uses this interface to report new incoming SIP packets that were not matched to an existing transaction. Upon detecting such a packet, the application can take different actions, which are introduced in section 2.2 Key Concept: Handling Unmatched Packets.
ISipCoreConfig	The application must use this ECOM interface to configure the stack. It is a central configuration interface for the stack. It offers, among other things, the stack startup and shutdown mechanisms, basic parser configuration and network related configuration. Refer to the API reference for more details on the various configuration options this interface offers.
ISipContext	The SIP Context is a central ECOM interface to the pluggable service architecture explained in section 2.3: Key Concept: Pluggable Service Architecture.

It is important to understand that even though Figure 2 shows the application as a monolithic block, it can be implemented as a set of different classes, each with their specific responsibilities.

2.2 Key Concept: Handling Unmatched Packets

After the SipTransport package receives a packet, it passes it to the SipTransaction package to detect retransmissions. If the packet does not match an existing transaction, it is provided to the application through the ISipCoreUser callback interface. The application must handle the packet differently depending on whether it is a user agent type of device or a proxy type of device. For this discussion, a user agent type of device is anything that will not forward a request as a SIP proxy (endpoint, B2B-UA, some application servers, etc). Furthermore, the application must handle the packet differently depending on whether it is a request or a response. The following sections describe the steps the different type of SIP entities must accomplish when handling a request or a response. The generic handling sections are for devices that simultaneously act as a proxy and a user agent.

2.2.1 User Agent Handling of Unmatched Requests

A user agent receiving a request that does not match an existing transaction should take the following steps:

Table 4- Steps to Handle Unmatched Requests

Step	Description
Step 1 – Authentication Check	Upon receiving a request, the application must check if it must challenge incoming requests. This is usually an application configuration option. If the application must challenge incoming requests, then it should use a stateless SIP Context with the stateless digest server authentication service to verify if the user has provided the proper credentials in the request. See the *ISipStatelessDigestServerAuthSvc API* for more information.
Step 2 – Dialog Matching Check	If the request is authenticated or if authentication was skipped, the application must try to match the incoming packet to an existing dialog by using the CSipDialogMatcherList object and calling the OnPacketReceived method with the packet it has just received. If the packet was handled by an existing dialog, the application can forget about the packet. If the packet was not handled by any dialog, the application may need to create a new SIP Context to handle the incoming request.
Step 3 – Create New SIP Context	If the packet was not handled by any existing dialog in the previous step, the application should create a new SIP Context to manage this incoming request. The application should look at the request type to decide exactly which services it must attach to the SIP Context. For instance, if an INVITE is received, the application should at least attach the session service, or if a SUBSCRIBE is received, the application should attach the notifier service. An application should only attach the services it requires; it is not necessary to attach all possible services on all SIP Contexts, as this would simply waste resources. If the application does not recognize this request or if it is not allowed to handle this type of request, it should create a SIP Context all the same with the necessary services to send back an error response. The generic service is usually used for this purpose. After the application has created the SIP Context and configured it with the necessary services, it must use the ISipContext::OnPacketReceived API to let the SIP Context and its services handle the incoming request. The application should ignore the reception of an ACK request that does not match any existing transaction or dialog. The reception of a CANCEL request that does not match any existing transaction or dialog should be answered with a 481 response.

2.2.2 User Agent Handling of Unmatched Responses

A user agent receiving a response that does not match an existing transaction should take the following steps:

Table 5 - Steps to Handle Unmatched Responses

Step	Description
Step 1 – Dialog Matching Check	The application must use the CSipDialogMatcherList object and call the OnPacketReceived method with the packet it has just received to see if it matches an existing dialog. If the packet matches a dialog, then the application can stop processing this response since a service attached to a context will have handled it.
Step 2 – Out of Context Responses	The application can ignore responses that do not match an existing transaction and dialog. Note that in case of response to INVITE requests, it is considered to be preferable to ignore the response and receive retransmissions than sending an ACK and misleading the other endpoint in thinking that a session is established.

2.2.3 Proxy Handling of Unmatched Requests

When a proxy receives a new request, it should take the following steps:

Table 6 - Steps for Proxy Handling of Unmatched Requests

Step	Description
Step 1 – Authentication Check	Upon receiving a request, the application must check if it must challenge incoming requests. This is usually an application configuration option. If the application must challenge incoming requests, it should use a stateless SIP Context with the stateless digest server authentication service to verify if the user has provided the proper credentials in the request. See the *ISipStatelessDigestServerAuthSvc API* for more information.
Step 2 – Request	If the request is authenticated or if authentication was skipped, the application can create a new SIP

Step	Description
Forwarding	Context and load the transaction stateful proxy service if the request is to be managed statefully. The application simply has to call OnPacketReceived on the ISipContext it has created. If the application must manage this request statelessly, it should then use a single SIP Context configured with a stateless proxy service for managing all requests and responses it must forward statelessly. The application would again call OnPacketReceived on the ISipContext that has the stateless service attached to it. The third alternative for proxy processing is the session stateful proxy service. This service keeps together all transactions related to one session into the same context instead of one transaction per context as with the transaction stateful proxy service. If your application needs to keep track of active sessions (dialogs created through INVITE requests), it must use this service attached to a new SIP Context. When doing so, a dialog matching step must be added prior to creating the new context just as with User Agent handling unmatched requests.

2.3 Key Concept: Pluggable Service Architecture

This section introduces the important concept of pluggable service architecture, which is central to the proper understanding of the M5T SIP SAFE capabilities.

2.3.1 The Services

A service is a modular piece of C++ code that implements a well-defined SIP or non-SIP feature. Services implement useful programming logic for the application, helping it accomplish different actions. They work together in a coherent way to offer a complete solution for interacting with other SIP entities. Care is taken when creating a service to balance functionality, flexibility and extensibility.

M5T SIP SAFE v4.1 supports only one type of service: the services that are attached to the stack through objects of the SipCore. These services are called **core services**.

Core services have a specific scope defined by the ISipContext instance upon which the services are attached. The ISipContext concept will be presented shortly; suffice to say that a core service can take decisions, report events or perform specific treatments only based on the SIP packets transiting through the ISipContext to which it is attached. A core service will live for as long as the ISipContext to which it is attached.

The services found under the SipCoreSvc, SipUserAgent and SipProxy packages are all core services. Please look at the available core services that are all described in the companion "*M5T SIP SAFE v4.1 - API Reference*" document. This document illustrates that the API of all the services are defined in modular interface classes "ISip[ServiceName]Svc". Accessing these interfaces will be explained later on.

2.3.1.1 ISipDestinationSeletionSvc and ISipViaManagementSvc

These two services MUST be added to all ISipContexts.

The ISipDestinationSelectionSvc is used to determine the next hop for the SIP packet. The next hop is the peer to which the packet will be sent. This can be either an IP address or a URI.

The ISipViaManagementSvc sets the top-most via header inside the SIP packet according to the local address used to send the packet. It has one method called SetViaManagementSvcMode, which is used to set either the stateful (default) or stateless mode. The stateless mode is used only by proxies and does not change the value of the branch parameter of the via header.

2.3.1.2 Core Service Ordering

The order in which the services are attached to the context is important. This determines the order in which the packet passes through the service chain and is modified by the chosen services. Some services need to be attached before others because they need to do some actions before the services that are next in the chain. The following table indicates the order in which the services should be attached.

Table 7 - Core Service Ordering

Services	Description
ISipUserAgentSvc	Basic user agent service that must always be attached on all services. It can be attached anywhere in the service chain.
Request Handler Services - Session, Transfer, Update, Publish, Subscriber, Notifier, Registration, Generic, etc.	These services will mostly always report their event last by design, as they are the handlers of the requests.
ISipUaAssertedIdentitySvc	UAC and UAS. Should report its event first if there is a problem or not, telling whether or not to trust the P-Asserted-Identity header. It will possibly add a header for outgoing requests.

Services	Description
ISipReplacesSvc	UAS service only, all server events are reported, so it can practically be anywhere.
ISipGlareSvc	UAC service, could practically be anywhere but best attached after the ISipUserAgentSvc as the other services will be silenced.
ISipSessionTimerSvc	UAC - To give priority for 422 handling over redirection.
ISipDigestClientAuthSvc	UAC - To give priority to 401/407 over redirection.
ISipRedirectionSvc	UAC, redirection as a last option when trying to contact someone.
ISipServerMonitorSvc	UAC - This service MUST be inserted after the ISipDigestClientAuthSvc and after the ISipRedirectionSvc to prevent it from interfering in authentication requests and redirection retries.
ISipTransactionCompletionSvc	This must be among the last service to add its event in the event queue for core services. This way we are sure all other events were reported.
ISipPrivacySvc	Will report event on server transaction, all events are reported, so it is ok. Will possibly modify outgoing packets to anonymize it. It must be among the last core service attached so it will modify a packet that is mostly complete.
ISipStatelessDigestServerAuthSvc	Used to authenticate requests.

2.3.1.3 Connection Service Ordering

The order in which the connection services are attached to a context are very important as each service will have a different behaviour, depending on how previous services have modified the packet. Some services need to be attached before others because they need to do some actions before the services that are next in the chain. The following table indicates the order in which the services should be attached.

Table 8 - Connection Service Ordering

Services	Description
ISipDestinationSelectionSvc	**Mandatory**. Attach it prior to attaching the ISipServerLocationSvc, or ISipOutboundConnectionSvc or ISipPersistentConnectionSvc. Sets the destination to which the packet is to be sent.
ISipEnumSvc	Changes a tel-URI into sip-URI.
ISipPersistentConnectionSvc	Will fix-up the destination according to what we were trying to contact. From here, the destination is known. It must not be attached with the ISipOutboundConnectionSvc. It can be used without the ISipServerLocationSvc, in which case no DNS request is performed and the persistent connection corresponding to the peer is used.
ISipOutboundConnectionSvc	Can be after the ISipServerLocationSvc or it can be used on a context for which the ISipServerLocationSvc is not attached. It MUST NOT be attached on a context with ISipPersistentConnectionSvc.
ISipServerLocationSvc	Provides DNS resolution of the target.
ISipViaManagementSvc	**Mandatory**: Properly sets the value of the Via sent-by according to the processing done in the services. The Via is set according to the network interface used to reach the peer destination.
ISipSymmetricUdpSvc	Makes sure to use a listening UDP port. It must be set after ISipViaManagementSvc as it sets the local address. This local address is overridden by the ISipSymmetricUdpSvc.
ISipOutputControllingSvc	Final packet that can be modified by the application.
ISipConnectionBlacklistSvc	May prevent sending to the chosen destination. It can be attached before the output controlling service if the application does not want the backlist to apply to the destination it sets in the packet.

2.3.2 The SIP Context

The SIP Context is defined as the ISipContext ECOM interface in the SipCore package. This important object is mainly a dumb container of core services. The application using the stack creates a SIP Context and then attaches the core services it wants or requires on it. The SIP Context supports user agent and proxy type of services, found in the SipUserAgent and SipProxy packages.

The application creates SIP Contexts for a specific purpose. It could be to register a local UA with a registrar server, receive a call, forward a request as a transaction stateful proxy, or any other specific SIP feature. Since an application may want to simultaneously do more than one such task, it is possible for the application to create as many SIP Contexts as necessary, limited only by the system's memory.

The application will attach different services on a SIP Context depending on its purpose. The first step is to determine whether this SIP Context is to act as a proxy or user agent. If it is to act as a proxy, then the application must load one of the proxy services available in the SipProxy package. If the SIP Context is to act as a user agent, then the application must load the ISipUserAgentSvc, available in the SipUserAgent package. The application then loads additional core services on the SIP Context. Which services to load depend on what the application is trying to achieve, what it wants to support, and what kind of help it wants from M5T SIP SAFE. As such, a SIP Context can be seen as a small SIP proxy or user agent instance, depending on the services attached to it.

All core services attached to a SIP Context work together to help the application with various SIP features. When the application uses a service to send a request, all other services attached to the SIP Context are immediately involved in the creation of the request, whether to specify additional information in the packet or just keep proper state for managing the response to the request. When a response associated with a SIP Context is received, again all services attached to the SIP Context are involved in the reception and may take action, depending on the response type and its content. The same is true when receiving a request and sending a response.

Example – SIP Context to register a user

To register a user (send a REGISTER request to a SIP registrar server), the application could load the following services:

- User agent service: to define the SIP Context as a user agent.
- Registration service: helps manage registrations and lets the application know when a registration must be refreshed.
- Client digest authentication service: helps the application manage 401 or 407 challenges.
- Server location service: determines the next hop where to send the request according to the rules found in RFC 3263.

The application could have chosen to load different services to send a REGISTER request to a registrar. For instance, the application could also send a REGISTER request by loading the following services:

- User agent service: to define the SIP Context as a user agent.
- Generic service: allows to send and receive any type of request.
- Server location service: determines the next hop where to send the request according to the rules found in RFC 3263.

Note however that if the application was to load this set of services, it would have to implement the service logic for the management of the registration expiration and the service logic required for providing credentials if the request is challenged.

Example – SIP Context for stateless proxy

To forward a request as a stateless proxy, the application could load these two services:

- Stateless proxy service: defines this context as a proxy service and helps the application determine whether or not it should perform a lookup in its location database.
- Server location service: determines the next hop where to send the request according to the rules found in RFC 3263.

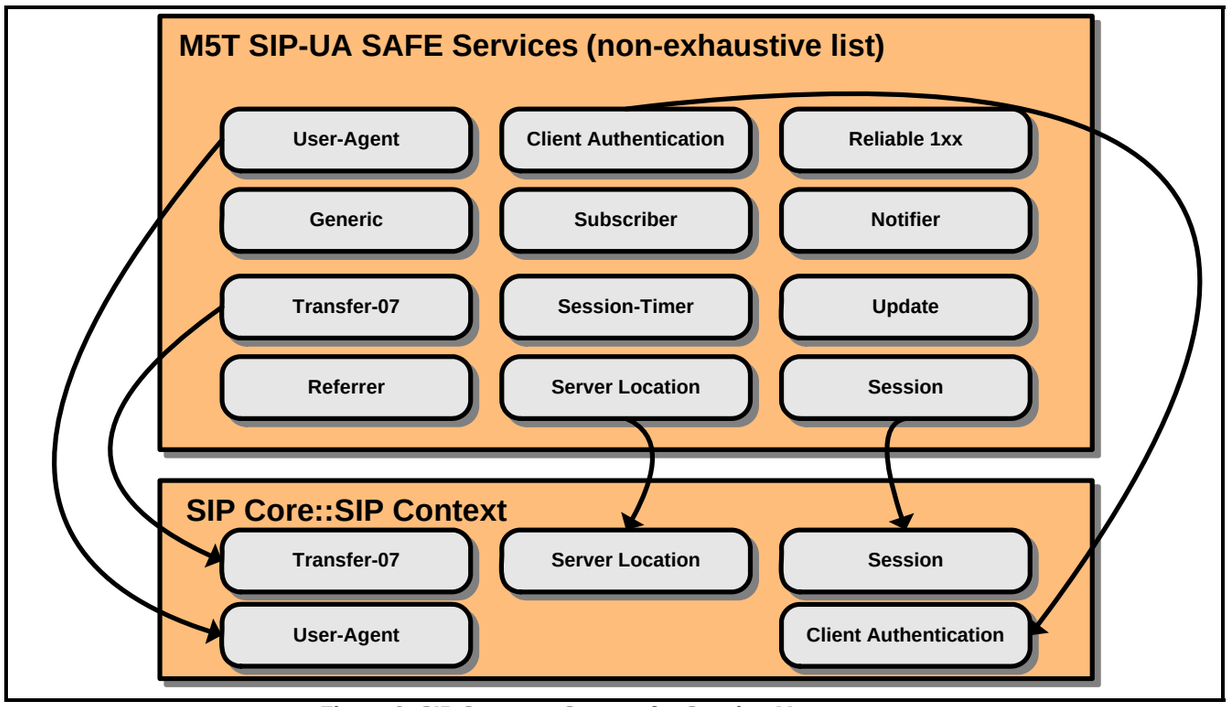


Figure 3: SIP Context - Context for Session Management

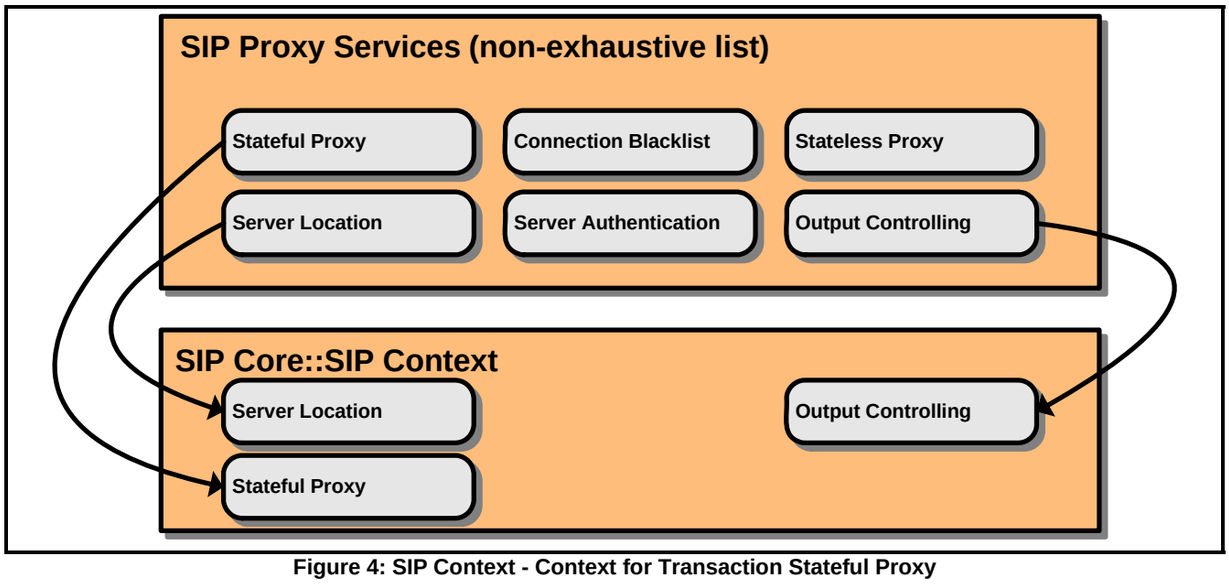


Figure 4: SIP Context - Context for Transaction Stateful Proxy

2.3.3 The UA SIP Context

It is important to understand the effect of attaching the user agent service on a SIP Context. Let's define a UA SIP Context as "a SIP Context with an attached user agent service". The user agent service is responsible for the following information:

- Local address: the From header of outgoing requests, including tag;
- Local Contact;
- Peer address: the To header of outgoing requests, including tag;
- Peer request-uri;
- Call-Id header;
- Local CSeq header;
- Peer CSeq header;
- Route (pre-loaded and established) and Record-Route headers.

When sending a request, the user agent service uses the aforementioned information to properly configure the SIP packet to send, while the other core services attached to the SIP Context may add additional information to the packet. The user agent service holds the above information and other core services update it when necessary. This information is used for dialog matching of SIP Contexts that have established dialogs.

It is important to note that the information held by the user agent service dictates that a UA SIP Context must only be used for one dialog, or for management of requests that do not establish dialogs.

A few simple rules:

- When sending requests with a UA SIP Context, the From, To, Call-ID, and Top Via headers are always the same. The CSeq header is properly incremented by one, unless the application changes this information, which it should not do after sending the initial request.
- A UA SIP Context can manage a SIP dialog. It keeps the From/To tags when necessary and keeps proper count of the dialog usage. It also keeps the pre-loaded route until a dialog is established.
- A UA SIP Context can manage a series of requests that do not establish a dialog, like the initial REGISTER request and subsequent registration refreshes.
- Requests that must be sent on a specific dialog must be sent by using services attached to the proper UA SIP Context that manages that dialog.

Once a UA SIP Context's purpose is fulfilled, it should be released, and will then automatically be removed from the system memory.

Sample uses of UA SIP Context

- Establishing a session with INVITE: All subsequent requests for this session must be made on the same UA SIP Context.
- Registering to a registrar: The initial and all subsequent refreshes are done on the same UA SIP Context for a specific address of record, specified in the To header.
- Sending MESSAGE requests to a single destination.
- Subscribing to a voice mail server for notifications: Notifications are received on the UA SIP Context used to subscribe. Subscription refreshes must also use the same UA SIP Context.
- Subscribing to a presence server.
- Acting as a presence server and receiving subscriptions.

If the application was to take the UA SIP Context it used for registering and try to reuse it for placing a call, then several problems would occur. Unless the application was changing the user agent service configuration, the INVITE request would be sent to the SIP registrar, and the To header would most likely correspond to the local user's address of record, instead of corresponding to the peer user's one. Moreover, if the application was to change the user agent service configuration, then it

would break future registration refreshes performed on this SIP Context. Each context should have a predefined use, defined by the services that are attached to it.

2.3.4 The Proxy SIP Context

An application may decide to create a Proxy Context with one of the following Proxy services:

- Session Stateful Proxy Service
- Transaction Stateful Proxy Service
- Stateless Proxy Service

It is important to note that the information held by the Proxy Context dictates the Proxy behaviour. Before using a Proxy Context, the application must add one of the above Proxy Services and provide a CSipProxyConfig to the Proxy Service.

A few simple rules about the Transaction Stateful Proxy Service:

- Used to manage one single transaction. This means the context live time is the time between the request and the response.
- The ACK or CANCEL request must not be managed by this service. They must be handled through the Stateless Proxy Service. Unmatched responses must also be managed through the Stateless Proxy Service.

