



amid.ukr software

Virtual Network Framework

27.09.2017

Dmytro Brazhnyk

LICENSE

This document can be distributed

Copyright 2014 Dmytro Brazhnyk <amid.ukr@gmail.com>

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

Executive Summary

This document is the technical specification for Neuron VNF(Virtual Network Framework). It contains goals, requirements and technical design for the framework.

The primary intention for this framework is to reduce complexity of Quantum Game Engine, by extracting the messaging layer into separate components that can be easily reused for any kind of software.

Table of Content

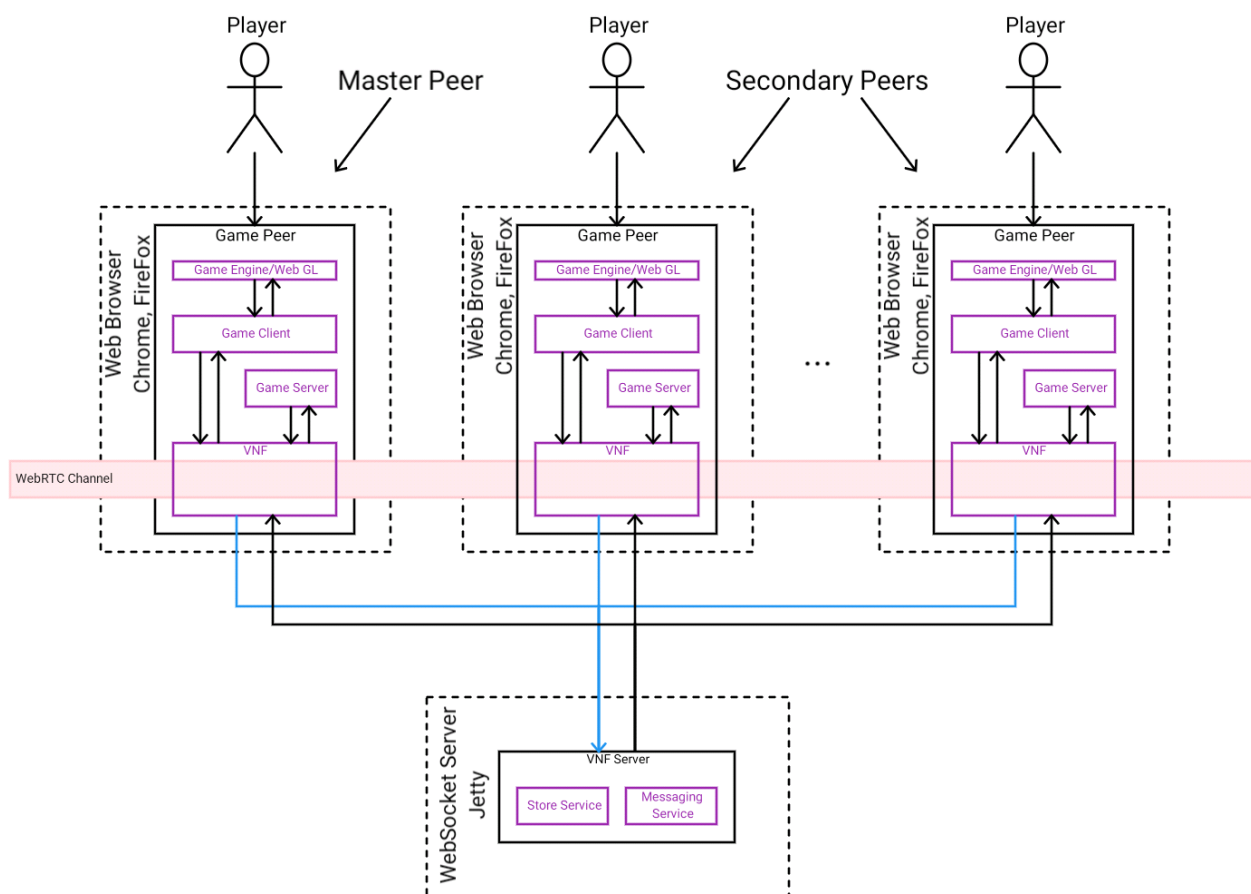
LICENSE	1
Executive Summary	2
Table of Content	3
Motivation	5
Goals	6
Objectives	6
Overview	6
Requirements	7
Channel Requirements	7
Registry Requirements	7
Technical Design Overview	9
Channel	11
Endpoint Interface Declaration	12
Endpoint Implementations	12
Reliable Hub	13
Requirements	13
Reliability Issues	13
Message Lost	14
Message Order Lost	15
Consistent Connection Establishing	15
Second Connection	16
Handshake Message Lost	16
Connection Lost	17
Buffer Overflow Prevention	17
Reliable Connection Sequence Diagrams	18
Reliable Concurrent Connection Sequence Diagram	19
Reliable Message Format	19
EVA Protection from Spoofing	21
Registry	21
WebSocket RPC	22

Reliability and Failover Scenarios	22
Message resend condition	22
Reconnection condition	23
Detecting Connection Lost	23
WebSocket Message Send Flow	23
WebSocket RPC Messages Format	24
WebSocket RPC Methods	25
VNF WebSocket Server	26
Command Processor Engine	26
Business Services Layer	27
Technology Stack	27
Browser Side	27
Server Side	28

Motivation

Build of this framework started as WebRTC messaging component of Quantum Game Engine. However, to reduce complexity of Game Engine, a decision was made to extract it as a separate framework, with an extensible layered approach to handle different aspects of messaging requirements.

Also this framework should provide possibility for peers to discover each other, so peers can register and do lookup. Prototype for discovery functionality is registering game matches in the game lobby.



Goals

1. Reduce cost and requirements for Game Server Hosting Hardware - build networking framework for distributed computing that is run on the Web Browser side.

Objectives

1. Build Framework for real-time peer-2-peer communication between Web Browsers.
2. Web Server should be only used as a signaling channel to do NAT traversal and initiate WebRTC communication.
3. Web Server should provide an In-Memory registry, for peer registry/lookup workflows.
4. Network channels should behave reliably, with message and message order lost recovery mechanisms.

Overview

Neuron VNF(Virtual Network Frameworks) is framework for reliable real-time peer-2-peer communication for applications executed in Web Browser. VNF is designed as a generic tool that can be easily integrated into any kind of application, however original intention was to build a framework that would be used as part of multiplayer p2p Game Engine, so main focus on performance and latency.

Requirements

VNF is split into two modules: channel and registry.

Channel Requirements

1. Central communication protocol for VNF is WebRTC, since it provides direct communication between Web Browsers.
2. WebRTC uses sophisticated NAT traversal mechanisms, and utilizes UDP protocol as well as TCP, so during tests on real application, reliability issues like connection loss was detected. Requirement for VNF to cover this gap and provide a solution for reliability issues. More details in specification for reliability in this document.
3. For performance reasons all capabilities of WebRTC channels should be utilized, so by default WebRTC channels should be used as reliable, and only in case of detection of delivery failure, recovery mechanisms should be executed, like Negative Acknowledgement, Heartbeats, Message Resend etc...
4. WebRTC has limitations on message size sent, VNF should support any.
5. In case WebRTC is not available, messages can be relayed through Web Socket server
6. Endpoints should be protected from Spoofing
7. Customizable layered design with different capabilities to reuse.
8. Single API for different channel implementations, like: WebSocket, WebRTC etc.

Registry Requirements

1. Peers can put entries with content into the registry.
2. Registry's primary purpose is discovery mechanism for service register and lookup functionality, so if peer lost WebSocket connection to server, entry automatically will be removed.
3. Entries are grouped into collections, so peers can lookup for all entries registered in a particular collection.
4. Each entry belongs to only one collection, Many-to-One relationship.
5. Collection is identified by name, while entry by entry name and collection name
6. Same WebSocket connection should be used for Registry API and message relay.



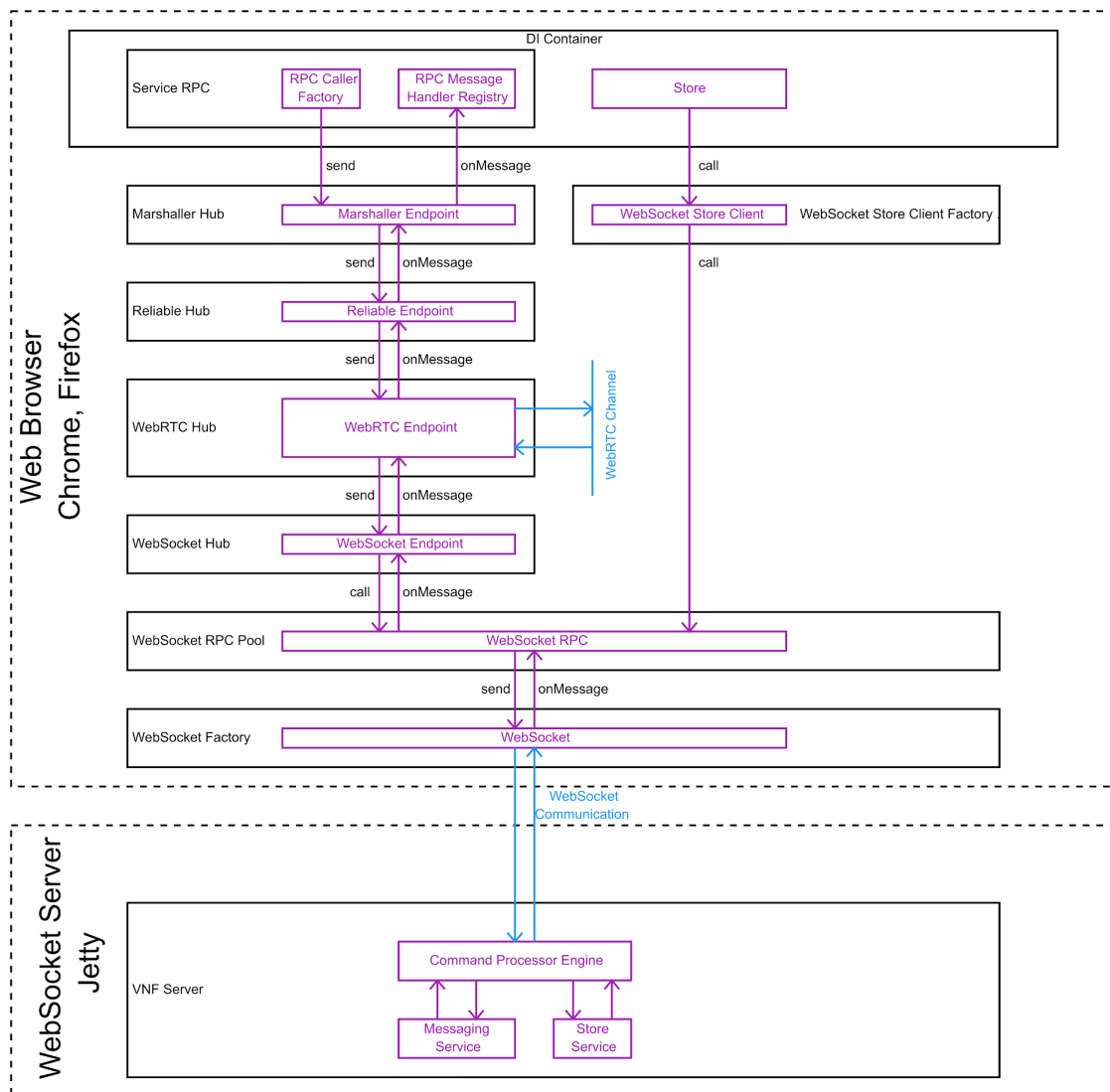
Technical Design Overview


VNF Framework presented by two modules:

- Channel
- Registry

And consists of two tiers:

- Web Browser
- WebSocket Server





Integration between Web Browser and WebSocket Server made with WebSocket connection.