Sample uses of Transaction Stateful Proxy Context

- Acting as a Proxy Server to forward requests and responses to SIP entities.
- Acting as a Proxy Server that implements serial and parallel forking.

Most generally, a new Transaction Stateful Proxy Context is created upon receiving a new request. That request is forwarded upstream, when its response is forwarded downstream and then the SIP context is released.

2.3.5 SIP Context Lifecycle

The following subsections describe in more details the lifecycle of a SIP Context object. Please refer to Figure 19 on page 37 for a graphical representation of this lifecycle.

The SIP Context and the core services are all ECOM objects. The ECOM mechanism is described in the "*M5T Framework SAFE v2.1 - Programmer's Guide*", in the section entitled "*ECom*". Please make sure to understand the ECOM mechanism before further reading this section.

2.3.5.1 Creating a SIP Context

To create a SIP Context, the standard ECOM factory mechanism must be used. All CLSID_ and IID_ parameters are documented in the "*M5T SIP SAFE v4.1 - API Reference*" document.

```
ISipContext* pSipContext = NULL;
result = CreateEComInstance(CLSID CSipContext,
                            NULL,
                            &pSipContext);
```

2.3.5.2 Attaching Core Services to a SIP Context

To attach any core service to a SIP Context, the application calls the AttachService API of the ISipContext instance upon which the service is to be attached.

```
result = pSipContext->AttachService(CLSID_CSipSessionSvc);
```

All CLSID_ parameters supported by M5T SIP SAFE are documented in the "*M5T SIP SAFE v4.1 - API Reference*" document.

Once a service has been attached to a SIP Context, it can be accessed through the mechanism described in the next section.

Some services require to be attached in a specific order, i.e. when the outbound connection service is used with the server location service, the last one must be attached before the outbound connection service.

2.3.5.3 Accessing Core Services Attached to a SIP Context

The SIP Context and all services attached to it can be seen as one big object. From any interface supported by this object, you can access any other interface it supports by using the ECOM QueryIf API.

```
// Get Context from Session service.
ISipSessionSvc* pSessionSvc = NULL;
mxt result result = pSipContext->QueryIf(&pSessionSvc);
if (MX RIS S(result))
{
    pSessionSvc->DoSomething();
    pSessionSvc->ReleaseIfRef();
    pSessionSvc = NULL;
}
```

The above examples show the QueryIf call being made on the SIP Context, but it could have been done on any other service interface that the SIP Context currently has attached.

```
// Get Session service from the SessionTimer service.
ISipSessionSvc* pSessionSvc = NULL;
```

```
mxt_result result = pSessionTimerSvc->QueryIf(&pSessionSvc);
if (MX_RIS_S(result))
{
    pSessionSvc->DoSomething();
    pSessionSvc->ReleaseIfRef();
    pSessionSvc = NULL;
}
```

2.3.5.4 Releasing a SIP Context

Once no longer needed by the application, the final reference held by the application on the SIP context must only be released after the call to Clear().

```
// Prepare the SIP Context for deletion.
pSipContext->Clear();

// Release ECom reference.
pSipContext->ReleaseIfRef();
pSipContext = NULL;
```

2.4 Key Concept: Event Management

The previous section introduced the core services, each having specific responsibilities regarding specific features. As described earlier, the goal of these core services is to help the application by providing as much service logic as possible, without giving up extensibility and flexibility. Obviously, the core services will have to communicate in some way with the application to report events.

Each service that needs to report events to the application has a corresponding manager interface, a C++ abstract base class that the application must implement. The list of events that a specific service can report is detailed in its manager interface.

Example

- The API of the session service is defined in the ISipSessionSvc interface. This service reports events to the application through the ISipSessionMgr interface, which must be implemented by the application.
- The API of the session timer service is defined in the ISipSessionTimerSvc interface. This service reports events to the application through the ISipSessionTimerMgr interface, which must be implemented by the application.
- The API of the user agent service is defined in the ISipUserAgentSvc interface. This service never reports any event, so it defines no manager interfaces.

Figure 5 represents an application using three services: the registration, user agent and session services. The registration and session services report events to the application through their manager interfaces. To receive such events, the application inherits from the manager interfaces.

What is important to note in this diagram is the one-to-one relationship between the service and its manager. When the application attaches a service to a SIP Context, it must immediately configure the service with its manager. All services that report events have a SetManager() API for this purpose.

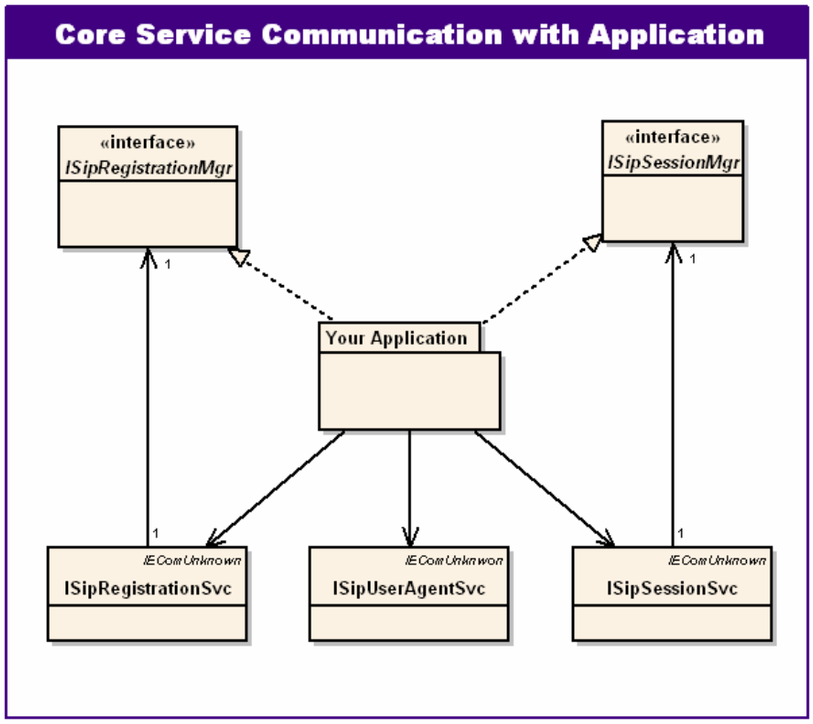


Figure 5: Event Management Class Diagram

This one-to-one relationship allows the application to have events reported to different instances of the application class implementing the manager interface.

There are three different types of events the core services can report to the application through the manager interfaces.

Table 9 - Event Types to Report

Event Types	Description
Client Events	Reported to the application when receiving responses to a request the application sent with a service. This type of event must receive a special treatment from the application.
Server Events	Reported to the application when the stack detects the reception of a new request. This type of event must receive a special treatment from the application.
Other Events	Unless otherwise specified in the event documentation, do not require any special treatment.

The following sections provide more information about these events and how the application must manage them.

2.4.1 Client Events

Client events are reported when receiving responses to a previously sent request. For a request sent by using a service attached to a specific SIP Context, all client events corresponding to this request will come from services that were attached to this same SIP Context.

Example

A UA application can support two simultaneous calls. This application currently has two active calls, represented by two SIP Contexts “A” and “B”, and each SIP Context has the same core services attached.

If the application uses the session service on SIP Context “A” to send a re-INVITE request, then all client events for this request will only be generated by the services found on this SIP Context. The services on SIP Context “B” never see the re-INVITE request going out and its responses coming in.

M5T SIP SAFE checks incoming responses to see if they match an existing transaction. If the response matches an existing transaction, the packet is immediately given to the SIP Context managing this transaction. All services attached to the corresponding SIP Context get to see the received response. When examining the response, a service may want to report an event to the application, in which case it adds the event to an event queue. When all services have seen the response and possibly added an event to the event queue, the first event in the queue is reported to the application.

All client events reported to the application have the following signature and parameters:

```
virtual void EvSomeEvent(IN ISipSomeKindOfSvc* pSvc,
                         IN ISipClientEventControl* pClientEventCtrl,
                         IN const CSipPacket& rResponse) = 0;
```

The pSvc parameter is a pointer to the service that is actually reporting the event. It can be used for any purpose, but it is really useful to find the SIP Context to which the service was attached. This is done by performing a query interface on pSvc.

The rResponse parameter is the actual response received. The application can use this if it needs to access further information from the packet.

The pClientEventCtrl parameter is central to the management of client events. An application communicates to the stack which type of event processing was done for the event through this interface.

When handling a client event, the application can use the following APIs on the pClientEventCtrl parameter:

- Do some processing and call ReIssueRequest.
- Possibly do some processing and use the CallNextClientEvent API.
- Do the event processing and call ClearClientEvents.

CallNextClientEvent, ReIssueRequest, and ClearClientEvents are all APIs that the ISipClientEventControl interface offers. Depending on the event processing the application has done, one of these methods must be called.

Table 10 - Method vs. Event Processing

Method	Description
ReIssueRequest	This method is the key to a very important feature of M5T SIP SAFE. First, it is important to understand the request caching done by the stack. All requests sent through user agent services are cached by the stack. This includes the request type, application-specified headers, and payload originally sent with the request. The application can use ReIssueRequest to re-send a cached request *after a failure response was received for the original request.* This is a very useful feature of M5T SIP SAFE as it frees the application from keeping the state for knowing which request was sent with which application-specified headers and payload. This allows the application to manage in a generic way a lot of failure events reported by the stack. For instance, an application receiving an “EvCredentialsRequired” from the digest client authentication service could get the credentials from the user, configure the credentials through the authentication service, and then call ReIssueRequest, without having to know whether the challenged request was an INVITE, SUBSCRIBE, MESSAGE, etc. The authentication service can then add authentication information to the new outgoing request. Calling ReIssueRequest clears the

Method	Description
	previously cached request and re-sends the request, which is then cached again. The event queue is also cleared (see CallNextClientEvent).
CallNextClientEvent	When a response is received on a SIP Context, all services have the chance to see the incoming packet and choose whether or not they would like to report an event. Each service that wants to report an event places its event into an event queue and the first event is then reported to the application. The CallNextClientEvent method allows the application to instruct the stack to report the next event in the event queue. Calling this method is useful, for instance, when an application receiving an "EvCredentialsRequired" from the digest client authentication service cannot get user's credentials, in which case another event may bring some other possibilities for making the request progress further. This cannot be called by the application when handling the event from the service that was used to send the request, as this is always the last event and there are no more events to call. It is important to note that calling this does not clear the previously cached request and only removes the next event from the event queue.
ClearClientEvents	This method allows the application to communicate to the stack that it is done handling events for the current response it is managing. When handling a provisional response (1xx), the application must absolutely call ClearClientEvents when it is done managing this response. When handling a final positive response (2xx), the application must also absolutely call this method, which clears the request cache. When handling a final negative response (>=300), the application must eventually end by calling either ClearClientEvents or ReIssueRequest, which clears the request cache and event queue. Note that zero or more calls to CallNextClientEvent are possible before calling one of these methods.

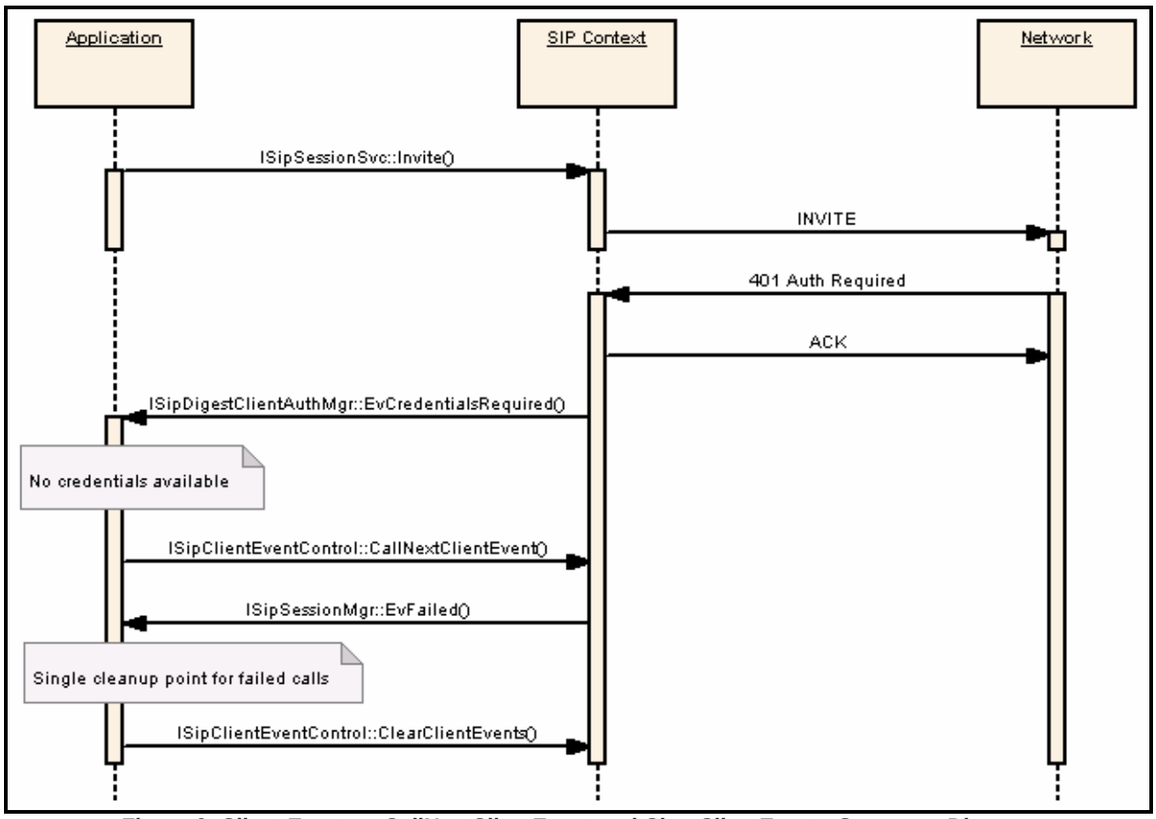


Figure 6: Client Events – CallNextClientEvent and ClearClientEvents Sequence Diagram

Event Ordering

The order in which events are reported to the application is determined by the order in which the services are attached to the SIP Context. The first service attached to a SIP Context gets to report its event first; the second service then gets to report its event, and so on.

However, there is an exception to this rule. The service that has been used for sending the request to which a response is received will always get to report its event *last*. This is simply because other services can be able to report some useful event to the application in case of a failure response (>= 300), while the service used for sending the request usually only reports that the request has failed.

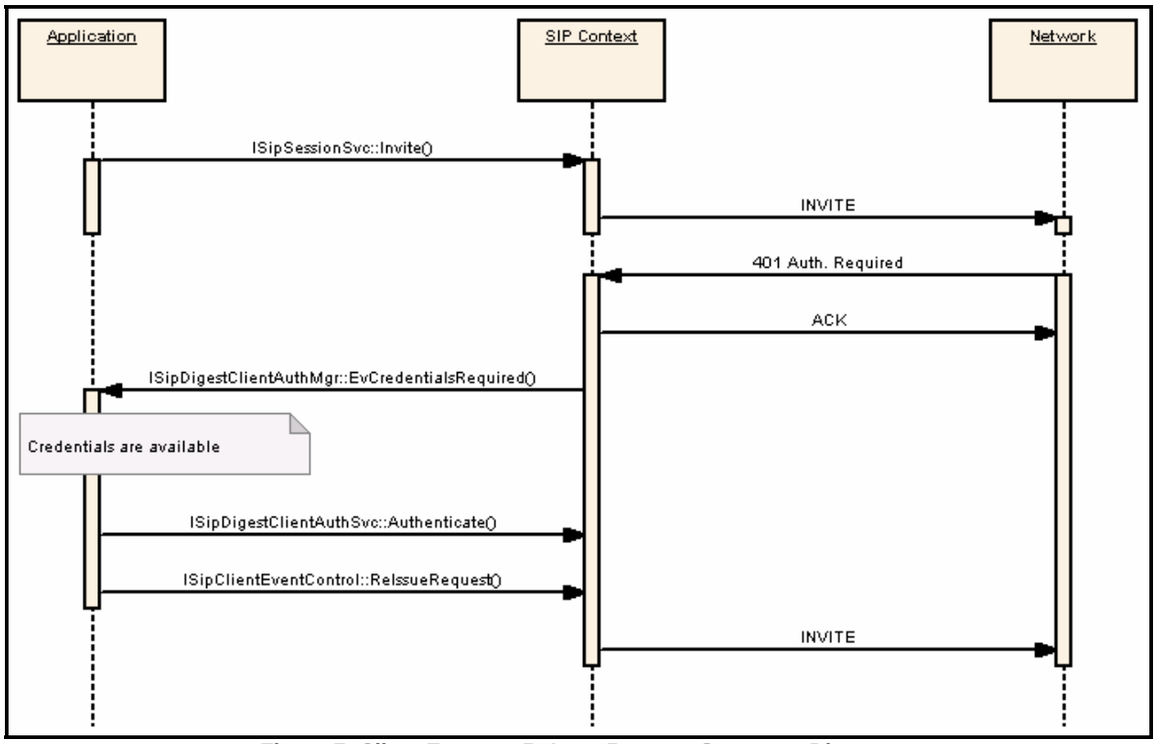


Figure 7: Client Events – ReIssueRequest Sequence Diagram

Client Event Handling

The application can manage reported client events either synchronously or asynchronously. Synchronously means the application can immediately manage the event and call any stack function necessary for this management. This possibly includes re-sending the request with ReIssueRequest or calling the same method that was called originally for sending the request.

The application is managing the event asynchronously if it somehow has to wait before being able to manage the event, like asking for user input, accessing a database, and so on. When managing an event asynchronously, the application must make sure it increments the reference count of ECOM objects it references, until it has managed the event, at which time it should release the reference it owns on such objects.

ISipClientEventControl Management

ISipClientEventControl is an ECOM object and its use must follow the ECOM rules. When the stack reports a client event and provides an ISipClientEventControl as a parameter, this ECOM object is guaranteed to remain valid during the entire function call. As soon as the event handler returns, this parameter must be considered as invalid, unless the application has kept an ECOM reference on it. An application managing an event asynchronously MUST keep an ECOM reference on the ISipClientEventControl parameter until either ClearClientEvents or ReIssueRequest is called on this object, after which the application should release its reference on it.

The following code shows two examples of ISipClientEventControl management.

```
// Synchronous management
void CGenericMgr::EvProgress(
                          IN ISipGenericSvc* pSvc,
                          IN ISipClientEventControl* pClientEventCtrl,
                          IN const CSipPacket& rResponse)
{
    // Do processing required by the application upon reception of a provisional response
    // to a request sent through the generic service.

    // Since this is the event reported by the service that was used to send the request
    // it is guaranteed to be the last event and hence there is nothing else to do than
    // clearing the client event control.
    pClientEventCtrl->ClearClientEvents();
}
```

```
// Asynchronous processing
void CSipDigestClientAuthMgr::EvCredentialsRequired(
                           IN ISipDigestClientAuthSvc* pSvc,
                           IN ISipClientEventControl* pClientEventCtrl,
                           IN const CSipPacket& rResponse)
{
    // Obtain the challenge from the response and present them to the user so it can
    // enter its username and password.

    // Store the service, the client event control and the response until the credentials
    // are received from the user. Do not forget to count a reference for each of these
    // elements.
    m pSvc = pSvc;
    m pSvc->AddIfRef();
    m pClientEventCtrl = pClientEventCtrl;
    m pClientEventCtrl->AddIfRef();
    m pResponse = &rResponse;
    m pResponse->AddRef();
}

void CSipDigestClientAuthMgr::CancelAuthentication()
{
    // When the user does not have credentials to give he will hit the "cancel" button
    // and this method will be called.

    // Call the next client event and release our references
    m pClientEventCtrl->CallNextClientEvent();
    m pSvc->ReleaseIfRef();
    m pSvc = NULL;
    m pClientEventCtrl->ReleaseIfRef();
    m pClientEventCtrl = NULL;
    m pResponse ->Release();
    m pResponse = NULL;
}

void CSipDigestClientAuthMgr::SetCredentials( IN const char* szUsername,
                                              IN const char* szPassword )
{
    // When the user gives proper username and password this method is called

    // Create the appropriate credentials from the username and password in parameter
    // and give them to the previously stored service.

    // Reissue the request
    m pClientEventCtrl->ReIssueRequest(opqTransactionId, OUT m pClientTransaction);

    // Clear references on the service, client event control and packet as
    // they are not needed anymore.
    m pSvc->ReleaseIfRef();
    m pSvc = NULL;
    m pClientEventCtrl->ReleaseIfRef();
    m pClientEventCtrl = NULL;
    m pResponse ->Release();
    m pResponse = NULL;
}
```

2.4.2 Server Events

Server events are reported upon receiving a new SIP request on a SIP Context. As with the client events, all services attached on the SIP Context see the incoming request and, when appropriate, place an event into the event queue. However, there is a fundamental difference between the client and server events; all queued server events are usually reported to the application. The exception to this is when a service detects an error within the received request and decides to send back a failure response. In this case only, all the remaining events are NOT reported to the application.

Request Manager

When a SIP Context is managing a received SIP request, it tries to find a service that actually knows how to manage this request. Only one service among the ones attached to the SIP Context must be able to manage this incoming request. This service is called the request manager. The other services attached to the SIP Context are considered as helper services for this transaction. When reporting server events, the helper services get to report their events first, followed by the event from the request manager, if no final response was sent by one of the helper service. For example, the request manager for the INVITE method is by default the session service.