Since p2p design implies to have Server and Client side in Web Browser, and Web Server is mostly used as a message broker, **Web Browser tier** will be used as a special term that is run on User Agent side.

On the Web **Browser** side VNF is implemented with three major components: **WebSocket RPC**, **Channel** and **Registry**.

WebSocket RPC is an RPC to send requests to WebSocket Server over WebSocket connection.

Channel component uses WebSocket RPC to send messages to other Web Browsers, and listen for messages from another Browser.

Registry uses WebSocket RPC to store and retrieve registry entries on WebSocket Server.

WebSocket Server is implemented as a set of services that handles commands from WebSocket RPC.

Channel

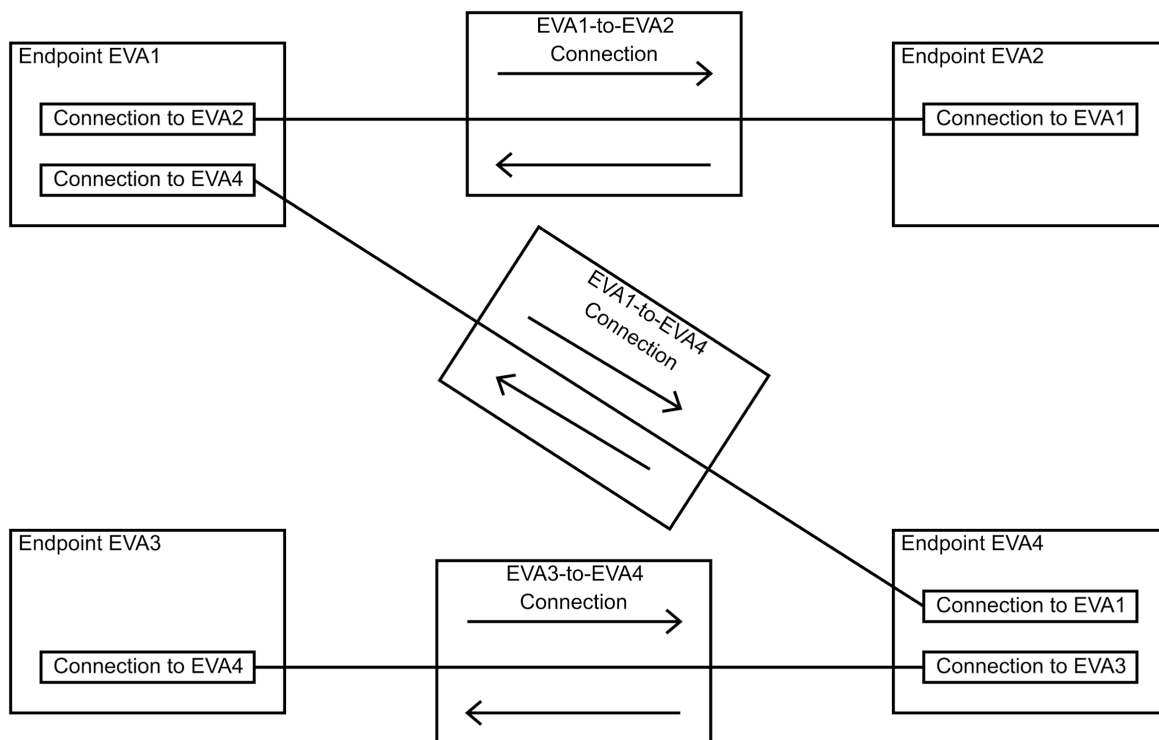
VNF network presented by **Endpoints** that can send and receive messages. Different Endpoints implementations have the same compatible interface, so client code will not see the difference.

Each Endpoint has a unique name **EVA**(Endpoint Virtual Address), that can be used to point the address of destinator, to whom a message should be sent.

Endpoints are grouped into hubs. **Hubs are** used as factories and containers, to open new endpoints in the same EVA address space. Two Endpoints opened in the same Hub, are available to each other. Two different Web Browsers can use WebSocket Hub created for the same WebSocket Server, so new Endpoints will be in the same EVA address space, and these endpoints can send messages to each other.

To send a message Endpoint should open **Connection** first. One Endpoint can have multiple connections to multiple Endpoints, to send and receive messages from multiple Endpoints.

Special **Proxy Endpoints** can decorate other Endpoints with extra functionality like reliable delivery extension, message marshalling etc.



Send message workflow looks follow:

1. Create or get Channel Hub
2. Generate EVA address for new Endpoint
3. Open Endpoint using generated EVA address
4. Get EVA address of another
5. Open connection to another Endpoint using EVA address
6. Wait, and send a message once the connection will be opened.
7. Receive messages and Send message to complete business operations
8. Close connection if necessary
9. Destroy Endpoint if necessary

Endpoint Interface Declaration

Hub:

- openEndpoint(eva)

Endpoint:

- openConnection(eva, callbacks)
- isConnected(eva)
- send(eva, message)
- Callback: onMessage(event)
- closeConnection(eva)
- Callback: onConnectionLost(eva)
- destroy()

Endpoint Implementations

Endpoints Hubs can be split into two categories:

- Endpoints Hub that are used to create EVA address space and deliver messages to designators by EVA address. Example of such endpoints WebSocket Hub, InBrowser Hub, WebRTC Hub
- Endpoints Hub that is used to proxy underlying hubs to extend communication with extra functionalities. Example: Reliable Hub, Marshaller Hub, RTC Hub.

Browser Hub - simplest implementation channel hub, mostly used for unit testing, built as Proof of Endpoint-Hub Concept. Browser Hub is ready to use Hub, to open new Endpoints, that do not require extra channels for communications, and uses direct js method calls. Each instance of In Browser Hub represents separate EVA address space.

WebSocket Hub - is a hub that uses WebSocket Server to send messages to other Endpoints, WebSocket Hub requires WebSocket RPC Pool, to instantiate WebSocket RPC for each endpoint.

RTC Hub - is a proxy and message delivery hub at once. RTC Hub uses WebRTC to deliver messages, however to establish RTC connection signaling channel is required, and for this reason RTC Hub requires it to work on top of another hub. In Browser Hub or WebSocket Hub can be used as parent or signaling channel. Opening RTC Endpoint it also will open a new Endpoint signaling channel hub.

Reliable Hub - is a proxy hub that decorates parent hubs with reliable delivery functionality. Can be used on top of any other hub like RTC, WebSocket or In Browser.

Marshaller Hub - is proxy hub for two purposes: 1) split big messages and send with smaller fragments via parent hub. 2) serialize JS objects to json - for performance reasons only string messages are supported by other hubs.

Reliable Hub

Requirements

1. Inherit Endpoint interface
2. Establish connection with other Reliable endpoints
3. Send message to other Reliable endpoint
4. Retrieve message from other Reliable endpoint.
5. Detects connection lost to other connected Reliable endpoints.
6. Use the underlying parent endpoint to send messages to other Reliable endpoints.
7. Reliable endpoint behaves conceptually correctly, even when parent endpoint behaviour is unexpected.
8. Reliable should handle any reliability issues of the underlying parent endpoint.

Reliability Issues

1. Parent endpoint can lose messages - lost message recovery required.
2. Parent endpoint can lose connection - connection recovery required.
3. Parent endpoint can lose connection without notification - detection mechanism required.
4. Remote endpoints can be destroyed without any notification from the parent endpoint - connection loss detection required.
5. Send/Receive over buffering protection required
6. Messages from previous/dropped connections can be retrieved - extra validation step is required.

7. Connection lost events should be consistent on both ends of connection.

Message Lost

Reliable Endpoint should guarantee delivery of all messages. Message lost can be divided into three categories:

- FML - First message Lost
 - MML - Message in Middle Lost
 - LML - Last message Lost
1. Each sent message should have a message index, so the recipient can use it to detect gaps and restore message ordering.
 2. Outgoing queue is required to store sent messages, and be used to resend in case of message loss is detected.
 3. Message indexing counting starts from zero for each new connection between endpoints.
 4. First message is defined by the moment of connection establishing, and has message index zero.
 5. For performance reasons, to not overcomplicate delivery, a reliable delivery mechanism of the parent endpoint should be used by default, however the recipient should verify that there is no gap in sequence of message index, otherwise **NACK**(negative acknowledgement) should be sent to re-request lost messages.
 6. For performance reasons, to reduce network redundancy, NACK is sent as part of Heartbeats, to give a chance to succeed with delivery of messages, so redundant message copy will not be sent.
 7. FML and MML have the same detection mechanism, and can be easily detected by the message index of the last retrieved message.
 8. LML uses a different approach to FML and MML, since the recipient does not have any notification that at least some message sent, **heartbeats** are required to detect LML.
 9. Heartbeat with LML NACK is sent every time even if no loss happened, just to check if the sender has any undelivered message in the send queue.
 10. For performance reasons at least two LML NACKs are required to re-send a copy of a message, because LML NACK can be sent concurrently to last message sending, and LML NACK be received concurrently to successfully delivery of last message - so no real message lost really happened.

Message Order Lost

Message Order should be guaranteed by a Reliable endpoint.

1. Incoming queue is required to store messages ordered by message index.
2. In case of any gap detected in the incoming queue, Message Lost fallbacks are required to fill that gap.
3. Reliable Endpoint client code should get messages from incoming queue only ordered by message index
4. In case of any gap, last messages after the gap should not be visible to client code.

Consistent Connection Establishing

Both endpoints should establish connection, only if both endpoints consistently agreed on that.

1. Extra handshaking message types are required to synchronize connection status on both Endpoints.
 - a. Handshake Message
 - b. Accept Message
 - c. Connection Acknowledgement Message
2. Endpoint that initiating handshake should get **Accept** message to be sure, that:
 - a. Channel available to deliver message in both directions
 - b. Endpoint on other side knows that new connection is being established
 - c. Connection session resource is allocated exactly in correspondence to session resource of initiator, so no other can interrupt this process.
 - d. Start heart beating and changing status to connected
3. Endpoint that accepting connection should get acknowledgement from initiator to be sure, that:
 - a. An Acceptance message is delivered to the initiator, and the channel is available in both directions.
 - b. Endpoint on other side changed status to connected, and connection is established
 - c. Initiator connection session resource is allocated exactly in correspondence to acceptor connection session resource.
 - d. Start heart beating and change status to connected.
4. Heartbeat message can be used as Connection Acknowledgement Message, because Heartbeat message can be sent only if Accept Message retrieve and endpoint status changed to connected.
5. Connection should be consistently established even in case of intensive concurrency, when both endpoints act as connection initiators.