Managing Server Events from Helper Services

As with all events reported by services in the stack, the events from the helper service specify the service that is reporting the event, the packet associated with the event, and an mxt_opaque parameter for the application's own use.

```
// Sample event from helper service.
//-----------------------------------
virtual void EvSomeHelperEvent(IN ISipSomeService* pSvc,
                               /* possibly some more parameters */
                               IN const CSipPacket& rRequest,
                               INOUT mxt_opaque& rApplicationData) = 0;
```

This mxt_opaque parameter is very important for the application. This is an opaque parameter for M5T SIP SAFE, so it is never used or interpreted by the stack. For each new request managed by a SIP Context, the application can associate an opaque parameter to be able to do some application level mapping of this transaction. The first server event reported to the application for a new request will always have this opaque parameter set to zero. Since it is an INOUT parameter, the application can change its value to anything else. This opaque parameter is usually set to a pointer to some class or structure holding the application state for the transaction and possibly some more state for the server event management. This opaque value set by the application is provided to all following server events reported to the application for the request being handled.

Once the application has finished handling the first helper event, the next helper event (if any) is reported, with the same mxt_opaque parameter that was set by the application in the initial event handler.

When handling a server event from a helper service, the application should keep the event information without further handling. Once handling the request manager server event, the application should look at which helper service events it has received and take the proper actions, based on the set of events it has received. It is important for the application to keep the state of the events from the helper services and not try to actually handle the event itself, as it is possible that a service later decides to send back a negative final response to the request. See the Server Events example section for examples of how an application should and should not handle events from helper services.

Managing Server Events from the Request Manager

Server events from the service that is responsible for this transaction have two different possible signatures. The following sample event is associated with a completely valid SIP request that is accepted by all services attached on the SIP Context. The important parameter in this event is the ISipServerEventControl parameter, which gives access to the mxt_opaque application data for this transaction. This is the same mxt_opaque parameter that is provided in the events from the helper services; thus if the application stored its state in this parameter to the previous server events, it can find back its state through the ISipServerEventControl::GetOpaque API.

```
// Sample event from request manager.
//-----------------------------------
virtual void EvSomeManagerEvent(IN ISipSomeManagerSvc* pSvc,
                                /* possibly some more parameters */
                                IN ISipServerEventControl* pServerEventCtrl,
                                IN const CSipPacket& rRequest) = 0;
```

The ISipServerEventControl parameter also allows the application to send back responses to this incoming request. Depending on the type of request received, the application can send zero or more provisional responses (1xx) and exactly one final response (200 or greater).

The signature of request manager events for invalid requests is described next.

Managing Service Reported Errors

There are some cases where a service (whether helper or request manager) will detect that an incoming request is invalid. In this case, the service automatically sends a response to the request and reports an event to the application to this effect.

The next code sample provides an example of an event when a service has detected that a request is invalid.

```
// Sample event when a service detects a bad request
//--------------------------------------------------
virtual void EvInvalidSomething(IN ISipSomeSvc* pSvc,
                                IN mxt opaque applicationData,
                                IN const CSipPacket& rRequest,
                                IN mxt_result resReason) = 0;
```

Again, an important parameter to such events is the mxt_opaque parameter, which allows the application to free the state that it previously had stored in the opaque parameter of previous events.

Event Ordering

The server events are reported following the order in which the services were attached to the SIP Context. The first service attached to a SIP Context gets to report its event first, if it has any. The request manager service is the last to report its event.

Server Event Handling

All events from helper services must be managed (mostly saving the state) synchronously; the application **MUST NOT** place these events into a message queue for later processing. Only the last event from the request manager can be processed synchronously or asynchronously, as the application has received all events for this packet and should have accumulated all necessary states to properly process this request.

The application is managing the event asynchronously if it somehow has to wait before being able to manage the event, like asking for user input, accessing a database and so on. When managing an event asynchronously, the application must make sure it increments the reference count of ECOM objects it references, until it has managed the event, at which time it should release the reference it owns on such objects.

ISipServerEventControl Management

The ISipServerEventControl object is an ECOM object, and as such must be properly managed according to the ECOM specifications. The ISipServerEventControl pointer provided in some server events is guaranteed to remain valid until the event call returns. If the application manages the last server event asynchronously, it must make sure it keeps a reference on the ISipServerEventControl until a final response is sent, after which the application can release this object.

Examples

Figure 8 shows a simplified example where a SIP Context is configured with the session and replaces service. Upon receiving an INVITE with a Replaces header, the stack reports an EvReplaces event from the replaces service. This is an event from a

helper service. When handling this event, the application must keep the proper state, remembering that another SIP Context is to be replaced with this new one. Once the application is done handling this event, the session service reports the EvInvited event, which is an event from the request manager. When handling this event, the application should get the previously stored state and detect that it must replace an existing SIP Context with the one on which this INVITE was just received.

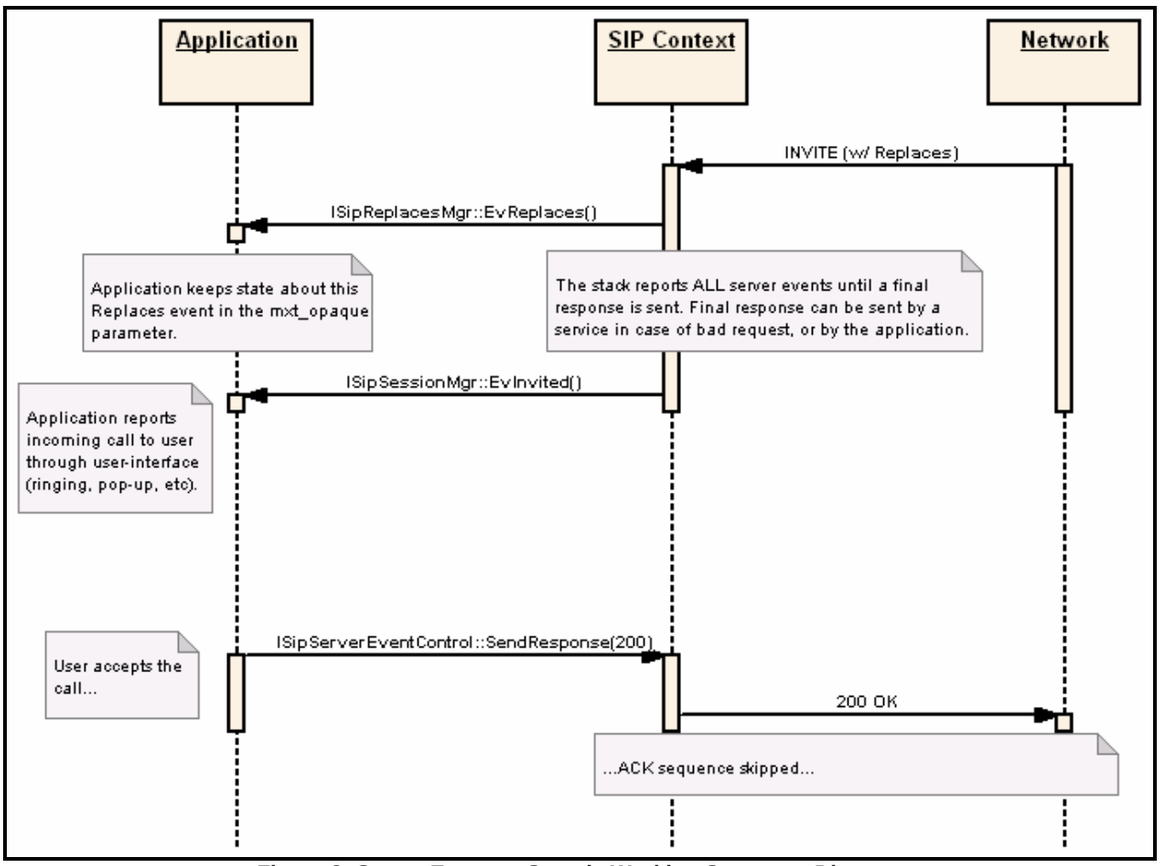


Figure 8: Server Events – Sample Working Sequence Diagram

Figure 9 shows again a SIP Context configured with session and replaces services, but this time the received INVITE is invalid. Notice that the EvReplaces event is still reported to the application, until the session service closely examines the request, where it detects the error.

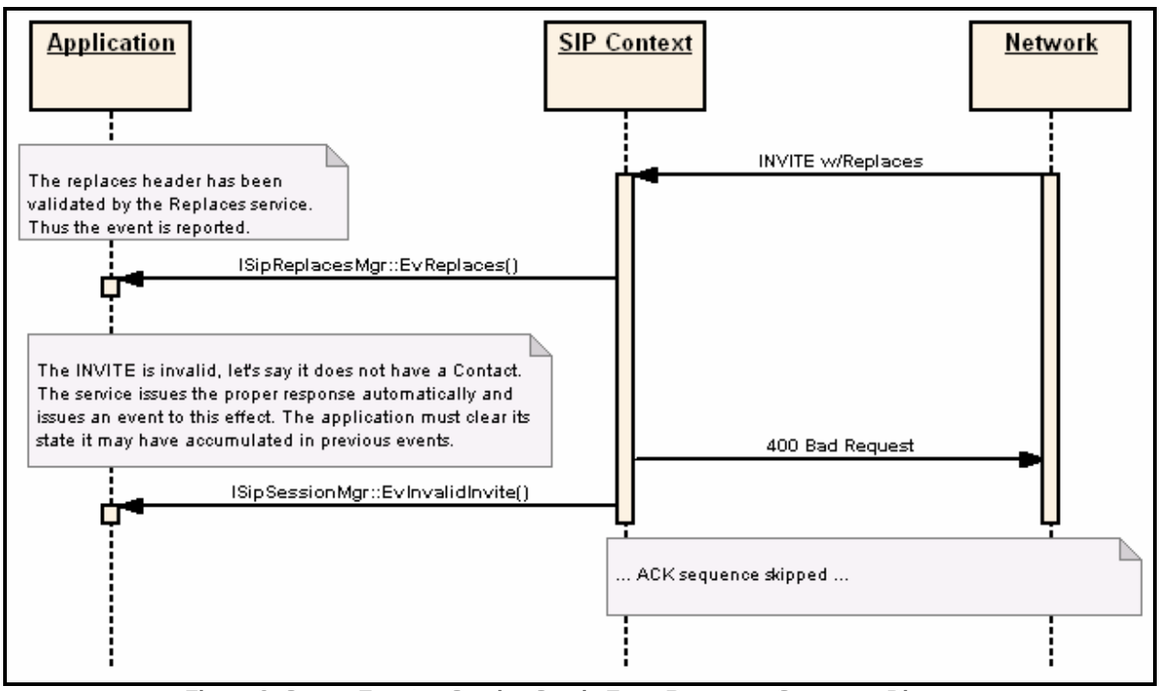


Figure 9: Server Events - Service Sends Error Response Sequence Diagram

Figure 10 shows again a SIP Context configured with session and replaces services. This sequence diagram highlights the potential problems the application can have if it tries to handle an event from a helper service. When handling such an event, the application only needs to update its state for this transaction. The actual event management must only be done when receiving the proper event from the request manager service.

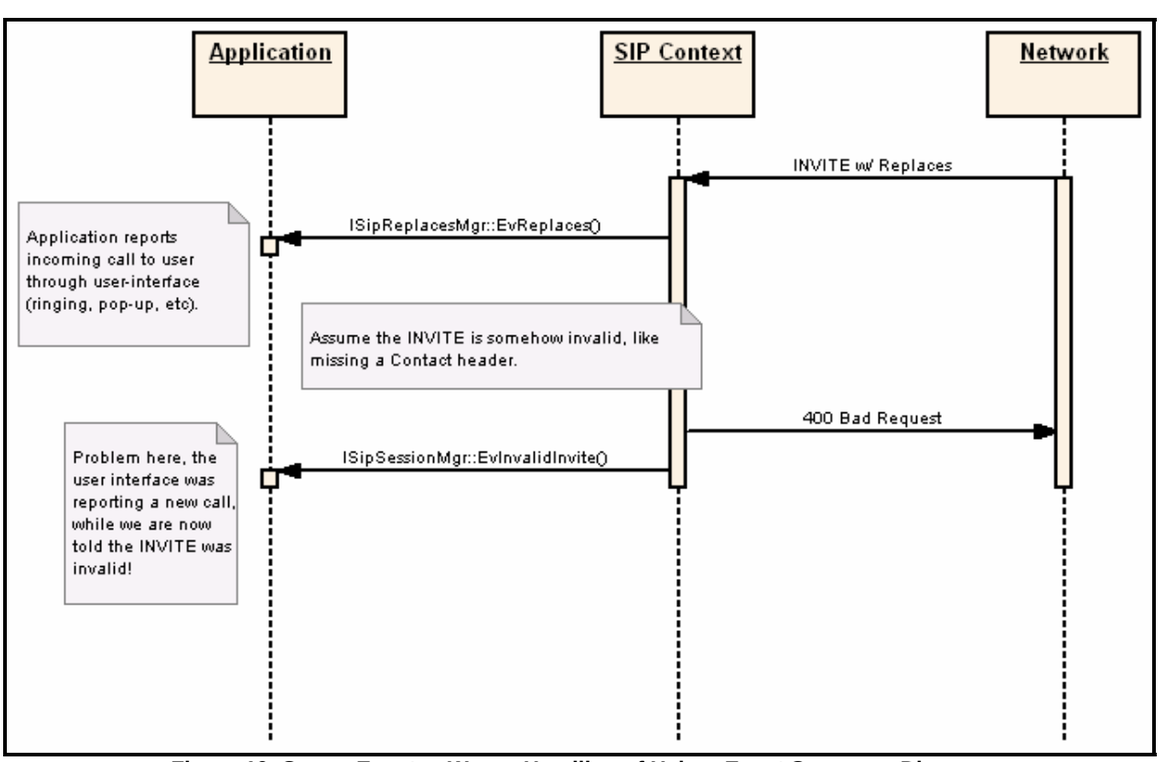


Figure 10: Server Events - Wrong Handling of Helper Event Sequence Diagram

2.4.3 Other Events

M5T SIP SAFE can also report events that are not based on the reception of a request or a response. These events are usually reported to the application based on timer expiration, and thus no ISipClientEventControl or ISipServerEventControl objects are provided as a parameter.

There are no special handling rules for the application when such an event is reported. Follow the documentation provided in the manager API.

2.5 Key Concept: Threading

M5T SIP SAFE offers a very flexible configuration mechanism for different types of applications, whether or not multi-thread operation is supported. Some parts of the stack require a thread to run, and the application is responsible for configuring these parts with the proper thread to use. This mechanism is based on the M5T Framework servicing threads.

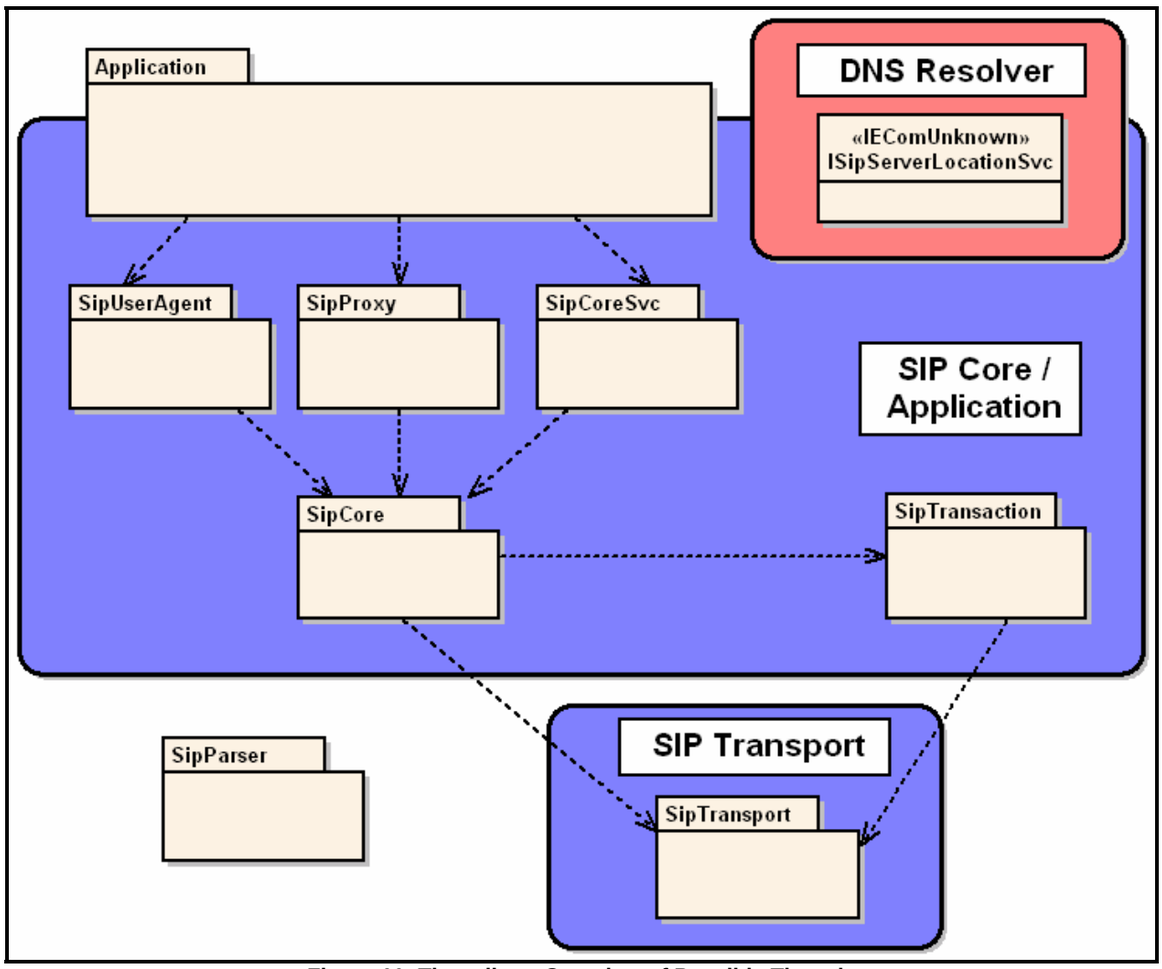


Figure 11: Threading - Overview of Possible Threads

Figure 11 represents the M5T SIP SAFE packages and the modules that require a servicing thread to properly run the stack.

Table 11 - Modules that Require a Servicing Thread

Module	Description
SIP Transport	This is where the stack is performing the stack parsing and generation of packets to be sent on the network. This is also where the connection management is performed, along with the asynchronous access to the network.
DNS Resolver	This is where the queries to the DNS servers are performed.
SIP Core / Application	This is where the application uses the SIP Context and services, along with transaction management. It is important that the part of the application that interacts directly with M5T SIP SAFE be running within the same thread as the SIP Core / Application servicing thread. None of the calls to the SipCore objects and the services are synchronized or protected by the stack in any way.

It is possible to configure M5T SIP SAFE to run within up to three different servicing threads, one for each above modules. It is up to the application to create the servicing threads and configure them for these modules. An application can choose to run all modules within a single servicing thread, or run all modules separately in different servicing threads.

Since standard OS calls for DNS queries are all blocking calls, it is usually quite important for the DNS resolver module to run within its own thread.

Applications that must support a high throughput of SIP message should probably make sure that each module runs within its own servicing thread. Devices with low SIP message throughput should run both the SIP Transport and the SIP Core / Application modules within the same servicing thread.

Creating a Servicing Thread

The servicing thread is an ECOM object and the application must create it by using the ECOM factory mechanism.

Please refer to the M5T Framework documentation for more information regarding the creation of servicing threads and ECOM objects.

Associating Servicing Threads with Modules

The application must use the ISipCoreConfig interface to associate the servicing threads with various modules that require a thread to run. This is easily done by calling one of the following methods from ISipCoreConfig:

```
//  Summary:
//      Sets the thread that will be used by the SIP core and transactions
//      modules.
//-----------------------------------------------------------------------------
virtual mxt result SetCoreThread(IN IEComUnknown* pThread) = 0;

//  Summary:
//      Sets the thread that is used by the SIP transport module and the
//      sockets it uses.
//-----------------------------------------------------------------------------
virtual mxt result SetTransportThread(IN IEComUnknown* pThread) = 0;

//  Summary:
//      Sets the thread that is used to serially process DNS requests.
//-----------------------------------------------------------------------------
virtual mxt_result SetDnsResolverThread(IN IEComUnknown* pThread) = 0;
```

Sharing the Application's Thread

It is easy for applications built over the M5T Framework to actually use and run within a servicing thread that would be shared with the SIP Core module. However, applications that are using their own threading mechanism instead of using the servicing thread must have a proper way to work together with the SIP Core's servicing thread.

It is possible, when creating a servicing thread, to configure it in such a way that it uses an external thread instead of actually starting a system thread. There are two different calls to the Activate method of the IActivationService:

```
// Creates a system thread
//------------------------
virtual mxt result Activate(IN const char* pszName = NULL,
                        IN uint32 t uStackSize = 0,
                        IN CThread::EPriority ePriority = CThread::eNORMAL) = 0;

// Periodic activation using application's thread.
//-----------------------------------------------
virtual mxt result Activate(IN uint64 t uTimeoutMs,
                            OUT bool* pbReadyToRelease) = 0;
```

The servicing thread implements this IActivationService interface. Perform an ECOM QueryIf for this interface on a servicing thread.

The first version of the Activate method actually creates a system thread and this thread is used by the servicing thread to offer threading services to the objects attached to it.

The second version of the Activate method is very useful when an application already has its own threading mechanism and wants to share an already existing thread with a servicing thread. The application must periodically call this second Activate method to provide thread time to the servicing thread, which is then dispatched to all objects attached to it.