6. If the user intended to call `openConnection`, and endpoint acting as Acceptor, however no responses to `ACCEPT` messages were retrieved, the acceptor should fallback to connection Initiator scenario, by sending a Handshake message, before notifying that connection failed.

Second Connection

There are exist multiple cases when connection can re-created to the endpoint with same EVA, for example use code could close connection between Reliable Endpoint, and then open connection again, or even one of endpoint can destroyed and recreated with the same EVA, In all this case Reliable endpoint should behave correctly, and **Phantom** messages should not affect current connection.

1. Each endpoint should generate a unique **Session ID**, so one connection will have two Session ID on both ends. Session ID will be used:
 - a. By Sender to specify whom message is addressed
 - b. By Recipient to validate if a message is addressed to the current active connection.
2. Each time connection drops a new unique Session ID, should be generated, for effective Recipient message validation.
3. Any message sent to Session ID that does not correspond to the current active session, should be invalidated as **Phantom** Message.
4. Connection initiator sends its Session ID to the acceptor so the acceptor will know which Session ID to use as the target Session ID for Accept Message.
5. Accept Message will also contain Session ID of acceptor as well as Handshake Message contains Session ID of initiator.

Handshake Message Lost

Handshake messages can be lost by the parent endpoint as well as any other messages. Resend mechanisms are required, to support Reliable connection establishment.

- 1) Handshake, Accept and Heartbeats are sent periodically with some Interval until connection state is changed.
- 2) After some amount of Handshake resend parent connection should be invalidated - closed and opened again, to be sure that parent connection is not an issue. Timeout intervals should be configured.
- 3) Even if the endpoint behaves as an acceptor, the user still should be able to get a successful notification that the connection opened.
- 4) Another timeout interval should be used, to detect that connection establishing failed.

Connection Lost

Parent endpoint can lose messages and connection, however the Reliable endpoint should strictly classify when connection to another Reliable endpoint is lost, or just the channel of the parent endpoint behaves unreliable, and a recovery mechanism should be applied.

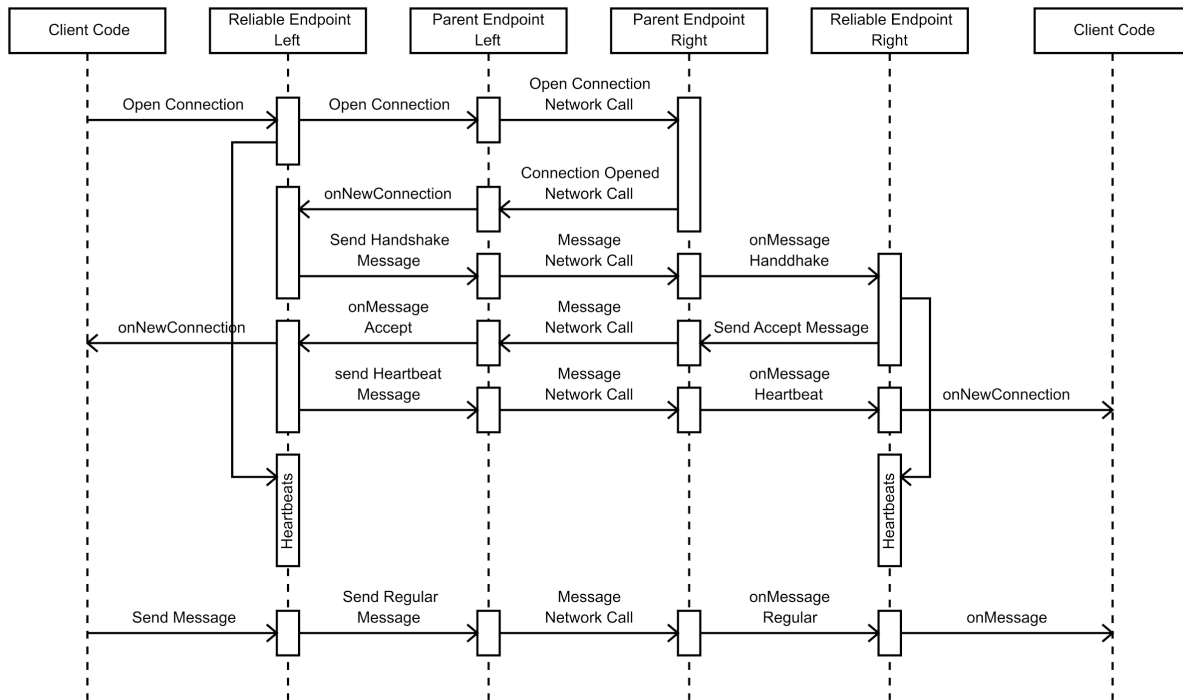
Connection lost events on both ends should be consistent.

1. Reliable approach to detect Connection Lost between Reliable Endpoint is silence - no messages, no heartbeats - for example somebody just cut wire.
2. Special case of Connection Lost is the remote endpoint destroyed.
3. If client code decides to close the connection, a Close Connection message should be sent, for graceful connection close on both ends.
4. Before connection closed detected by silence, connection of parent endpoint should be invalidated, by closing and opening, multiple time configured by connection invalidate timeout interval.
5. With use of Phantom detection, connection will be consistently closed on the other side due to silence, even if graceful close connection messages were not delivered.