Running M5T SIP SAFE without Threads

It is possible to run all of M5T SIP SAFE within a single application thread or process by creating servicing threads that do not start an internal thread and by periodically calling the proper Activate method of the IActivationService interface.

Message Queues in the System

Figure 12 provides a simplified sequence diagram illustrating the message queues in M5T SIP SAFE. Each time a function call is done from one servicing thread to another, the function call is processed through a message queue and managed asynchronously. Almost all such function calls are managed asynchronously, meaning that the servicing thread of the function caller can do other things while the servicing thread where the function is being executed processes the request.

Synchronous calls, where the function caller actively waits for the function processing to terminate before continuing, is avoided.

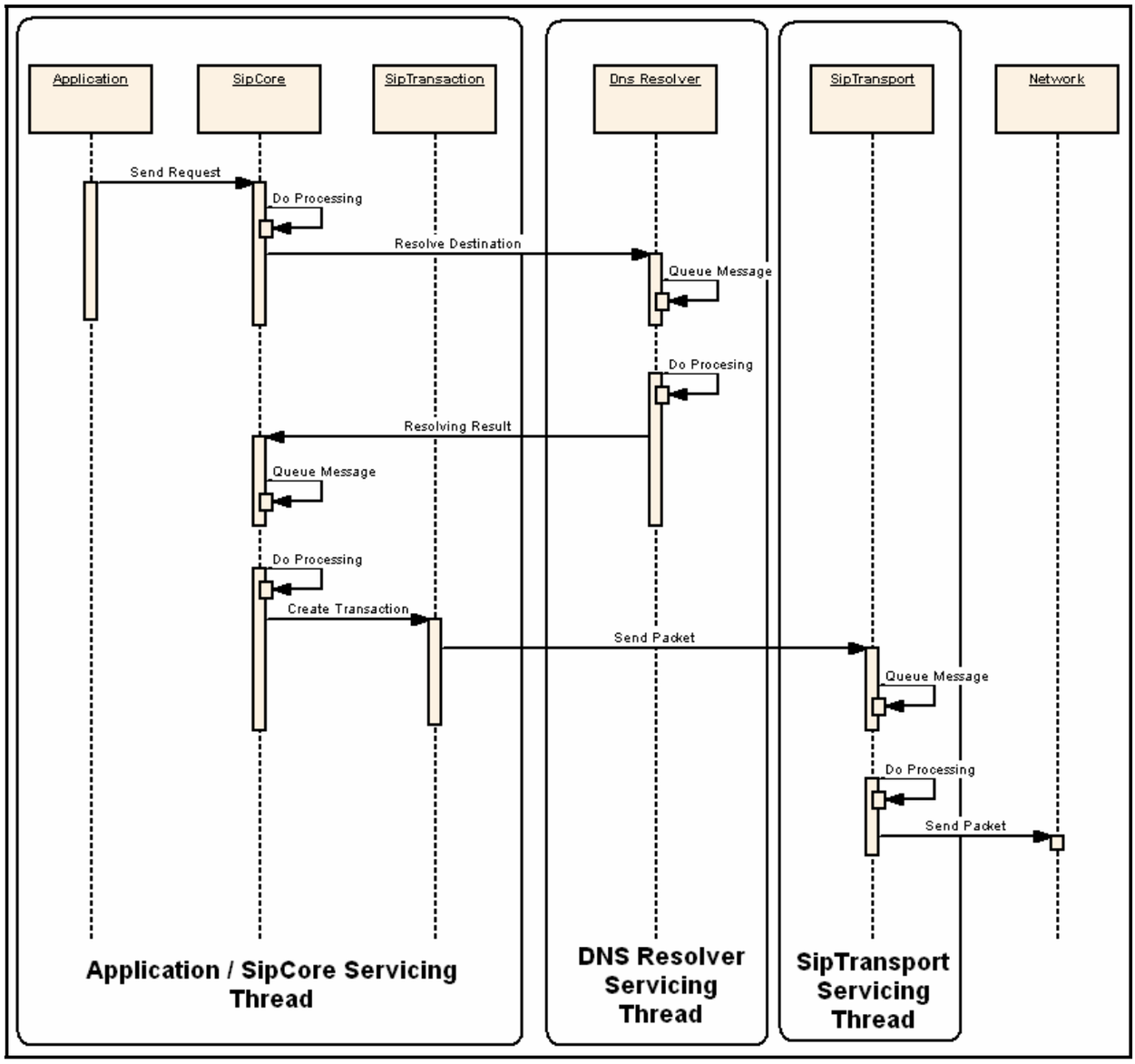


Figure 12: Threading - Highlights of Message Queues

2.6 Key Concept: Shutdown of the Stack

The shutdown of the stack can be achieved through the Shutdown() method of the ISipCoreConfig ECOM interface. Before shutting down the stack, the user must take special care of the following conditions:

- Clear() must be called on every ISipContext to which the application has a reference.
- Final responses must be sent on every pending ISipServerEventControl.
- The application must make sure it has released all of its references on any ISipServerEventControl, ISipClientEventControl and ISipContext it may have.

2.7 Key Concept: IPv6

The M5T SIP stack supports IPv6 and the dual IP stacks mode. M5T enables applications to take advantage of this next-generation Internet Protocol.

The application may however configure whether it wants to support only IPv4, only IPv6, or both IPv4 and IPv6. This is then used by the SIP stack to determine if it needs to perform A or AAAA queries when resolving addresses. Note that the MXD_SIPSTACK_IPV6_ENABLE_SUPPORT macro must be defined to be able to use IPv6. If MXD_SIPSTACK_IPV6_ENABLE_SUPPORT is not defined, IPv4 is enabled by default. If it is defined, IPv4 and IPv6 are enabled by default.

To configure the supported IP version, the application must call the SetSupportedIpVersion () method of the ISipCoreConfig ECOM interface as in the following example:

```
ISipCoreConfig* pConfig = NULL;

CreateEComInstance(CLSID CSipCoreConfig,
                   NULL,
                   IID ISipCoreConfig,
                   OUT reinterpret cast<void**>(&pConfig));

if (pConfig != NULL)
```

```
{
    pSipCoreConfig->SetSupportedIpVersion(ISipCoreConfig::eCONFIG IPV4);
    pConfig->ReleaseIfRef();
}
```

2.8 Key Concept: Spiralling Service

The spiralling service is an optimization in the SIP stack. The purpose of this service is to detect packets that are destined to the stack itself (i.e. sent in loopback). Instead of sending the packet up to the IP stack, the packet is detected at the transport layer of the SIP stack and returned back immediately. The parsing and serializing of the packet and the process of sending the packet in the IP stack of the operating system is then avoided.

2.9 Key Concept: Session Stateful Proxy Service Forking

The Session stateful proxy service has been modified to use the new dialog grouper mechanism. In order to support forking with the service, an ISipForkedDialogGrouperMgr must be set in the service by using ISipSessionStatefulProxySvc::SetForkedDialogGrouperMgr. The manager will then be notified when a supplemental context needs to be created to handle a packet. The manager should create a context similar to the one from which the packet was received and then use ISipForkedDialogGrouper to handle the request.

2.10 Key Concept: Persistent Connection Service

There are two types of connections available in the SIP stack: persistent connections and automatic connections.

Automatic connections are created by the stack when it needs to send a packet to a peer. An example of such a connection is an ephemeral UDP connection to send an INVITE. The lifetime of these connections is handled entirely by the stack and may be deleted if more resources are needed for creating additional connections or if an error occurs on a connection.

A persistent connection is not an ephemeral connection: the SIP-UA continuously monitors the state of the connection and reports any state changes to an application. State changes include connection errors, connection establishment and connection termination. The SIP-UA also automatically tries to reconnect the connection if an error occurs. They can be used for UDP, TCP or TLS connections.

An application is able to create and manage its own connections by using the CSipPersistentConnectionList.

```
ISipCoreConfig* pConfig = NULL;
mxt opaque opqPersistentConnection = 0;
mxt result res = resS OK;
uint16 t uLocalPort = 0;
ISipPersistentConnectionMgr* pMgr = this;

CreateEComInstance(CLSID CSipCoreConfig,
                   NULL,
                   IID ISipCoreConfig,
                   OUT reinterpret cast<void**>(&pConfig));

if (pConfig != NULL)
{
    CSipPersistentConnectionList* pPersistentConnectionList =
          pConfig->GetPersistentConnectionList();

    if (pPersistentConnectionList)
    {
        res = pPersistentConnectionList->Establish(eTCP,
                                                   opqLocalAddr,
                                                   uLocalPort,
                                                   "www.peeraddress.com",
                                                   pMgr,
                                                   opqPersistentConnection);

// Or
        CSocketAddr peerAddr("192.169.0.1".5060);
        res = pPersistentConnectionList->Establish(eTCP,
                                                   opqLocalAddr,
                                                   uLocalPort,
                                                   peerAddr,
                                                   pMgr,
                                                   opqPersistentConnection);

    }
```

The code sample above shows the basics of creating a TCP connection to www.peeraddress.com or to a specific IP address. The opqLocalAddr opaque parameter is the opaque value returned by a call to ISipCoreConfig::AddLocalAddress as described in 3.1.

Once a connection is successfully established, the ISipPersistentConnectionMgr:: EvConnectionEstablished event is reported to the application.

2.10.1 Terminating a Persistent Connection

A persistent connection will stay alive as long as the application wants. To close the connection, the application must explicitly request it by using the CSipPersistentConnectionList::Terminate method.

```
CSipPersistentConnectionList* pPersistentConnectionList =
          pConfig->GetPersistentConnectionList();

if (pPersistentConnectionList)
{
    res = pPersistentConnectionList->Terminate(opqPersistentConnection);
}
```

This closes the persistent connection that is related to the opaque opqPersistentConnection.

2.10.2 Persistent Connection Events

Once Establish has been called, the following events can be called on the ISipPersistentConnectionMgr:

- EvConnectionEstablished: The persistent connection on the opaque has been successfully connected between the local and peer address specified. This connection is active and can be used by the application to send and receive SIP packets.
- EvConnectionTerminated: The application has successfully terminated the persistent connection with the CSipPersistentConnectionList::Terminate method.
- EvErrorOnConnection: Reports that the stack was not able to establish or re-establish a connection to the peer or that an error occurred on an established connection. This event is reported once per unsuccessful establishment, re-establish attempt or error. An error code is returned for the application to use if it needs to report a specific error.

2.10.3 DNS Failover Behaviour

If a FQDN is specified in the call to Establish, a DNS query is done by the CSipPersistentConnectionList and a connection is established to the first SRV record returned. The behaviour in the case of a connection error then depends on whether or not failover is configured by using the `CSipPersistentConnectionList::SetAllowFailover` method.

```
CSipPersistentConnectionList* pPersistentConnectionList =
          pConfig->GetPersistentConnectionList();

if (pPersistentConnectionList)
{
    pPersistentConnectionList->SetAllowFailover(true);
// Or
    pPersistentConnectionList->SetAllowFailover(false);
}
```

If true is passed to the SetAllowFailover method, all of the entries returned by the DNS query are tried before a new DNS query is made. This is the default value. This means that if the connection to the first address fails, the second and subsequent values returned by the DNS query are tried. If all entries are tried, then a new DNS query is done.

If false, the stack reissues a new DNS query and takes the first value.

2.10.4 Configuration of Connection Re-Establishment

By default, a persistent connection retries to establish itself exponentially every 30000 milliseconds and capping at 480000 milliseconds.

This behaviour can be configured by using the CSipPersistentConnectionList::SetRetryConfig method.

```
CSipPersistentConnectionList* pPersistentConnectionList =
          pConfig->GetPersistentConnectionList();

if (pPersistentConnectionList)
{
    pPersistentConnectionList->SetRetryConfig(eRETRY_EXPONENTIAL,
                                              uTimeout1Ms,
                                              uTimeout2Ms,
                                              uTimeout3Ms);
}
```

The SetRetryConfig has three timer modes:

- eRETRY_PERIODIC: periodically tries to establish the connection, once every uTimeout1 milliseconds.
- eRETRY_EXPONENTIAL: retries to connect the persistent connection exponentially, starting at uTimeout1ms milliseconds and capping at uTimeout3ms milliseconds.
- eRETRY_OUTBOUND: retries to establish the connection by using the formula as specified in the outbound draft: W = min(max-time, (base-time * (2 ^ consecutive-failures))).

2.10.5 Using the ISipPersistentConnectionSvc

The ISipPersistentConnectionSvc enables the use of persistent connection on the service to which it is attached. Once a persistent connection opaque is set in the ISipPersistentConnectionSvc, then all requests sent using this service use the specified opaque value. If this opaque value is NULL, then the default connection matching is used.

To set the opaque of a persistent connection, the ISipPersistentConnectionSvc::SetConnectionOpaque method is used.

It is always possible to know if a persistent connection opaque is set by using the GetConnectionOpaque method.

2.11 Key Concept: Outbound Connection Service

This connection service allows the application to manage outbound persistent connections (flows) when acting as a user-agent entity. The service selects the flow to use when sending a request and updates the SIP packets as described in the draft-ietf-sip-outbound-15. The application can use this service to associate flows with a context. It is possible to hold multiple flows to the same domain. As such, the service allows the application to specify which flow must be used when sending a request to a domain name for which there is more than one flow.

The outbound service can be used with or without the server locator service attached, but in the first case, the outbound service MUST be attached AFTER the server location. The behavior changes depending on where the service is attached. Attaching the outbound service on a context without the server location service allows bypassing DNS resolving and using an outbound connection directly. Attaching the service after the server locator allows using the DNS resolving before selecting an outbound connection.

The service can be configured with an ordered list of persistent connections to use. This list is used only when the outbound service is attached on a context without the server location. When outbound connections are configured, the content of the target (Request-URI or top Route) is ignored. The first available flow is selected and the peer address of the packet is set with the peer address of the selected flow.

Attaching the service AFTER the server locator allows load balancing as the DNS tells which target to try first when sending the request. In such a mode, the server locator handles the failover. When a DNS lookup is done and a persistent connection is not found in the flow list, the stack asks the application to create a new persistent connection by using the eEvConnectionNeeded( ) event, which is defined in the ISipOutboundConnectionMgr interface. In this case, the application has two possibilities: it can create a new persistent connection in an asynchronous way or it can refuse to create it. In the second case, the request will be blocked. See ISipOutboundConnectionMgr.h for more information.

REGISTER requests are handled in a different way. In this case, the stack does not ask the application to create a new persistent connection. If no server location service is attached, the stack uses only the first connection in the flow list. If the REGISTER can not be sent using this connection, the request is blocked. On the other hand, if the server location service is attached, the stack tries to send the REGISTER by using the available persistent connections and no new connection creation is requested. For more information, see the SetPersistentConnectionsPreferredOrder( ) method in ISipOutboundConnectionSvc.h.

The persistent connection service MUST not be attached to a context on which the outbound connection service is also attached. It is also the responsibility of the application to attach the service in the proper order. See the description of the ISipOutboundConnectionSvc interface.

2.11.1 Additional Persistent Connection Events

The following additional events can be reported when MXD_SIPSTACK_ENABLE_SIP_KEEP_ALIVE_SVC_SUPPORT is defined and the keep alive mechanism is used:

- EvStartKeepAliveResult: Reports that the keep alive mechanism used for outbound connection has been successfully started.
- EvSendResult:: Reports the status of the sending of a keepalive packet on the specified flow.

2.11.2 Keepalive Service and Persistent Connection List

The keep alive mechanism for connection-oriented (TCP, TLS over TCP) flows is handled by the stack itself. For datagram flows, it is the responsibility of the application to handle the keep alive mechanism (using STUN for example).

The following methods are available to the application to notify the CSipPersistentConnectionList of problems occurring with a flow or to control the keep alive mechanism:

- EvFlowSucceeded: first keep alive succeeded on the connection.
- EvFlowFailure: keep alive on the flow failed.
- StartKeepAlive: starts the CRLF keep alive on the flow.
- Send: Sends data by using the persistent connection for the keep alive.

Please refer to the CSipPersistentConnectioinList API reference for more details on these methods.

3. Getting Started

This section identifies the key points a programmer has to do in order to get started with the use of M5T SIP SAFE v4.1.

3.1 Local Addresses

M5T SIP SAFE v4.1 supports the dual IP stack. This allows the application to listen and use both an IPv4 and IPv6 stack.

The following are the key concepts of the SIP stack configured with multiple IPv4/IPv6 local addresses:

- When sending a request or a response to an IPv4 address, the SIP stack uses a local IPv4 address. The same is true for IPv6.
- When sending a request to an IPv4 address, the SIP stack uses a Via that is either an IPv4 address or an FQDN. The same is true for IPv6.
- When sending a request or response with an IPv4 destination address, the SIP stack uses a Contact that specifies either an IPv4 address or an FQDN. The same is true for IPv6.
- While sending a SIP request, the SIP stack can automatically switch from a local address to another when it tries to reach a destination address. This causes automatic Via and Contact headers update.
- An entity ID can be associated with a listening opaque and a SIP context. If specified, requests and responses sent from a context will only use the interface with the same entity ID. By default, 0 is set everywhere. See «*M5T SIP SAFE v4.1 - API Reference*» for additional information about how to configure entity ID.

3.1.1 Local Address Configuration

To achieve the above features, the SIP stack must be properly configured by using the ISipCoreConfig ECOM interface. Thus, local addresses must be added in the SIP stack configuration during the application initialization process.

The following example shows how to configure the SIP stack in the dual stacks mode:

```
// Obtains the ISipCoreConfig interface.
//--------------------------------------
ISipCoreConfig* pConfig = NULL;
CreateEComInstance(CLSID CSipCoreConfig,
                   NULL,
                   IID ISipCoreConfig,
                   OUT reinterpret cast<void**>(&pConfig));
MX ASSERT(pConfig != NULL);

// IPv4 network (default interface).
//----------------------------------
mxt opaque opqIPv4Address = 0;
CSocketAddr ipv4Addr("1.2.3.4");
CVector<CString>* pvecstrIPv4Fqdn = MX NEW(CVector<CString>);
pvecstrIPv4Fqdn->Append("MyIPv4Network.example.com");
pConfig->AddLocalAddress(ipv4Addr,
                         TO pvecstrIPv4Fqdn,
                         NULL,
                         OUT opqIPv4Address);

// IPv6 network.
//---------------
mxt opaque opqIPv6Address = 0;
CSocketAddr ipv6Addr(CSocketAddr::eINET6, "1234::ABCD");
CVector<CString>* pvecstrIPv6Fqdn = MX NEW(CVector<CString>);
pvecstrIPv6Fqdn->Append("MyIPv6Network.example.com");
pConfig->AddLocalAddress(ipv6Addr,
                         TO pvecstrIPv6Fqdn,
                         NULL,
                         OUT opqIPv6Address);

// Releases the ISipCoreConfig interface.
//---------------------------------------
pConfig->ReleaseIfRef();
```

In the above example, the SIP stack is configured with the following local addresses:

Table 12 - Local Address Example

Family	Address	FQDN
IPv4	1.2.3.4	MyIPv4Network.example.com
IPv6	[1234::ABCD]	MyIPv6Network.example.com

Warning
In version 4.1, at least one local address must be added in the SIP stack. A local address is required to enable socket listening, send SIP requests, and receive SIP responses.

3.1.2 Via Address Preference

The SIP stack v4.1 manages the Via header automatically. That header is generated by the SIP stack from the local addresses configuration according to the local address that is selected by the server location service when the SIP stack tries to reach a peer.

The Via address generation is made based on one of the following user preferences:

1. FQDN representation; this is the default configuration.
2. IP address representation.

The ISipCoreConfig::SetViaAddressType interface configures the Via address preference.

3.1.3 Routing Table Configuration

When adding a local address, the application may provide information on which remote addresses are reachable by that local address. This effectively builds a routing table required by the SIP stack to control which network interface to use when sending a SIP packet to a destination address.

The routing table is built from the pvecDestinations parameter of the AddLocalAddress method. This parameter holds a vector of accessible network structure.

3.1.3.1 Routing Table Example 1

Let's assume a device with an interface to 10.1.x.x, it could then configure m_address to 10.1.0.0 and m_mask to 255.255.0.0.

If the stack has to send a packet to 10.1.2.3, the destination address would be masked with m_netmask, yielding 10.1.0.0. Since the masked destination address and m_address match, the address for which this SAccessibleNetwork is configured would be used.

3.1.3.2 Routing Table Example 2

Let's assume a device that is able to reach the following networks from a single address, 10.2.5.100:

```
- 10.0.x.x
- 10.1.x.x
- 10.2.x.x
- 10.3.x.x
- 10.4.x.x
- 10.5.x.x
- 10.6.x.x
- 10.7.x.x
```

There are multiple ways in which the application could configure its list of accessible networks. One way would be to associate with its 10.2.5.100 local address multiple accessible networks:

```
- 10.0 network: m address: 10.0.0.0, m netmask: 255.255.0.0
- 10.1 network: m address: 10.1.0.0, m netmask: 255.255.0.0
- 10.2 network: m_address: 10.2.0.0, m_netmask: 255.255.0.0
- 10.3 network: m address: 10.3.0.0, m netmask: 255.255.0.0
- 10.4 network: m address: 10.4.0.0, m netmask: 255.255.0.0
- 10.5 network: m_address: 10.5.0.0, m_netmask: 255.255.0.0
- 10.6 network: m address: 10.6.0.0, m netmask: 255.255.0.0
- 10.7 network: m_address: 10.7.0.0, m_netmask: 255.255.0.0
```

Another way would be to tweak m_netmask and add a single entry for all networks:

```
- 10.[0-7] network: m_address: 10.0.0.0, m_netmask: 255.248.0.0
```

When sending to a destination that is part of one of the networks, let's say 10.4.3.65, the destination is masked with m_netmask. This yields 10.0.0.0. The masked destination matches m_address, thus the associated local address would be used to reach this destination.

When sending to a destination that is not part of one of the networks, let's say 10.9.22.98, the destination is masked with m_netmask. This yields 10.8.0.0. The masked destination does not match m_address, thus the associated local address would not be used to reach this destination.

Notice that the mask is applied in binary. Thus for this example, representing only the important part of the address in binary:

```
Destination     Dest.Binary         Mask                Masked Result
-------------------------------------------------------------------------
10.4.3.65       10.b00000100.3.65   255.b11111000.0.0   10.0.0.0
10.9.22.98      10.b00001001.22.98  255.b11111000.0.0   10.b00001000.0.0
```

3.1.4 Local Address and Routing Table Configuration Example

An application has two local addresses: 10.2.5.100 to access a private network and 205.237.248.241 to access the public internet.

- From its 10.2.5.100 address, it has access to 8 networks, from 10.0.x.x to 10.7.x.x.
- From its 205.237.248.241 address, it has access to the public network.

The application would then configure its 10.2.5.100 as its first address with a single SAccessibleNetwork that has the following parameters:

```
        - m address = 10.0.0.0
        - m_netmask = 255.248.0.0
```

The application would afterwards configure its 205.237.248.241 address as its second address, but this time, it would specify no accessible networks, meaning that this is the default address to use if no previous address can be used.

The following is a code example:

```
// Private network configuration.
//--------------------------------
CSocketAddr privateAddr("10.2.5.100");

ISipCoreConfig::SAccessibleNetwork stLocalAccessibleNetwork;
stLocalAccessibleNetwork.m address.SetAddress("10.0.0.0");
stLocalAccessibleNetwork.m_netmask.SetAddress("255.248.0.0");

CVector<ISipCoreConfig::SAccessibleNetwork>* pvecLocalDestinations =
    MX_NEW(CVector<ISipCoreConfig::SAccessibleNetwork>);
pvecLocalDestinations->Append(stLocalAccessibleNetwork);

mxt_opaque opqprivateAddress = 0;
pConfig->AddprivateAddress(privateAddr,
                           NULL,
                           TO pvecLocalDestinations,
                           OUT opqprivateAddress));

// Public network configuration.
//-------------------------------
CSocketAddr publicAddr("205.237.248.24");

CVector<CString>* pvecPublicStrFqdn = MX NEW(CVector<CString>);
pvecPublicStrFqdn->Append("m5t.com");

mxt opaque opqPublicAddress = 0;
pConfig->AddLocalAddress(publicAddr,
                         TO pvecPublicStrFqdn,
                         NULL,
                         opqPublicAddress));
```

3.2 Listening Mechanism

3.2.1 Description

The following methods are used to start and stop listening on a socket. They are accessible through the ISipCoreConfig ECOM interface.

- ListenA
- StopListeningA

The above methods are executed asynchronously and their return codes are returned through the EvCommandResult method of the ISipCoreUser interface. Consequently, applications must implement the ISipCoreUser interface as in the following example:

```
//==========================================================================
//====  ISipCoreUser METHOD  ===============================================
//==========================================================================
void CMySipEngine::EvCommandResult(IN mxt_result res, IN mxt_opaque opq)
{
    m_pSyncSemaphore->Signal();
}

//==========================================================================
//====  INITIALIZATION METHOD  =============================================
//==========================================================================
void CMySipEngine::Initialize()
{
    // Adds a network interface.
    //--------------------------
    CSocketAddr localAddr("1.2.3.4 ");
    mxt_opaque opqLocalAddress = 0;
    pConfig->AddLocalAddress(localAddr, NULL, NULL, OUT opqLocalAddress);

    // Starts listening on UDP with the above local address.
    //------------------------------------------------------
    mxt_opaque opq = eSTART_LISTENING_UDP;
    mxt_opaque opqListen = 0;
    pConfig->ListenA(opqLocalAddress,
                     5060,
                     eUDP,
                     this,
                     opq,
                     OUT opqListen);

    // Waits for the asynchronous result.
    //-----------------------------------
    m_pSyncSemaphore->Wait();
}
```

3.3 Creating SIP applications

The following subsections describe at a high level how to build applications using M5T SIP SAFE: a proxy and a user agent.

Table 13 highlights the general steps used in building any M5T SIP SAFE application.

Table 13: General Steps for Creating M5T SIP SAFE Applications

Step	Description
Step 1 – Static Stack Configuration	The stack supports a number of configurable items that can be enabled or disabled at compile time through various `#define`. To do so, the application must create a file named PreSipStackCfg.h that will contain all the `#define` to be used by the stack. Please refer to the Config package in the API reference, which describes this process along with all existing `#define`. The M5T Framework must also be configured in PreMxConfig.h and PreFrameworkCfg.h. See the API reference and the Programmer's Guide of the M5T Framework.
Step 2 – Stack Initialization	Before any APIs of the stack can be used, it must first be initialized. • Initialize the framework with CFrameworkInitializer. • Initialize the SIP stack with CSipStackInitializer. Please refer to the API reference for more information.
Step 3 – Implement the ISipCoreUser	As depicted in section 2.1 Key Concept: Stack / Application Interfaces, some class in the application must inherit from the ISipCoreUser. The main responsibility of this class is to properly dispatch incoming SIP packets that do not match any existing transactions. As detailed in section 2.2 Key

Step	Description
	Concept: Handling Unmatched Packets, the application must take certain steps when handling incoming packets. The core user must be configured in the stack by calling `ISipCoreConfig::SetCoreUser()`.
Step 4 – Dynamic Stack Configuration and Startup	The stack must be configured according to the application's specifications before it can be used. To configure the stack, the application must use the ISipCoreConfig interface. This is an ECom object created through the standard ECom creation mechanism. There are a few things that must be created and configured with the ISipCoreConfig before starting the stack: • Create and set the DNS, Core and Transport threads. Section 2.5 Key Concept: Threading describes this in more details. • Configure the local interfaces. Section 3.1 Local Addresses describes this in more details. • Start listening on interfaces. Section 3.2 Listening Mechanism describes this in more details. • Set a default dialog matcher list. After the configuration is done, the stack can be started by using the `ISipCoreConfig::Startup()` method. Please refer to the documentation of ISipCoreConfig in the API reference to obtain more information about all the possible configuration options supported by the stack.
Step 5 – Implement the desired service's managers	Manager classes are responsible for managing events generated by the stack – they are the callbacks. They implement the manager interfaces defined by the stack, each service having its own manager interface to report events. Typically, an application would define a number of these manager classes, depending on the services used and the application architecture. For instance, a CRegistrarMgr class could implement the ISipRegistrationMgr interface used by the registration service (ISipRegistrationSvc). A CBasicCallMgr class could implement both the ISipSessionMgr and the ISipGenericMgr interfaces used by the session (ISipSessionSvc) and the generic (ISipGenericSvc) services, respectively. And so forth.

3.3.1 Getting Started with Proxies

Let's start with a potential high-level design of the proxy. Let's assume that the application wants to be able to manage registration requests and act as a transaction stateful or stateless proxy.

Figure 13 is an UML class diagram representing a proposed design. The application classes are highlighted with the "app" stereotype. These classes must be implemented over the M5T SIP stack. The other classes and interfaces all exist within the M5T SIP stack.

In this diagram, the entry point for the application is the CStackController class, which inherits from the ISipCoreUser interface. The application needs to create only a single instance of this class. The responsibilities are highlighted in the three requirements right over it. It manages the initialization and shutdown of the M5T SIP Stack and also handles the reception of packets that do not match any existing transactions.

The sequence diagrams in Figure 14 and Figure 15 show what should happen upon receiving various packets, given our proposed design.

According to Table 13:

- CStackController is responsible for Step 3: Implement the ISipCoreUser.
- CStackController is responsible for Step 4: Dynamic Stack Configuration and Startup.
- CStatefulProxy, CStatelessProxy and CSipRegistrar are responsible for Step 5: Implement Manager Classes.

To detail the example further, let's describe the tasks performed by CStatefulProxy.

- CStatefulProxy creates a new SIP context and attaches it to the server location service (ISipServerLocationSvc) and the transaction stateful proxy service (ISipTransactionStatefulProxySvc). Note that the SipServerLocation service is required in order to send a packet. Even if only IP addresses are used, the service is necessary in order to manage transports.
- It then configures itself as the manager of the proxy service – because it implements the manager interface of this service (ISipTransactionStatefulProxyMgr).
- When receiving an unhandled packet via ISipCoreUser::EvOnPacketRecieved(), it passes the incoming packet to its ISipContext and waits for events from the proxy service. These events come through the aforementioned ISipTransactionStatefulProxyMgr interface.

CStatelessProxy and CSipRegistrar perform very similar functions and are not detailed here.

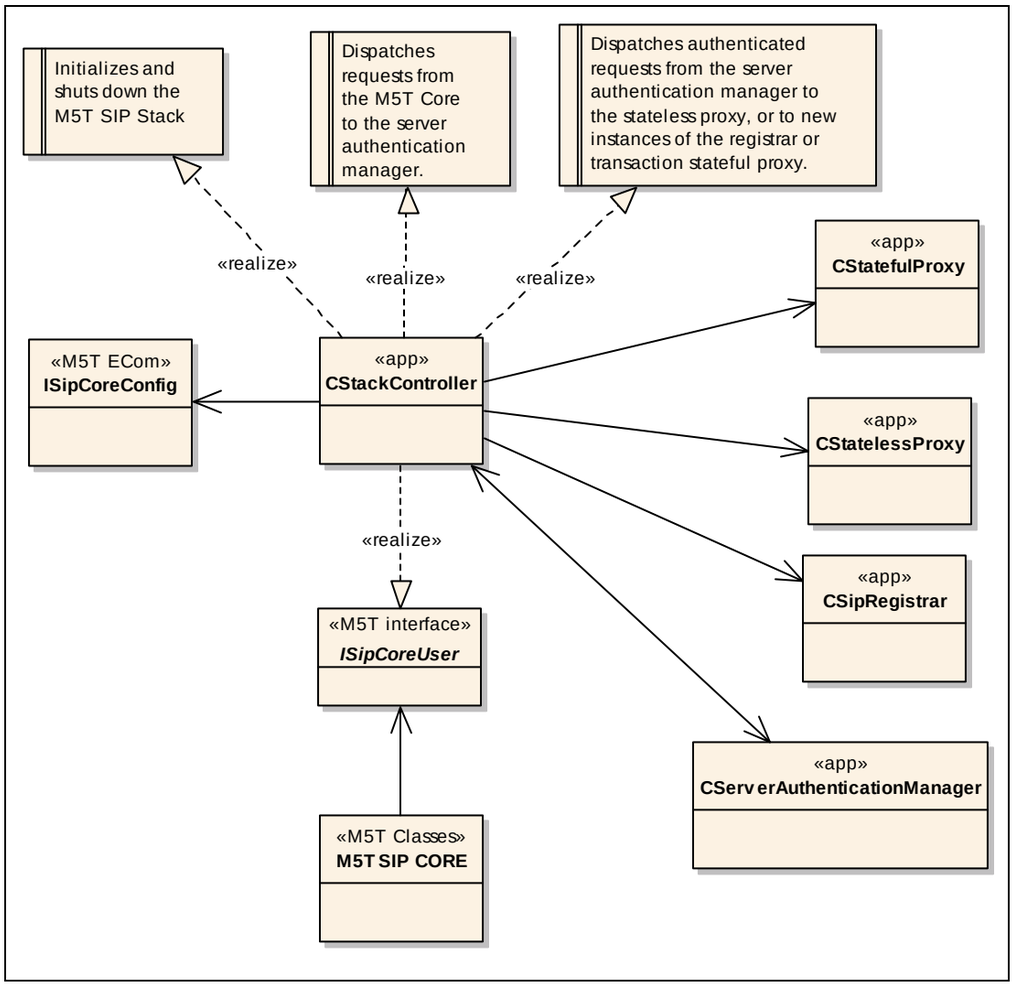


Figure 13: Sample Proxy Application Design

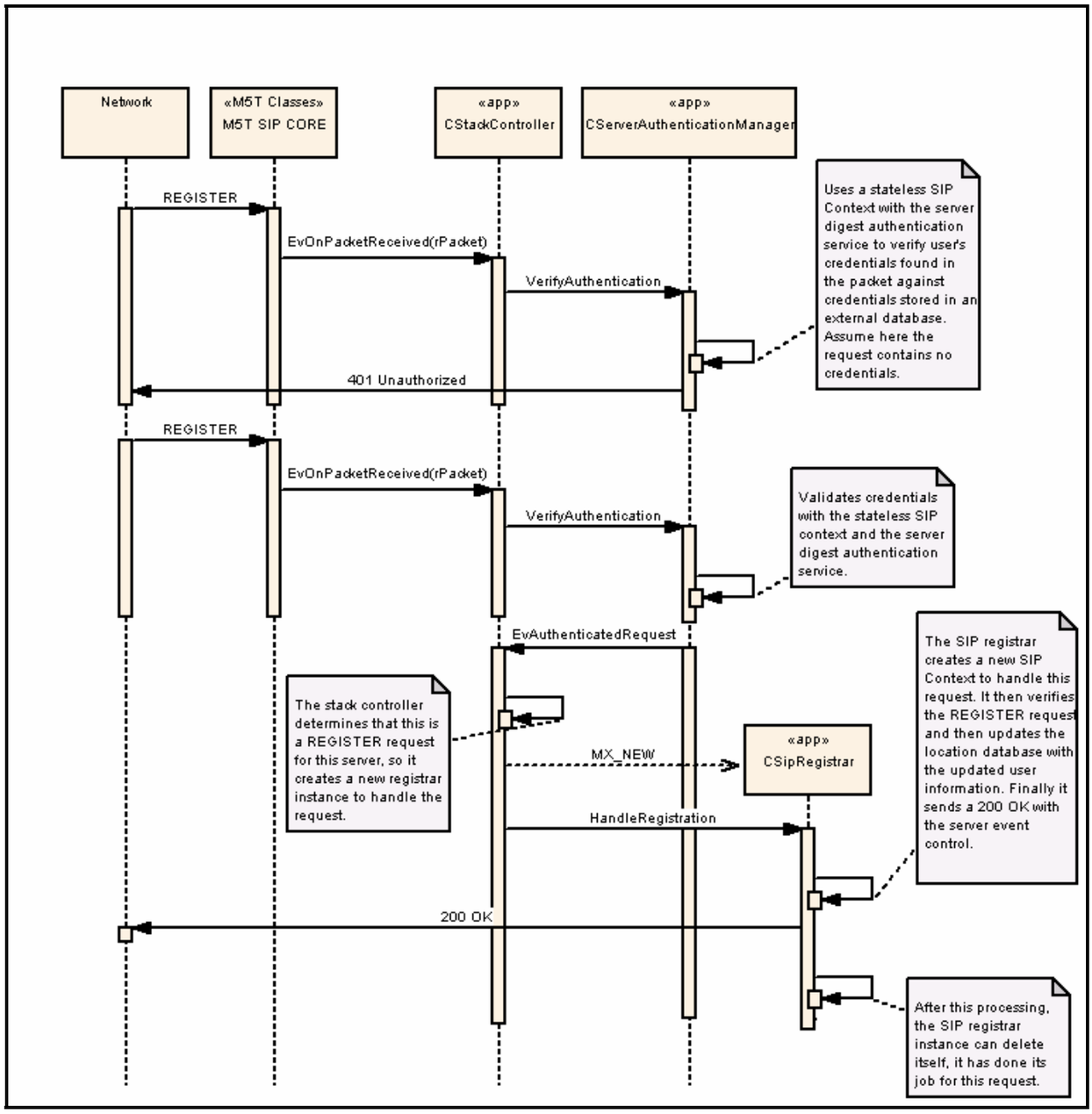


Figure 14: Proxy Handling of REGISTER Requests

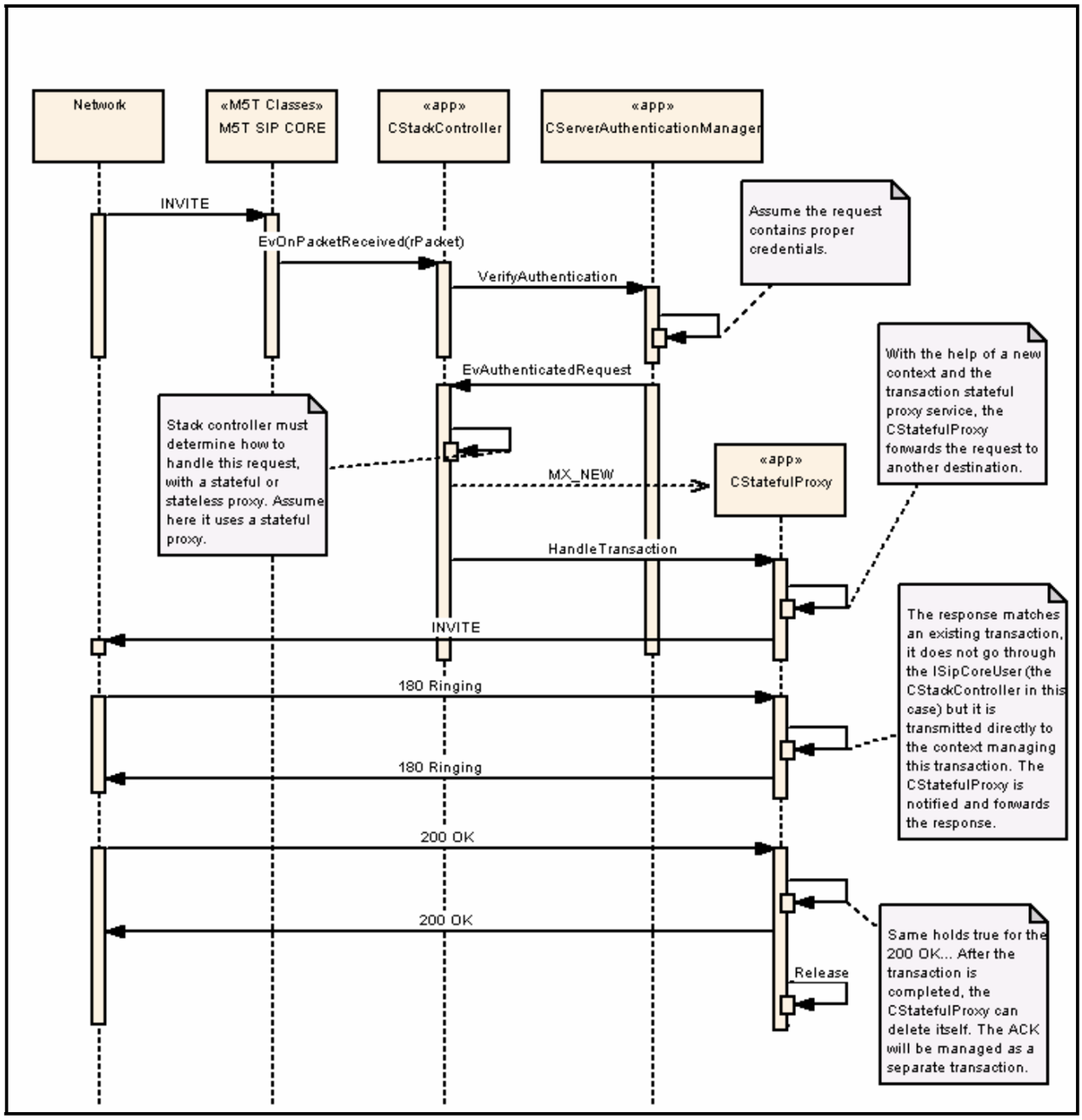


Figure 15: Proxy Handling of Other Requests

3.3.2 Getting Started with User Agents

The previous section described the high level design of a proxy application. Let's now turn our attention to building a SIP user agent. The class diagram for this application is shown in Figure 16.