Buffer Overflow Prevention

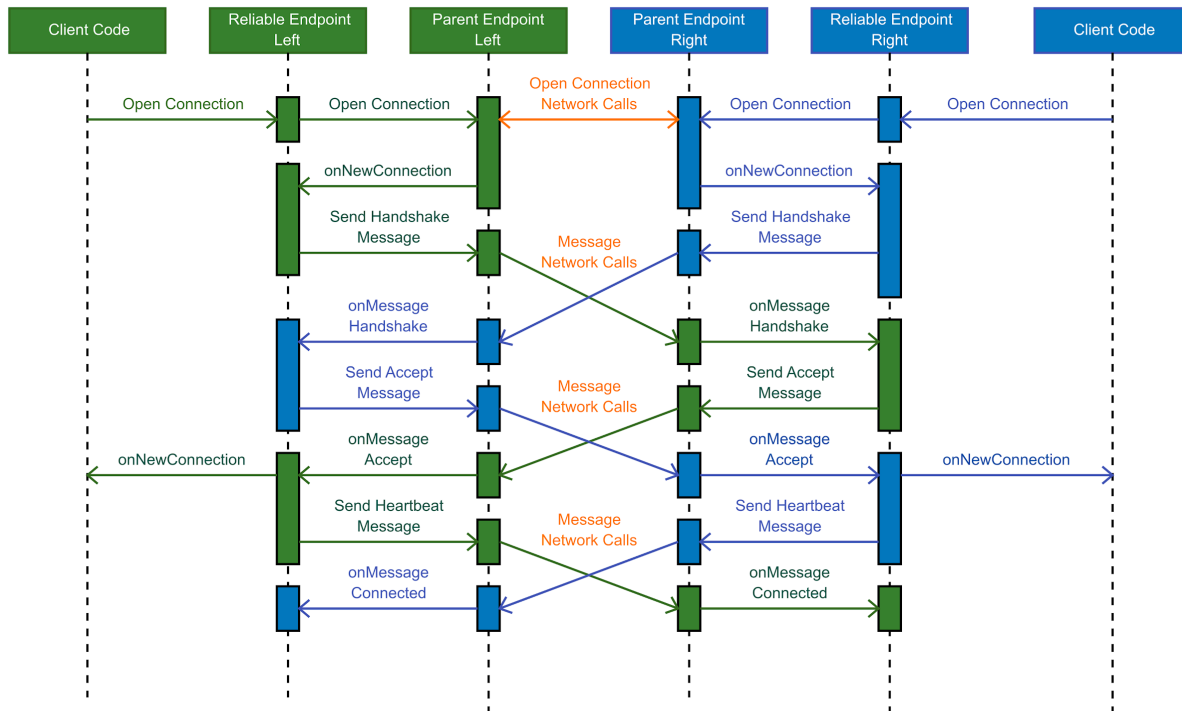
1. Incoming buffers can remove messages that were delivered to client code. In most cases in case no message is lost, the incoming buffer should be empty.
2. Outgoing buffer should hold a message, until heartbeat retrieves, which specifies from the last message index that was delivered, all messages before that index can be removed.

Reliable Connection Sequence Diagrams

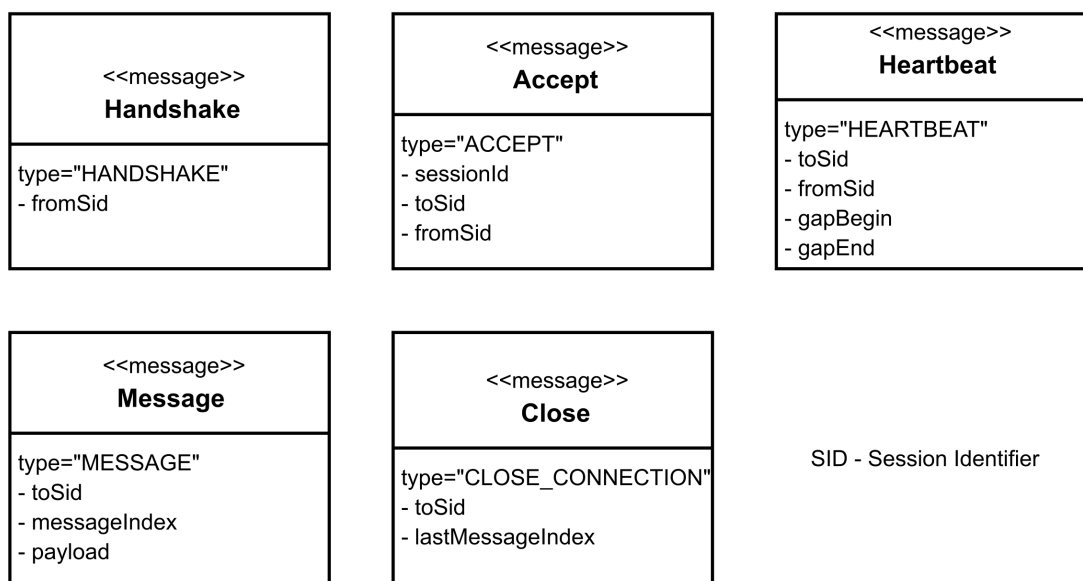


- Four connection states exists caller: HANDSHAKING, CONNECTED
 - callee: ACCEPTED, CONNECTED
 .
 Flow:
 caller: openConection
 • HANDSHAKING state
 • HANDSHAKE message sent
 callee: HANDSHAKE message got
 • ACCEPTED state
 • ACCEPT message sent
 caller: ACCEPT message got
 • CONNECTED state
 • HEARTBEAT message sent
 callee: HEARTBEAT message got
 • CONNECTED state

Reliable Concurrent Connection Sequence Diagram



Reliable Message Format



Phantom Detection Table

Table below describes required checks for Phantoms messages.

Rows represent messages retrieved.

Column represents the state of the endpoint.

If **Phantom is detected** then processing of that message should be aborted.

Normal Behaviour - means that this message type is expected by this endpoint state by default, however Session Id still should be checked. Other conditions are also possible, due to the reliable nature of network connections.

Retry Message - message send second time

Concurrent Connection - case when connection initiated on both endpoints concurrently.

HEARTBEAT missed - case when, connection is established on the other side, and dropped then, but HEARTBEAT missed to be delivered before CLOSE-CONNECTION.

	NO-ACTION	HANDSHAKING	ACCEPTING	CONNECTED
HANDSHAKE	Normal Behaviour: no checks	Concurrent Connection: no checks	Retry Message, verify: - fromSid	Phantom
ACCEPT	Phantom	Normal Behaviour, verify: - toSid	Concurrent Connection, verify: - toSid - fromSid	Retry Message, verify: - toSid - fromSid
HEARTBEAT	Phantom	Phantom	Normal Behaviour, verify: - toSid - fromSid	Normal Behaviour, verify: - toSid - fromSid
MESSAGE	Phantom	Phantom	Phantom	Normal Behaviour, verify: - toSid
CLOSE-CONNECTION	Phantom	Phantom	HEARTBEAT missed, verify: - toSid - fromSid	Normal Behaviour, verify: - toSid - fromSid

EVA Protection from Spoofing

EVA is property of Endpoint, it is reasonable on Web Browser side generate EVA code, because WebSocket Server just another channel for communication, it can be rebooted, etc, so it is responsibility of Endpoint to make it available in EVA address for other Endpoints, however if decision made on Web Browser side what will be an EVA, this approach can be easily spoofed other endpoint can identify himself with stolen EVA.

For protection asynchronous encryption should be used, with private and public keys. Public key can be used as an EVA, so only the holder of the private key can prove to him that this Public key belongs to him.

Authorization could be made in two place:

- Web Server could authenticate EVA identity. In this case, all endpoints that communicate through WebSocket Hub or use it to establish RTC connection, can be sure that VNF WebSocket Server already made authentication.
- For more sophisticated configurations without central authorities like VNF WebSocket Server, an extra proxy hub SecureHub should be used, which will block openConnection, if EVA identity cannot be proven. In this case also protection against Man-in-the-Middle should be implemented, using secure public key exchange techniques.

Registry

Registry is tightly coupled with the channel framework, and also requires EVA for internal purposes. Registry API is presented by two interfaces:

Registry Client Factory - interface to instantiate new registry clients.

Registry Client - to access the registry by putting and retrieving entries from EVA address space.

Two basic implementation is supported:

In Browser Registry Client Factory - in browser registry, do not require any external connections. Instance of In Browser Registry Client Factory represents EVA entry space.

WebSocket Registry Client Factory is a factory to create registry clients to In Memory registries on the WebSocket Server side. To establish connection with WebSocket Server, the registry client requires a WebSocket RPC instance, so the WebSocket Registry Client Factory requires the WebSocket RPC Pool.

WebSocket RPC

WebSocket RPC is a Web Browser component to communicate with WebSocket Server. It provide interface and supports 2 communication pattern:

- Request/Response with WebSocket Server
- Push notification from WebSocket Server to WebSocket RPC

WebSocket RPC consists of WebSocket RPC Pool, WebSocket RPC and WebSocket Factory.

WebSocket RPC component that represents WebSocket connection and provides interfaces for pattern above.

WebSocket RPC Pool is a factory and container for WebSocket RPC. WebSocket RPC designed in that way, so a single instance could be used by multiple consumers. Each instance of WebSocket RPC is identified by EVA. Role of WebSocket RPC Pool is to do reference counting for automatic allocation and destruction based on EVA as identification key.

WebSocket Factory - factory for WebSocket object, all WebSocket connections should point to the same WebSocket Server, so WS URL is mandatory property of WebSocket Factory.

Reliability and Failover Scenarios

WebSocket protocol is already considered as a reliable TCP based protocol with available servers, however during server restart connection loss can be detected.

1. System should be resilient for any exception from WebSocket.send.
2. In case of use WebSocket RPC for channeling Reliable Hub can take care about all requirements.
3. In the case of WebSocket RPC for Registry, during connection lost WebSocket Server will remove all entries, so it is the responsibility of WebSocket Registry Client to recreate entries, WebSocket RPC API should notify about re-establishing of WebSocket connection.

Message resend condition

1. TCP already guarantees gap detection, and lost messages can be only during connection loss.
2. During the WebSocket connection phase, all commands to WebSocket Server should be buffered and sent to the server once connection is established.
3. Due to failure on WebSocket Server, WebSocket RPC should stop resending messages after some bigger timeout.

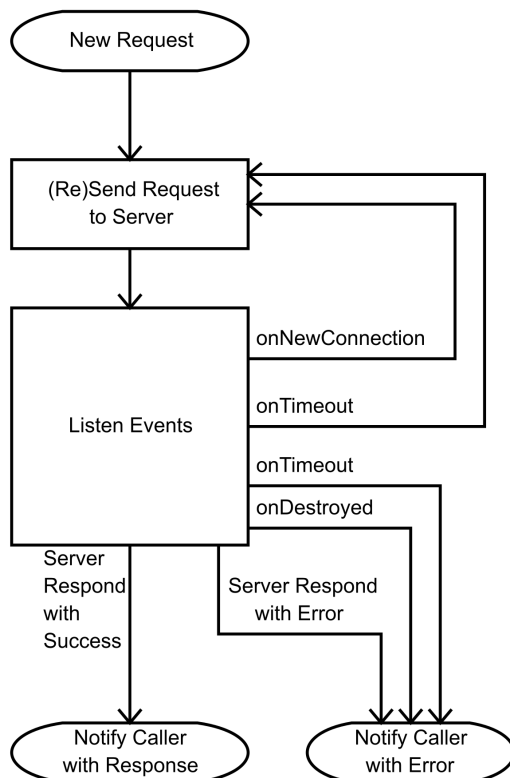
Reconnection condition

1. Connection should be reestablished automatically
2. Destruction of WebSocket RPC should disable all timer and automatic reconnect
3. WebSocket.close can fire an onclose event and can not, this behaviour depends on browser implementation - needs to be sure that component still works correctly.