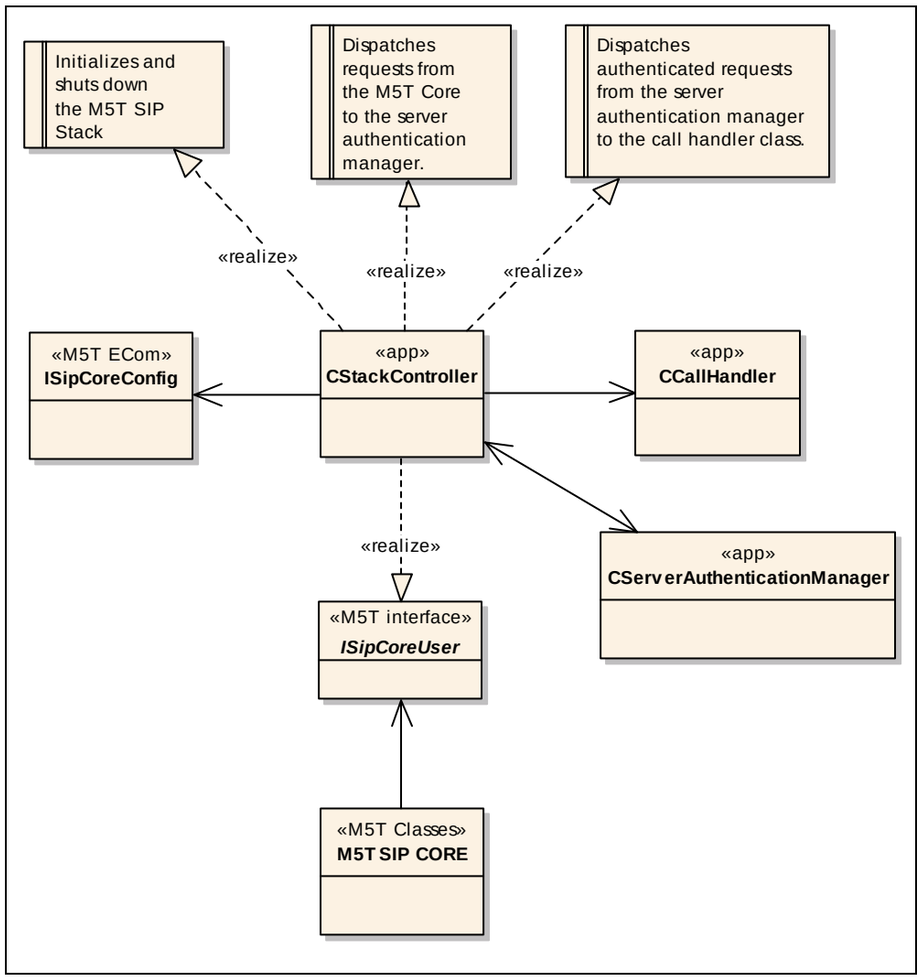


Figure 16: Sample User Agent Application Design

Both applications are very similar thanks to the standardized interfaces provided by M5T SIP SAFE. In fact, almost all classes have the same name and perform identical tasks as in the proxy application. The only new class is CCallHandler and it performs tasks very similar to CStatefulProxy:

- Implements one or more manager interfaces.
- Creates a SIP Context, attaching the appropriate service(s), for instance, the session service (ISipSessionSvc). Note that the SipServerLocation service is required in order to send a packet. Even if only IP addresses are used, the service is necessary in order to manage transports.
- When receiving an unhandled packet, passes the packet to the SIP Context and then waits for incoming events through the manager interface(s).
- Contrary to CStatefulProxy, initiates outgoing INVITEs.

Handling of incoming requests is almost identical to what was done for the proxy application in Figure 15. Please refer to this figure for more information. Handling of outgoing INVITEs is detailed in Figure 17.

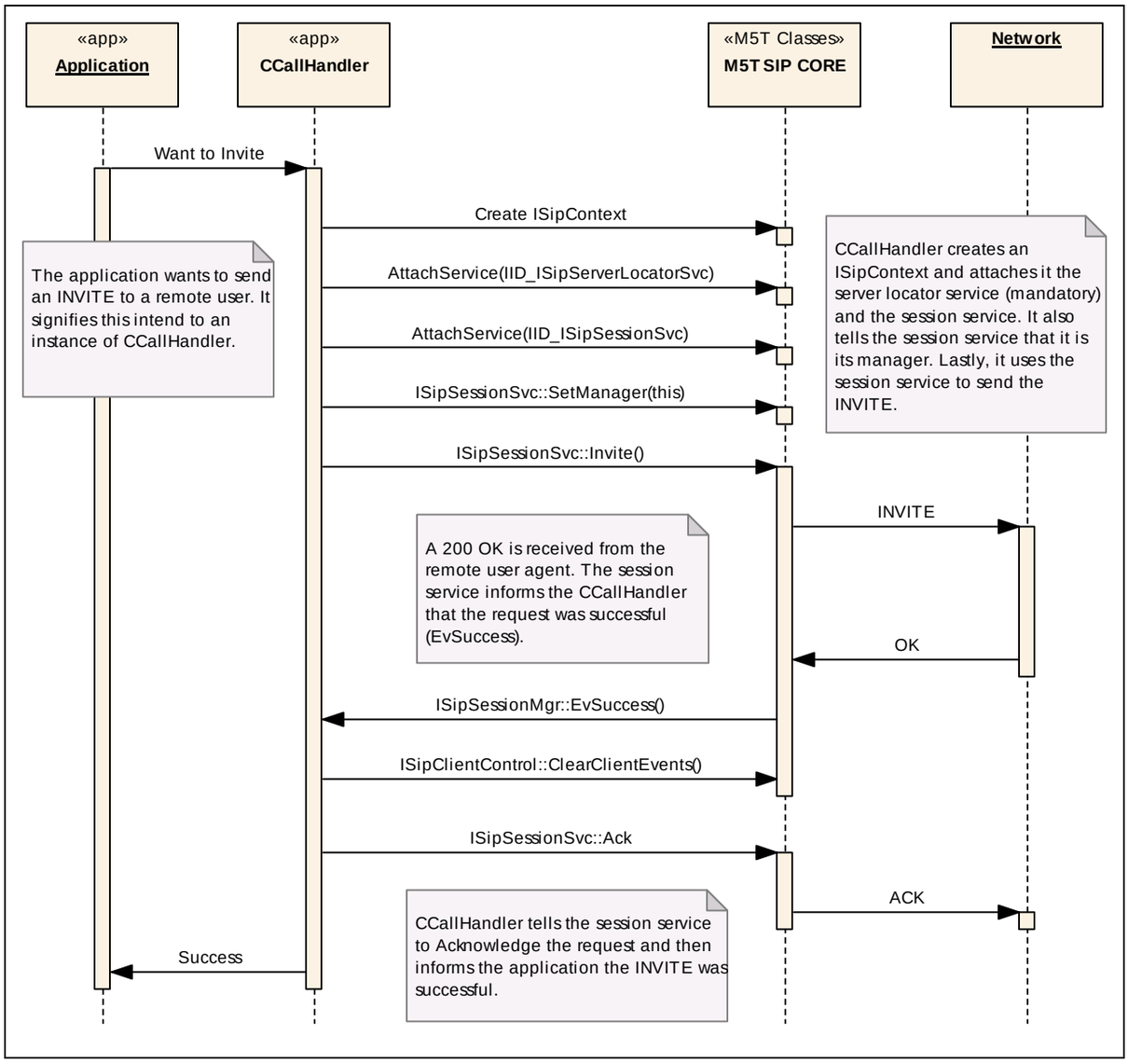


Figure 17: User Agent Outgoing INVITE

3.4 Forking

Forking is a situation that occurs when a proxy forwards a request to more than one host. Each of these hosts can then in turn send a response to the original UA through the proxy that forked the request. Each of these responses will carry its own To-tag. The additional responses that are received in addition to the first one are called supplemental responses.

3.4.1 UA Forking

3.4.1.1 Receiving an Additional Response

The first received response matches with the context that sent the request and is considered the original context. Any other non 100 Trying 1xx or 2xx responses that are received from another peer cannot be handled on the same context.

When a supplemental response comes in on the context, the ISipSessionMgr::EvNewSessionNeededForOriginalInviteResponse method is called. The application must create a new ISipContext, attach and configure the services on the context to properly handle this new response The same target must be set on the supplemental ISipUserAgentSvc as on the original one. ISipSessionSvc::HandleOriginalInviteResponseNewSession MUST then be called on the ISipSessionSvc parameter of the EvNewSessionNeededForOriginalInviteResponse call. The packet will then follow normal stack processing as shown in Figure 18 Call Establishment with Forking.

It is possible for the application to receive more than one 2xx response to an INVITE. If this is the case, it is up to the application to decide exactly how it wants to handle this case. Please read the M5T SIP SAFE v4.x - Best Current Practices - Forking.pdf document for more information and notes on how an application can handle this.

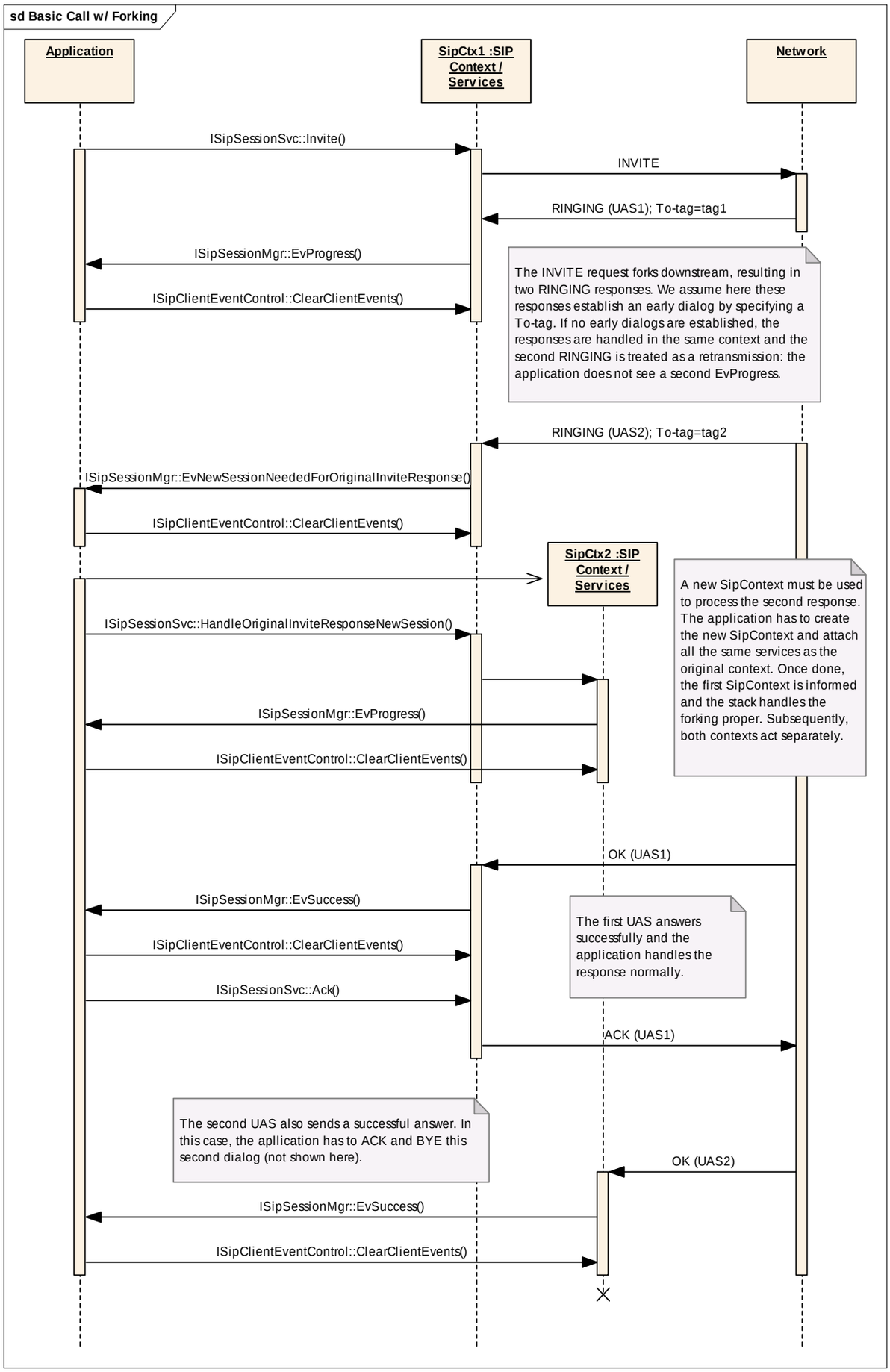


Figure 18 Call Establishment with Forking

3.4.1.2 Early Dialogs

Once a supplemental response has been received and a new context created, this new context is considered as a completely independent context from the original one with its own lifetime.

The reception of a final 2xx response on one of the contexts starts a timer in every context except the one that received the response. This timer waits for a 2xx response on each of the early dialogs. If one is not received, a final negative response is sent on each context when its timer fires and the event ISipSessionMgr::EvFailure is generated.

3.4.1.3 Receiving a Final Negative Response

When the application receives a final negative response from the proxy, it means it will never receive another type of final response. In this case, all contexts that are created on the case of the forked request receive a final negative response. The originator receives the final negative response from the network and the rest have a 487 request terminated response sent on them by the stack.

4. Sequence Diagrams

This section presents a number of high-level sequence diagrams to help understand the interactions between the application, M5T SIP SAFE, and the external agents (network).

4.1 SIP Context Lifecycle

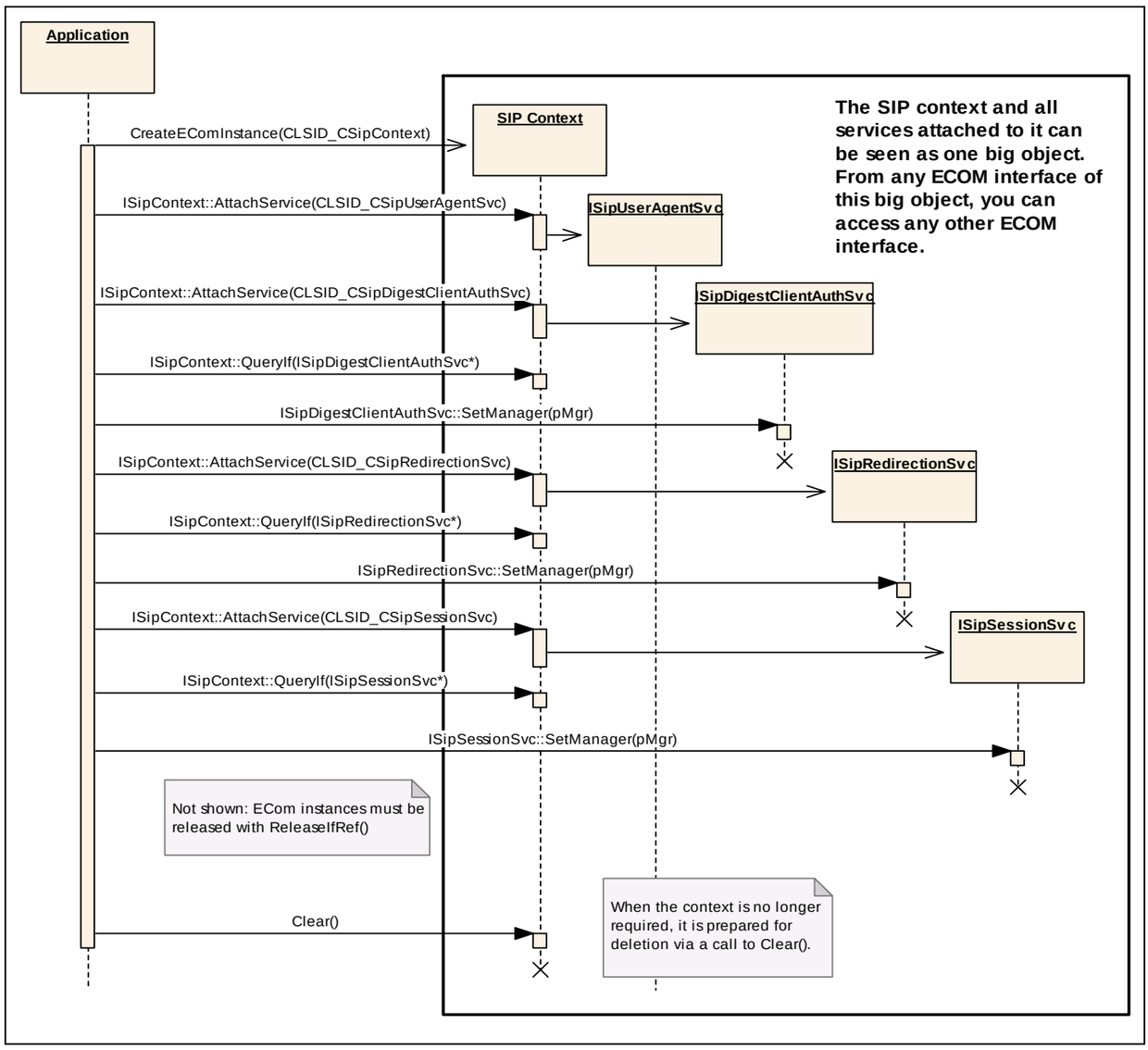


Figure 19: SIP Context Lifecycle

4.2 Successful Registration with Registration Refresh

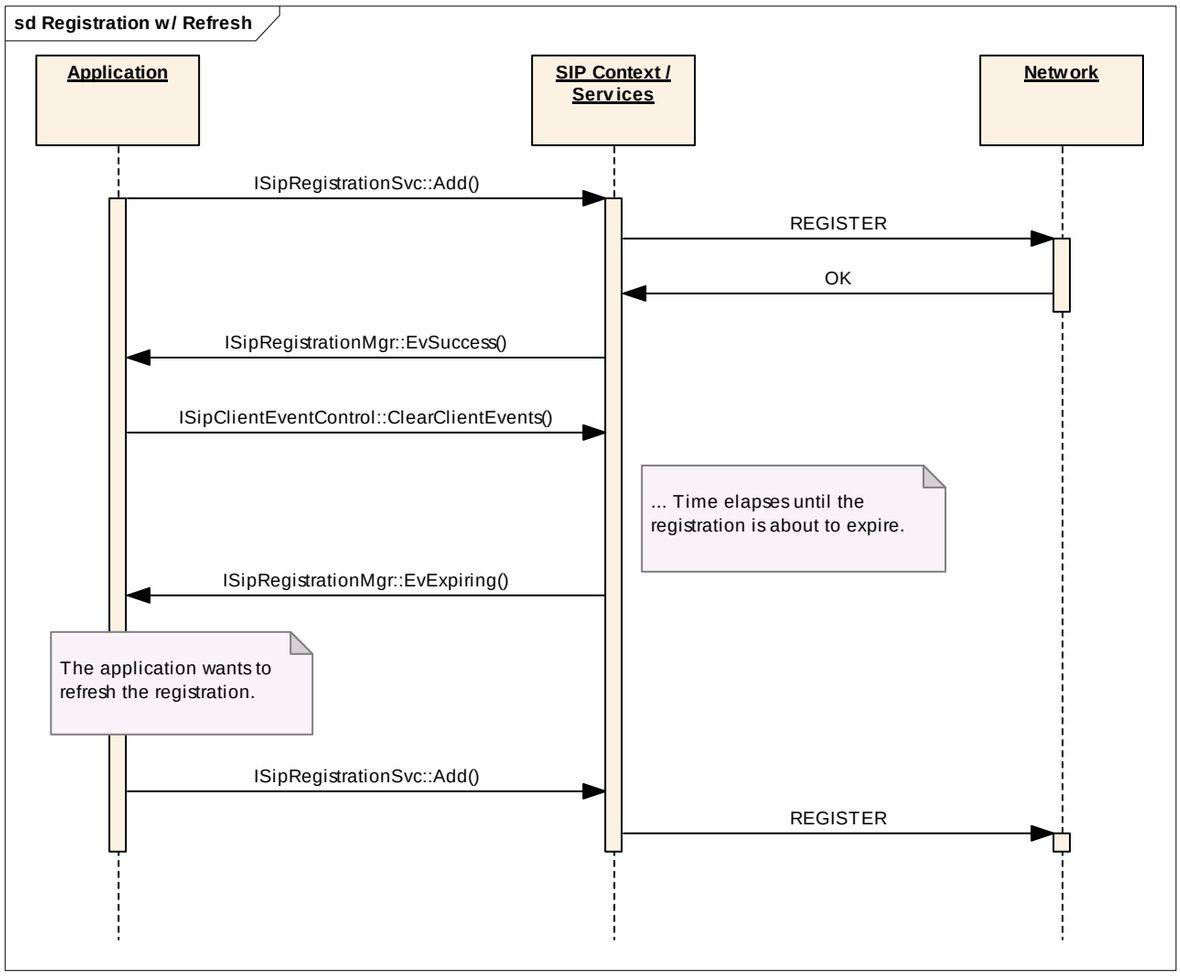


Figure 20: Successful Registration with Registration Refresh

4.3 Successful Unregistration

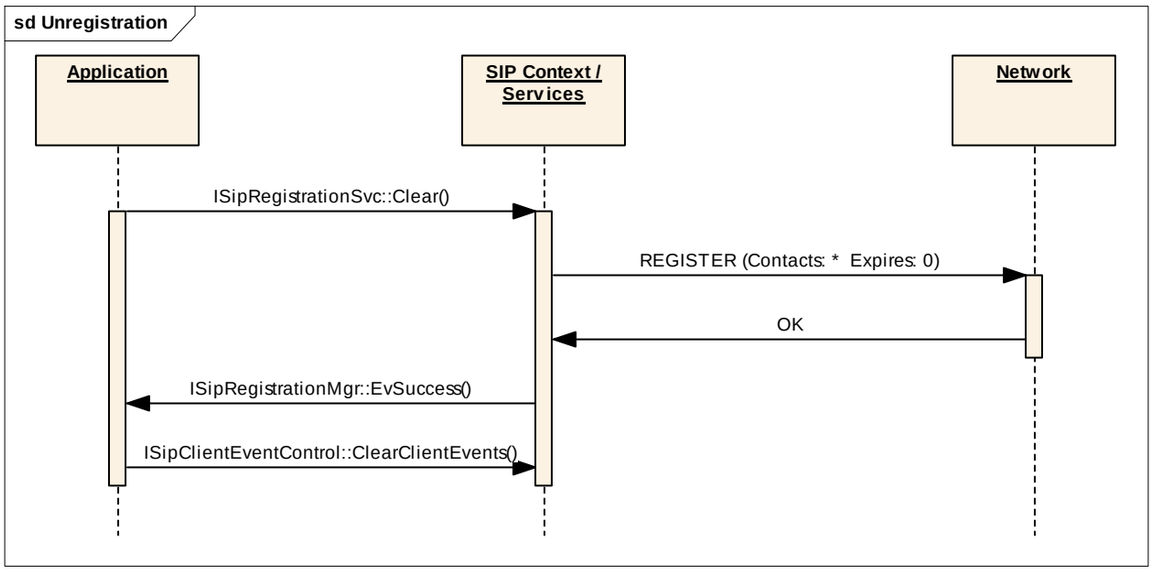


Figure 21 : Successful Unregistration

4.4 Registration with a Challenge

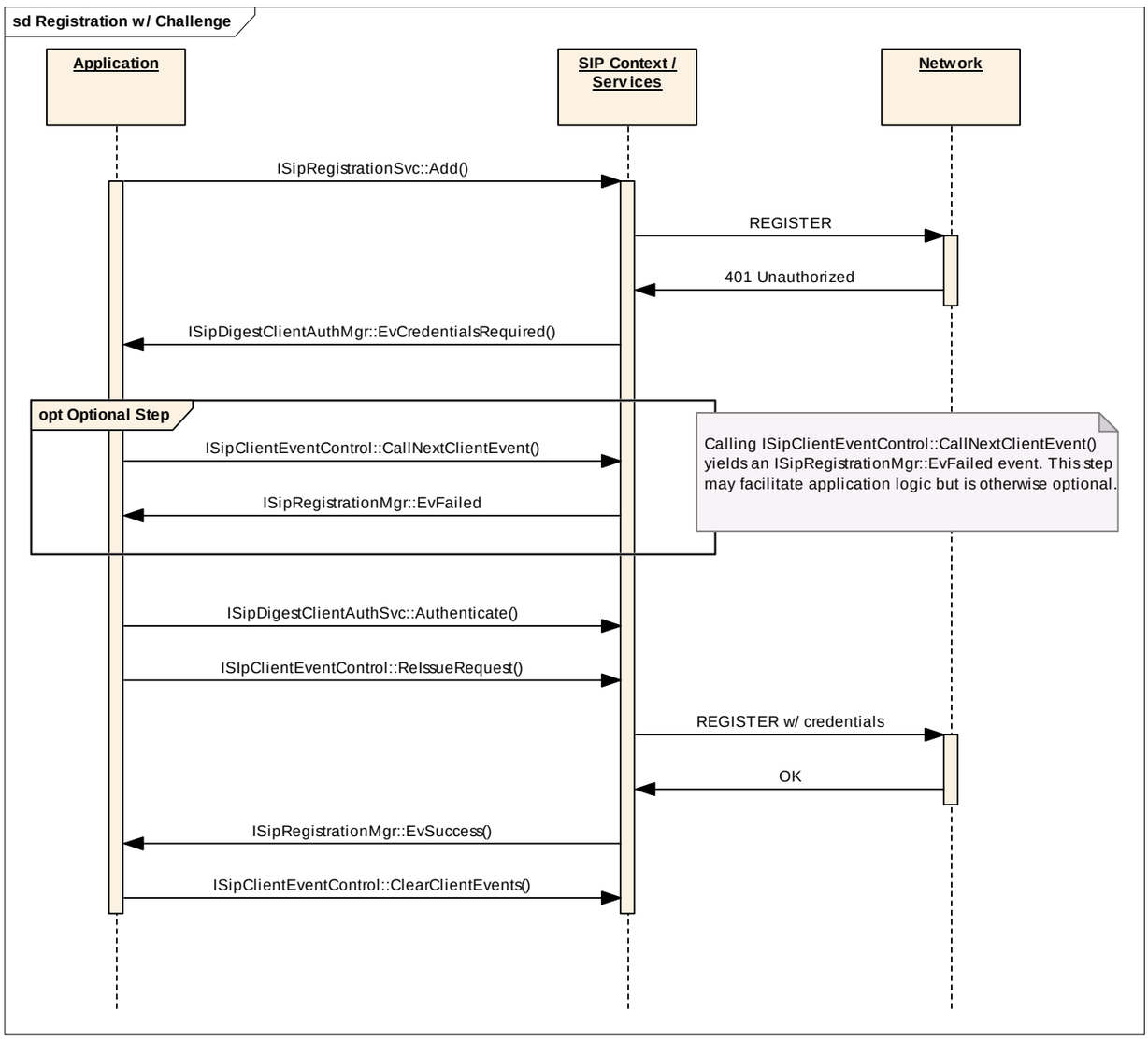


Figure 22 : Registration with a Challenge

4.5 Call Establishment and Teardown

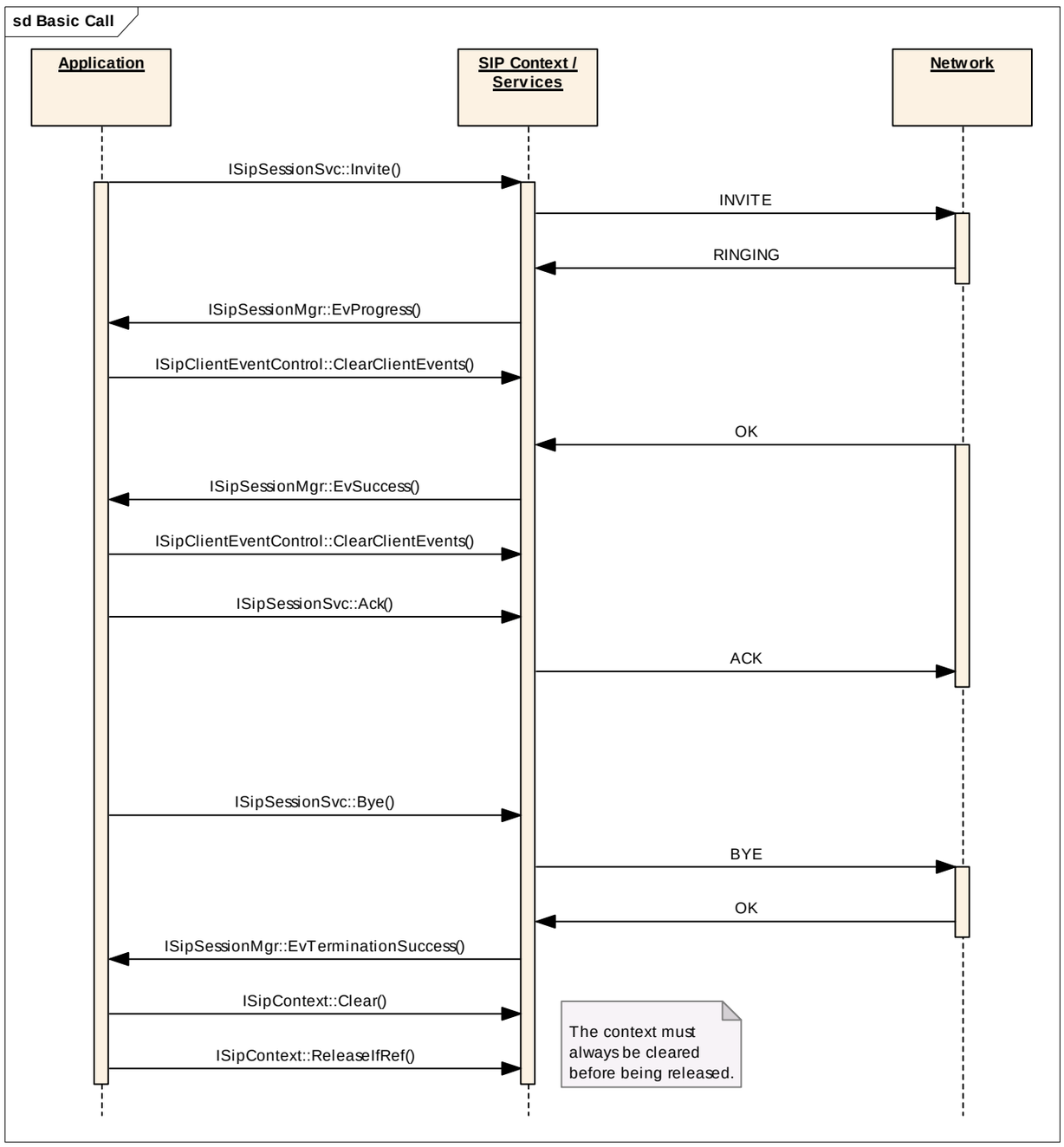


Figure 23 : Call Establishment and Teardown

4.6 Receive a Call and Teardown

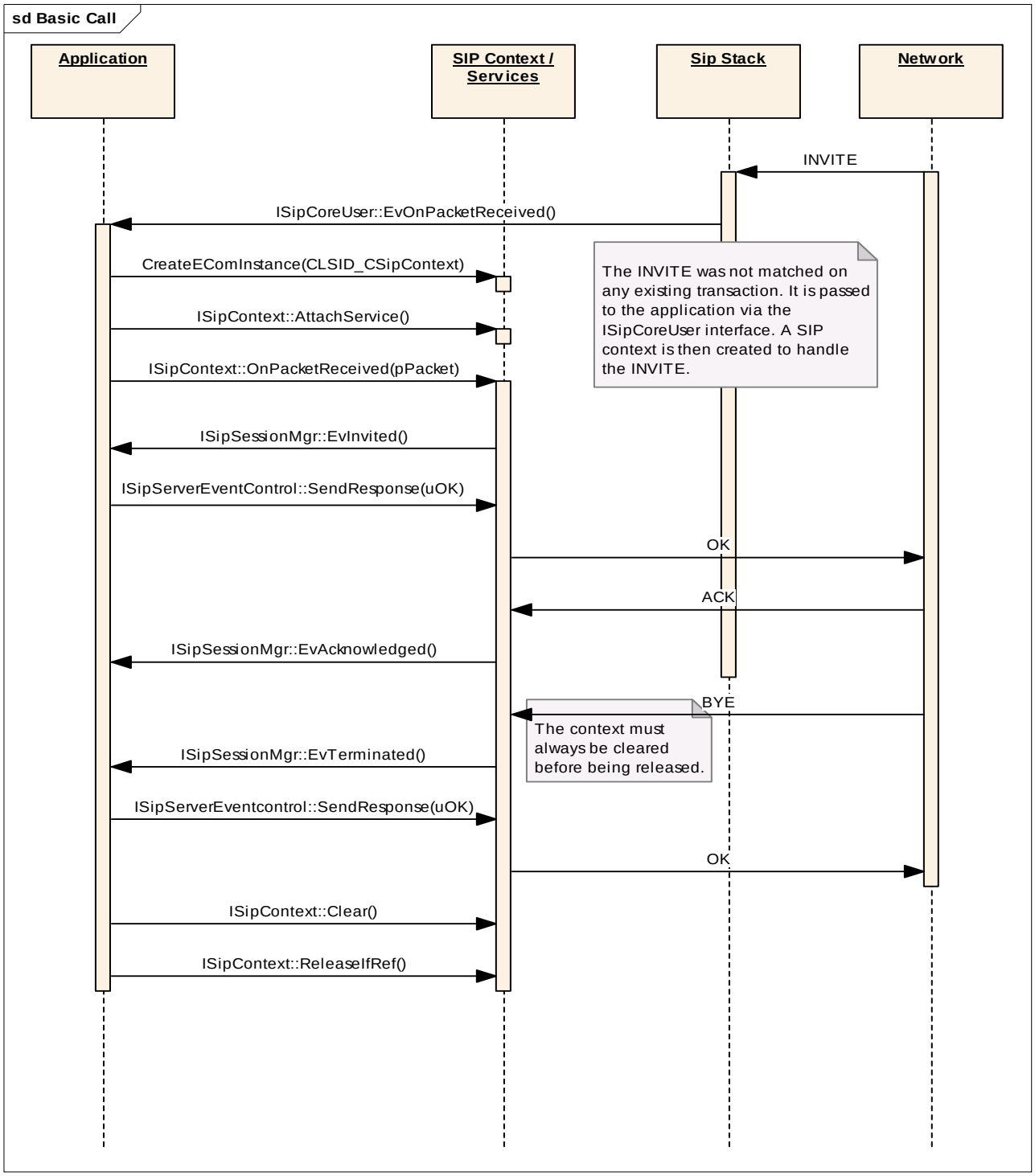


Figure 24 : Receive a Call and Teardown

4.7 Call Merge

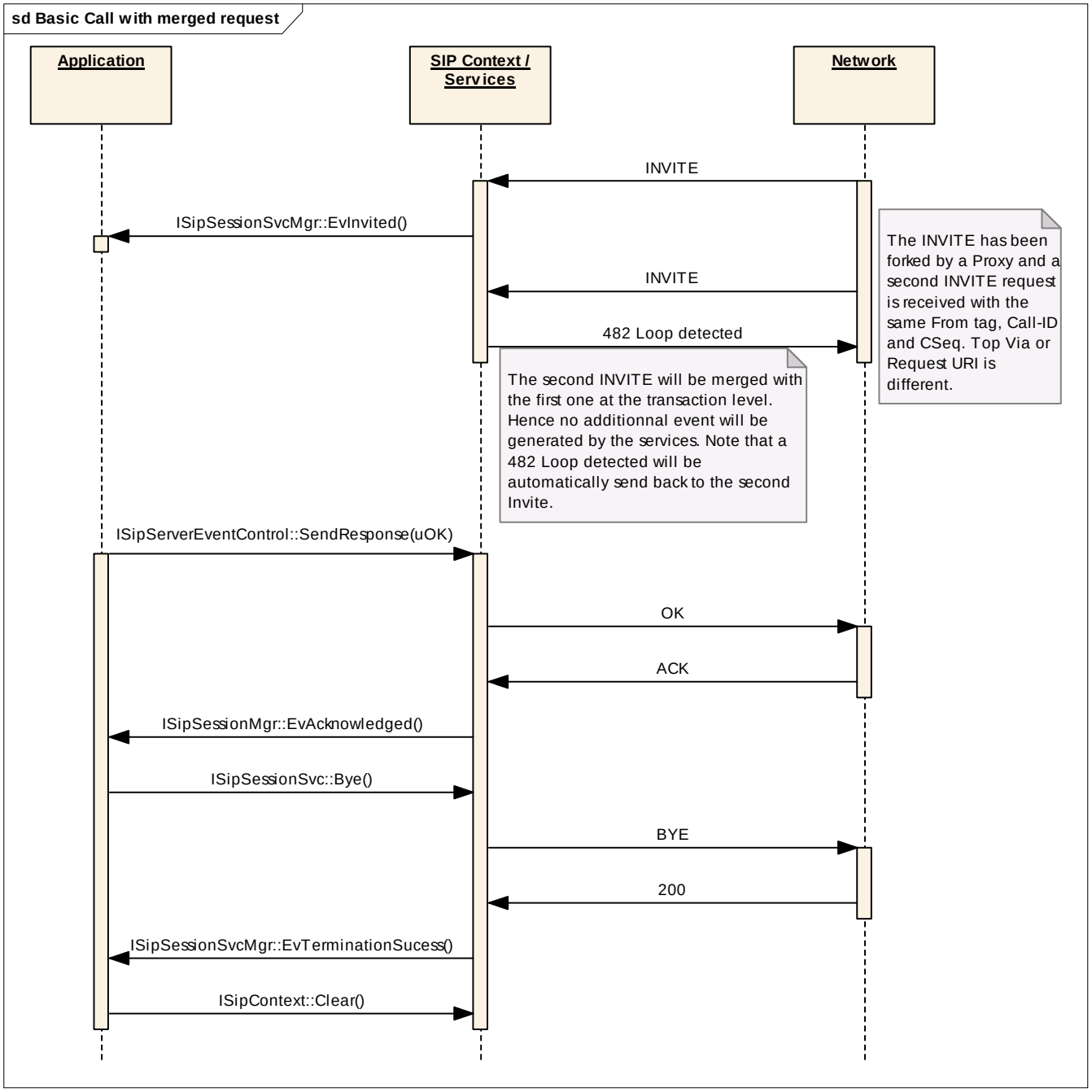


Figure 25 : Call Merge

4.8 Receiving and Forwarding a Call

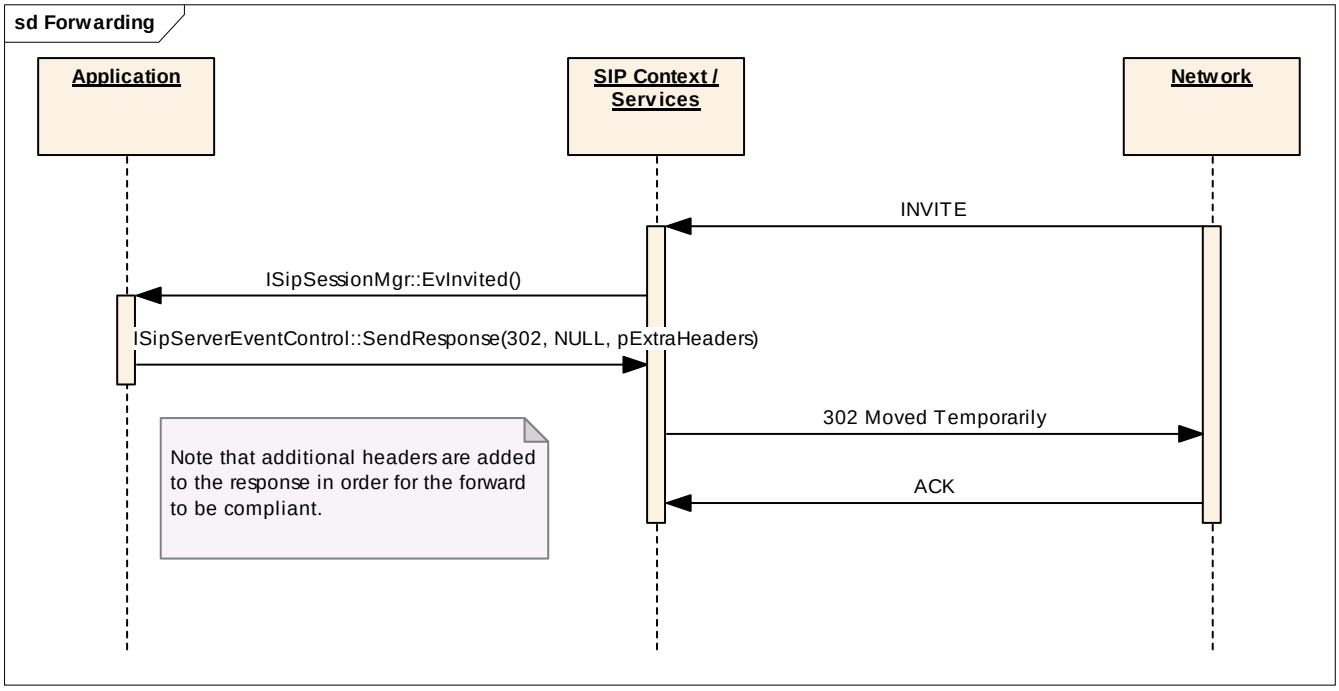


Figure 26 : Receiving and Forwarding a Call

4.9 Being Forwarded

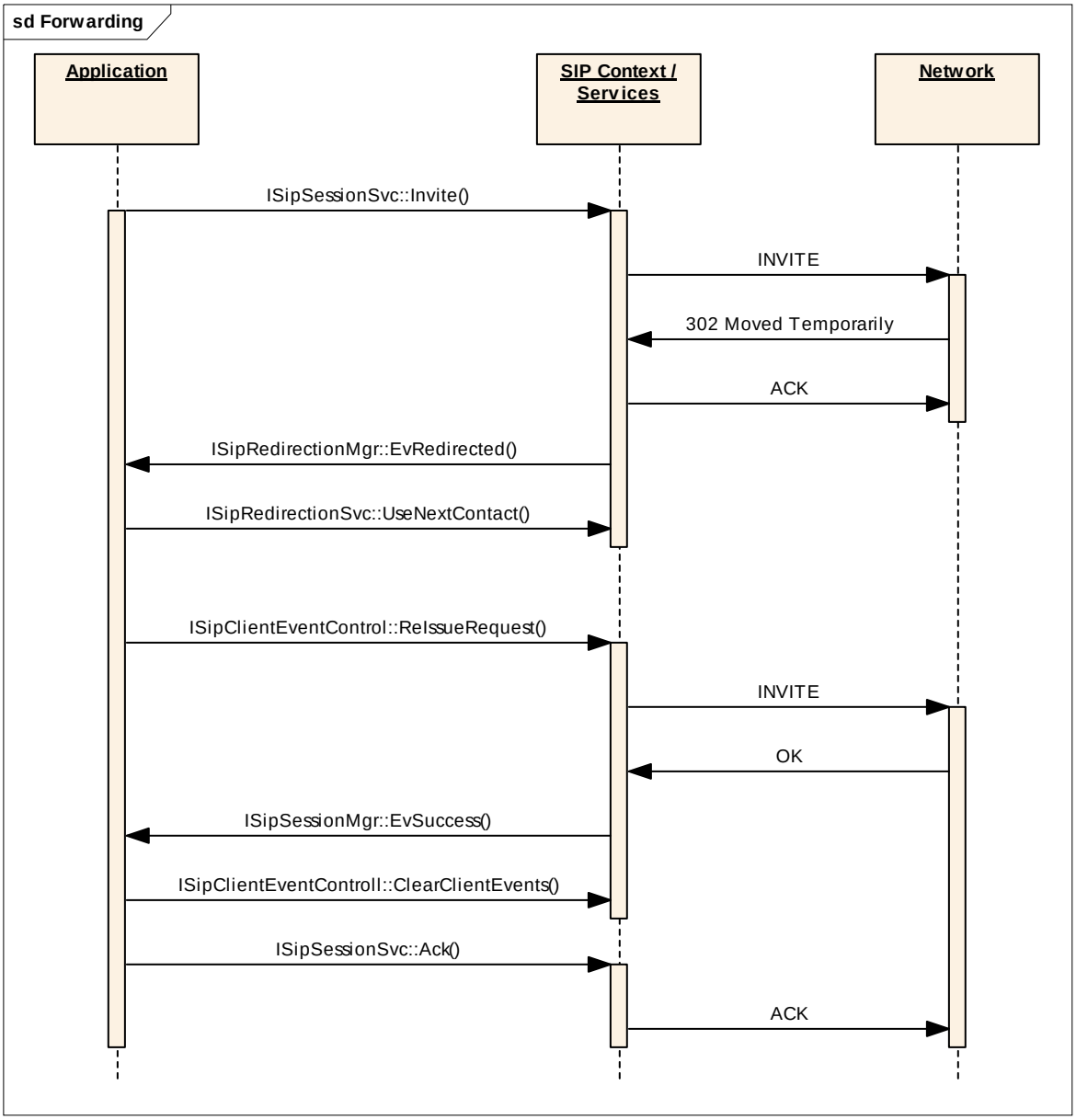


Figure 27 : Being Forwarded

4.10 Subscribing

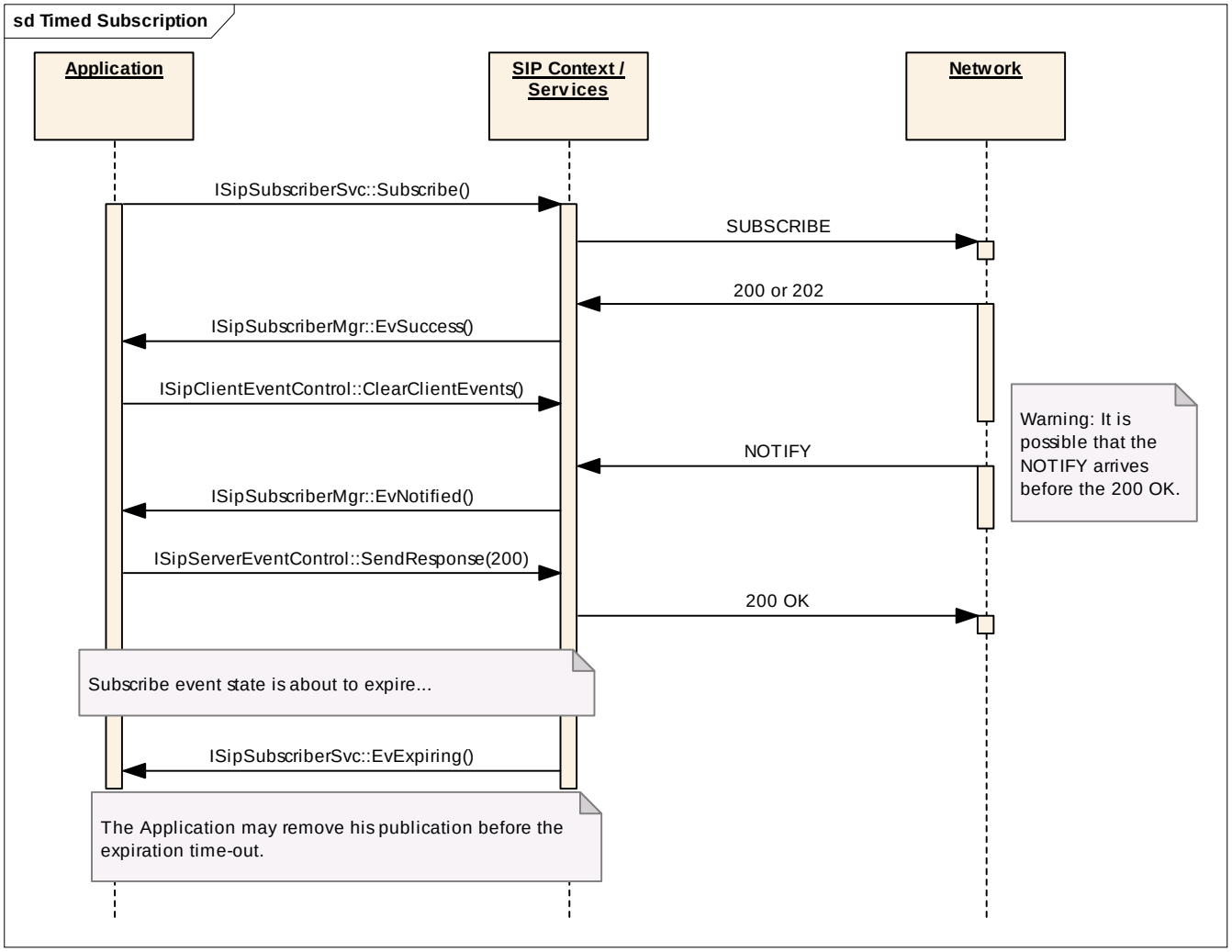


Figure 28 : Subscribing

4.11 Publishing State with Publish

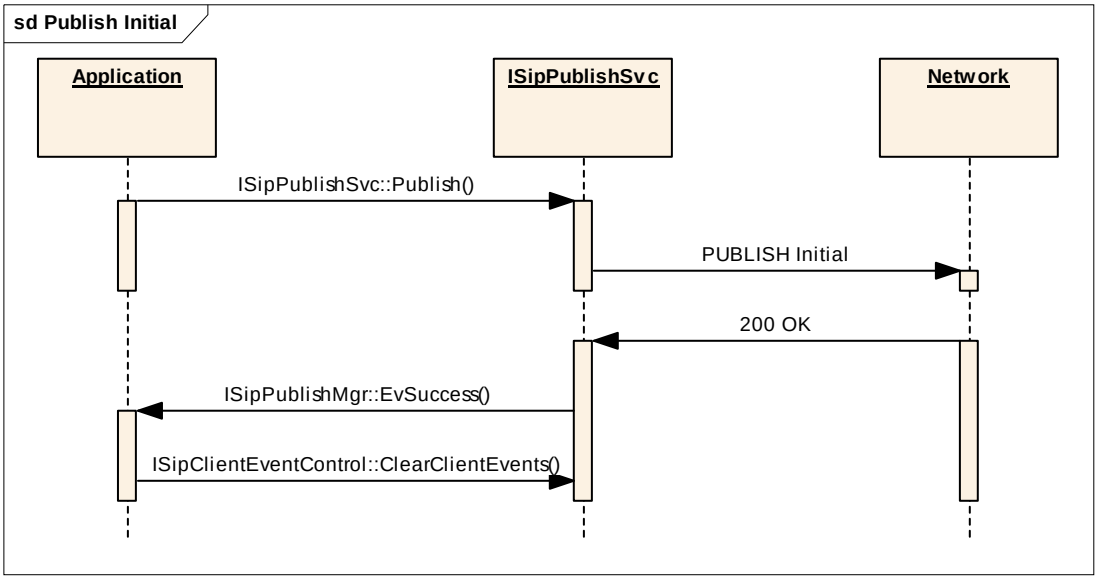


Figure 29 : Publishing State with Publish

5. Transport

Questions are often raised with regards to how M5T SIP SAFE handles various aspects of transport:

- How is the destination server selected?
- How is the transport protocol chosen for various SIP headers (Via, Contact, Route and Record-Route)?
- What are the timeout mechanisms related to transport?

The following sub-sections provide more details on these topics.

5.1 Preliminary Considerations

5.1.1 Routing Table

Section 3.1.3: Routing Table Configuration describes the routing table in some details. This routing table is obviously very important for some transport-related algorithms.

5.1.2 Server Location Service

The server location service is a normal service that must be attached to a SIP Context as explained in section 2.3.5.2: Attaching Core Services to a SIP Context. Hence, each SIP Context has its own instance of the service.

However, contrary to most other services, this service is mandatory because it performs required transport-related tasks for the SIP Context. These tasks are detailed below and include:

- Destination selection
- Via header handling
- Route header handling

Typically, this service should be attached to the SIP Context last but applications are allowed to break this rule. Indeed, it is conceivable that an application would want to attach other services after the server location service (notably the output controlling service, to inspect the outgoing packet).

5.1.3 UDP Maximum Size Threshold

RFC 3261 section 18.1.1 specifies that a request within 200 bytes of the path MTU, or larger than 1300 bytes if the path MTU is unknown, MUST be sent using a congestion controlled transport protocol, such as TCP.

In accordance with this specification, M5T SIP SAFE has a configurable UDP maximum size threshold, above which packets are first tried using TCP, if TCP is an available protocol. This parameter defaults to 1300 bytes.

5.2 Destination Selection

Selection of the destination server is accomplished by the aforementioned server location service. An in-depth treatment of this topic is outside the scope of this document but a brief explanation is given below about how this is accomplished.

5.2.1 Supported Transport Protocols

By default, M5T SIP SAFE supports UDP, TCP and TLS over TCP. However, this is configurable by the application, which can remove transports from this list.

5.2.2 Handling Responses

If the outgoing packet is a response, the destination address, port and transport are already known: from the Via header or from the connection being already established in case of TCP, etc. In case of send failures, processing is similar to that of requests, as discussed below. However, for the sake of brevity, responses are not specifically discussed further.

5.2.3 Building the Destination Hosts List

The server location service contains an ordered list of hosts. When first asked to process an outgoing packet, the hosts list is empty and must be built. The service thus:

- Determines the URI of the next hop, as per RFC 3261. When the packet is a request:
  - If the Request-URI is not empty, uses it by default.
  - If there is a Route header, uses the top Route. If the router is strict, performs necessary backward-compatibility processing.
- Resolves the next hop URI, in accordance with RFC 3263.
  - The *target* is:
    - The value of the *maddr* parameter, if present.
    - Otherwise, it's the host value of the hostport component of the URI.
  - A list of transport protocols is built.
    - If the *transport* parameter is present in the URI, this sole transport is used.
    - Otherwise, if the target is not a numeric IP address and no port is explicitly specified, a NAPTR request is performed.

Unsupported transports are removed from the list.

If NAPTR fails, SRV requests are tried directly with supported transports.

    - If the transport list is still empty, supported protocols are added.
  - A hosts list of IP addresses, ports and transports is built that maps to *target*, using:
    - The previously built transport list.
    - NAPTR, SRV and A/AAAA requests.
- If the packet size is larger than the UDP maximum size threshold:
  - The hosts list is reordered so that all UDP destinations are last.

The result of the resolve operation is the aforementioned list of hosts. This list contains everything necessary to send requests to the multiple remote hosts targeted by the next hop URI.

5.2.4 Sending Requests

Once the hosts list is built, the server location service is able to determine which hosts should be tried. Hosts are tried according to the following simple rules:

1. If the hosts list is empty: overall send failure, report to application.
2. Remove the first host from list.
3. Try the host.
4. On send failure: repeat from #1.

As can be seen, destination resolution is relatively straightforward and fully done in accordance with RFC 3263.

5.2.5 Connection BlackList Service

The Connection BlackList service automatically manages a blacklist of destinations to use. A blacklist is a list of destinations that are not used to contact a peer. A destination is an IP address, a port number, and a transport.

A packet is not sent to a blacklisted destination, unless all possible destinations to reach a peer are blacklisted. In this case, the preferred one is used.

When sending a packet to a destination fails, this service adds the destination to the blacklist for the configured amount of time. After the blacklisted destination expires, it is removed from the blacklist.

When sending a packet to a destination succeeds, this service removes the destination from the blacklist.

5.3 Transport Protocols

5.3.1 Via Headers

The handling of Via headers was briefly discussed in section 3.1.2: Via Address Preference. Let's now examine how Via headers are generated.

Via headers are managed by the server location service. When this service is about to send a request to a peer using a specified transport:

- The Via transport field is set to the same transport. For instance, if TLS is used to send the packet, the Via header contains "TLS".
- The Via Sent-By address is set with a local address that will be used to send the packet to the destination address. See section 3.1: Local Addresses for more information on local addresses and how they match remote addresses.

5.3.2 Route & Record-Route Headers

Route header processing with regards to transport has been discussed in section 5.2.3: Building the Destination Hosts List. Basically, when a Route header is present, the target is not the Request-URI but the top Route header.

Likewise, a UA receiving a packet containing Record-Route headers will preserve them. When subsequent requests are sent, this information is used to build Record headers. Processing is then as described above.

5.3.3 Contact Headers

The UserAgent service is responsible for setting the Contact in packets. According to the local interface that will be used to send the packet to the destination and the transport used, the UserAgent service will find the corresponding contact in the contact list that has been set in the service.

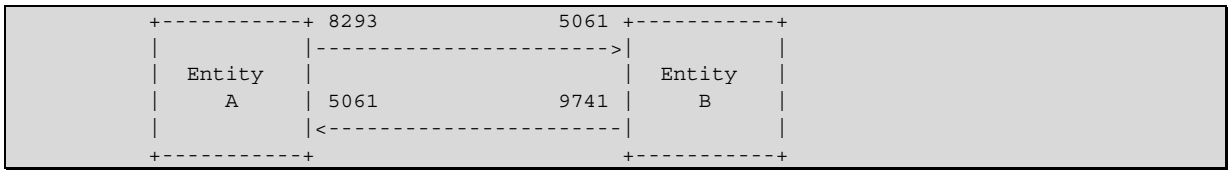
6. Security Considerations