Detecting Connection Lost

1. WebSocket onclose default event to detect connection lost
2. Custom reliable connection lost detectors
 - a. Busy case - every 20 second check if at least one RPC call got a response, otherwise connection loss detected.
 - b. Idle case - if no call was made, do ping call every 5 minutes, and verify as for busy case condition

WebSocket Message Send Flow



WebSocket RPC Messages Format

Request Message

	Template	Example
Request	%REQUEST_ID% %COMMAND% %REQUEST_PAYLOAD%	58231 PUT-DATA /QG/EU/ROOM-8 { "name": "QG Room", "región": "EU", "type": "quantum" }
Response with result	%REQUEST_ID% %COMMAND% %REQUEST_PAYLOAD%	58231 PUT-DATA SUCCEED
Response with error	CALL_ERROR %REQUEST_ID% %COMMAND% %ERROR_TYPE%	CALL_ERROR 58231 PUT-DATA UNKNOWN_METHOD_CALLED

ERROR_TYPE could be:

- UNEXPECTED_EXCEPTION
- UNKNOWN_METHOD_CALLED
- Any user error

Server Push

Template	Example
%MESSAGE_TYPE% %PAYLOAD%	ENDPOINT_MESSAGE EVA-5 {...JSON...}

WebSocket RPC Methods

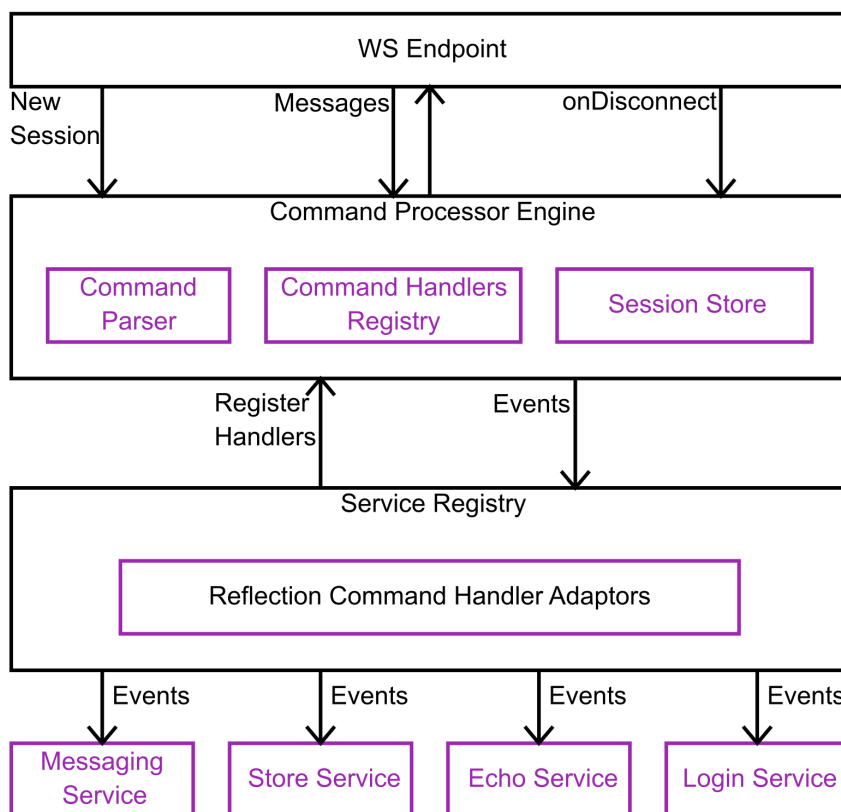
Channel
Send Message
Retrieve Message
Close Connection
Connection on Close

Registry
Put Entry
Get Entry
List of Entries
Compare and Set

VNF WebSocket Server

VNF WebSocket Server is a server that provides services which cannot be done on the Web Browser side. WebSocket Server services fully compatible with WebSocket RPC. It contains three modules: messaging, store and common.

Design of WebSocket Server is divided into Command Processor Engine and Business Service Layer.



Command Processor Engine

Command Processor Engine - is core component WebSocket Server, that integrates all parts together: it contains handlers registry, takes raw unparsed string command as input, parses it, and gives control to appropriate Command handler.

WS Endpoint - is a servlet that converts WS events to Command Processor Events.

Service Registry - adopts annotation based command handling.

Business Services Layer

- Login Service - service to register WS Connection in EVA address space.
- Echo Service - used to ping servers.
- Messaging Service - relay service to retransmit messages between endpoints
- Registry Service - to store registry data.

Technology Stack

Browser Side

Tool/Library	License
JavaScript	Open Source
WebRTC	Built-in browser
WebSocket	Built-in browser
WebPack	Open Source
ESLint	Open Source
Karma, QUnit	Open Source
gulp	Open Source

Server Side

Tool/Library	License
JDK	Open Source
Jetty/JSR-356	Open Source
Slf4j	Open Source
TestNG	Open Source
Gradle	Open Source