This section explains how to use and setup the SIP UA SAFE in order to have a secure communication.

6.1 How TLS is Used in Most Deployments

6.1.1 The Client has its own Certificates

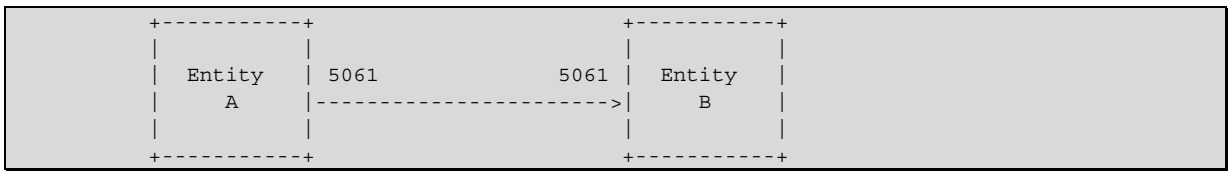
The SIP protocol includes mechanisms ensuring that responses to a request reuse the existing connection, which is typically still available, and also include provisions for reusing existing connections for other requests sent by the originator of the connection. However, new requests sent in the opposite direction (routed from the target of the original connection toward the originator of the original connection) are unlikely to reuse the existing connection. This frequently causes a pair of SIP entities to use one connection for requests sent in each direction, as shown below.



In TCP, this does not create any particular problem. However, in order for Entity B to connect back to Entity A, Entity A need to have a personal certificate for the TLS handshake.

6.1.2 Connection Reuse

If the Entity A does not have a personal certificate, the client has to trick the server to prevent it to initiate a new connection to send the response. The mechanism consists in creating a connection to the server from the local port that the client advertises as its listening port. By default, this is port 5061 for TLS. Thus, the client establishes a connection from port 5061 to port 5061 on the server.



As soon as the client establishes the connection to the server, it is required to send a request to the server, allowing the server to challenge the request and authenticate the user of the connection. This is usually done by sending a REGISTER request.

Note that since the client does not have a personal certificate, it is not possible to perform mutual TLS authentication. When the client establishes the connection, the client is able to verify that the certificate presented by the server is trusted. However, since the client presents no certificates to the server, the server relies on the challenge of the first incoming SIP request to authenticate the user that created this connection.

When the server stores the information pertaining to this connection, it stores it as a connection to the client's IP address, port 5061. Now, if the server wants to send a request to the client, it looks to see if it has an existing connection to port 5061 on this client. Since it did establish its connection from 5061, the client established connection is reused by the server. This effectively tricks the server into reusing the connection. There are however a few limitations and drawbacks to the use of persistent TLS connections.

The server could possibly manage the connection it establishes in a pool separated from the connection it has accepted. This would possibly break the connection reuse trick described above.

6.2 How to Use the Stack to Fit in these Deployments

6.2.1 The Client has its own Certificates

The appropriate certificates must be configured in the TlsContextFactory. See ISipTlsContextFactory in the API reference for more information on how to configure the default certificates.

6.2.2 Connection Reuse

Persistent connection must be used to connect to the wanted server. See CSipPersistentConnectionList and ISipPersistentConnectionSvc in the API reference for more information.

6.3 How to Properly Configure the Stack

- When sending a request to a SIPS Request-URI, a SIPS Contact must has been previously configured in the ISipUserAgentSvc too.
- When the first Route entry is a SIPS URI, a SIPS Contact must has been previously configured in the ISipUserAgentSvc too.
- When sending a request to a SIPS Request-URI, all pre-loaded Routes must be set to SIPS.
- When receiving a dialog-forming request with a SIPS Request-URI, a SIPS Contact must has been previously configured in the ISipUserAgentSvc too.
- When receiving a dialog-forming request with a SIPS URI in the top-most Record-Route, a SIPS Contact must has been previously configured in the ISipUserAgentSvc too.
- When receiving a dialog-forming request with a SIPS Contact and no Record-Route, a SIPS Contact must has been previously configured in the ISipUserAgentSvc too.

7. References

7.1 Internet-Drafts and RFCs

The following is a list of RFCs that are supported by M5T SIP SAFE and that users might find useful.

Main Normative References

- http://www.ietf.org/rfc/rfc2543 - SIP - Session Initiation Protocol
- http://www.ietf.org/rfc/rfc3261 - SIP - Session Initiation Protocol
- http://www.ietf.org/rfc/rfc3262 - Reliable 1xx Responses
- http://www.ietf.org/rfc/rfc3263 - Locating SIP Servers
- http://www.ietf.org/rfc/rfc3265 - SIP Specific Event Notification
- http://www.ietf.org/rfc/rfc2045 - Multipurpose Internet Mail Extensions (MIME) Part One
- http://www.ietf.org/rfc/rfc2046 - Multipurpose Internet Mail Extensions (MIME) Part Two
- http://www.ietf.org/rfc/rfc2047 - MIME (Multipurpose Internet Mail Extensions) Part Three
- http://www.ietf.org/rfc/rfc2617 - HTTP Auth - Basic and Digest
- http://www.ietf.org/rfc/rfc2806 - URLs for Telephone Calls
- http://www.ietf.org/rfc/rfc2976 - The SIP INFO Method
- http://www.ietf.org/rfc/rfc3311 - The SIP UPDATE Method
- http://www.ietf.org/rfc/rfc3323 - A Privacy Mechanism for SIP
- http://www.ietf.org/rfc/rfc3325 - Private Extensions to SIP for Asserted Identity within Trusted Networks
- http://www.ietf.org/rfc/rfc3326 - The Reason Header Field for SIP
- http://www.ietf.org/rfc/rfc3420 - Internet Media Type message-sipfrag
- http://www.ietf.org/rfc/rfc3515 - The SIP REFER Method
- http://www.ietf.org/rfc/rfc3665 - SIP Basic Call Flow Examples
- http://ww.ietf.com/rfc/rfc3761 - E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM)
- http://www.ietf.org/rfc/rfc3842 - Message Summary and MWI Event Package for SIP
- http://www.ietf.org/rfc/rfc3856 - A Presence Event Package for SIP
- http://www.ietf.org/rfc/rfc3891 - The SIP Replaces Header
- http://www.ietf.org/rfc/rfc3892 - The SIP Referred-By Mechanism
- http://www.ietf.org/rfc/rfc3903 - SIP Extension for Event State Publication
- http://www.ietf.org/rfc/rfc3966 - The TEL URI
- http://www.ietf.org/rfc/rfc4028 - SIP Session-Timers

7.2 WWW References

SIP Implementor's Mailing List

SIP users can use the SIP Implementor's mailing list to exchange about the protocol. This is the list to use to ask questions about the protocol and its implementation.

http://lists.cs.columbia.edu/mailman/listinfo/sip-implementors

SIP Working Group

The SIP working group is the IETF group responsible for the core development of the protocol.

WG Home Page: http://www.ietf.org/html.charters/sip-charter.html

Supplemental Home Page: http://www.softarmor.com/sipwg/

SIP Mailing List:

The SIP mailing list is also available for discussions on the current developments of the protocol. Do **NOT** use this mailing list to ask questions on how to use the protocol! The SIP Implementor's Mailing List must be used if you have questions concerning implementation of the protocol.

General Discussion: sip@ietf.org

To Subscribe: Send email to: sip-request@ietf.org with "subscribe" in body (without the quotes)

Archive: http://www.ietf.org/mail-archive/working-groups/sip/current/maillist.html

SIPPING Working Group

The SIPPING working group is the IETF group responsible for documenting the use of SIP for telephony and multimedia applications and for developing requirements and extensions to SIP needed for those applications.

WG Home Page: http://www.ietf.org/html.charters/sipping-charter.html

Supplemental Home Page: http://www.softarmor.com/sipping

SIPPING Mailing List:

General Discussion: sipping@ietf.org

To Subscribe: Send email to: sipping-request@ietf.org with "subscribe" in body (without the quotes)

Archive: http://www.ietf.org/mail-archive/working-groups/sipping/current/maillist.html

SIMPLE Working Group

This IETF working group develops SIP extensions for presence and instant-messaging.

WG Home Page: http://www.ietf.org/html.charters/simple-charter.html

Supplemental Home Page: http://www.softarmor.com/simple/

SIMPLE Mailing list:

General Discussion: simple@ietf.org

To Subscribe: Send email to: simple-request@ietf.org with "subscribe" in body (without the quotes)

Archive: http://www.ietf.org/mail-archive/working-groups/simple/current/maillist.html

Henning Shulzrinne SIP Page

Original SIP author: http://www.cs.columbia.edu/~hgs/sip/

8. Index

Programmer's Guide
M5T SIP SAFE – Version 4.1

Media5 Corporation