The ObjectWeb Consortium

Dream Framework Library

Authors:
M. Leclercq (INRIA)
V. Quéma (INPG)
J.-B. Stefani (INRIA)

Released September 20, 2005
Status Draft
Version 0.9

General Information

Please send comments on this document to `dream@objectweb.org`. Authors would be glad to hear from people using or extending Dream.

Trademarks

All product names mentioned herein are trademarks of their respective owners.

Contents

List of Figures

1 Introduction

This document presents the components available within the DREAM library. Readers are expected to have a good knowledge of the Fractal component model [1] and of the DREAM core documentation [2].

The DREAM component library encompasses several components that implement the various functionalities required for building communication middleware. Among these components, we can find protocols, queues, channels, routers, etc.

2 Queues

This section describes the various queues that are available within the DREAM library. These queues are located into the `org.objectweb.dream.queue` package. Queues are generally composite components following a common architectural pattern: they are made of a set of components that implement the various functionalities needed to build a queue. Each of these components exists in several versions that allow building queues with various characteristics: *push* vs. *pull* inputs/outputs, ordered vs. unordered, active vs. inactive.

We start this section with a description of this architectural pattern. Then, we describe the components found in this pattern. Finally we show how they are assembled to build queues.

2.1 Architectural pattern

The architectural pattern followed by queues is depicted in figure 1. Queues encompass three main components:

- the *Incoming Handler* component handles incoming messages, i.e. messages that are to be stored in the queue. Its role is, for example, to handle overflow by deciding either to block, to delete a message, or to throw an exception.
- the *Buffer* component is in charge of storing messages. It may implement an ordering policy.
- the *Outgoing Handler* component is in charge of delivering stored messages. Its role is, for example, to implement a policy for handling an empty queue: block, returns null, throw an exception, etc.

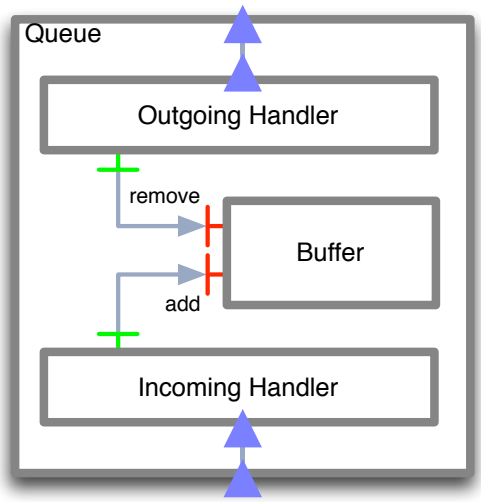

Figure 1: Extensible queues architectural pattern

2.2 Buffers

Buffer components generally provides two interfaces: one to add messages, one to get or remove them. Multiple versions of these interfaces are provided. The simplest are `org.objectweb.dream.queue.BufferAdd` and `org.objectweb.dream.queue.BufferRemove`. Others interfaces allow for instance, to add or remove messages associated with a key.

Buffers provided in the `org.objectweb.dream.queue` package implement the `org.objectweb.dream.queue.BufferAdd` and `org.objectweb.dream.queue.BufferRemove` interfaces. These implementations differ on the ordering of the messages. Messages may be sorted according to their insertion order (`BufferFIFO`, `BufferLIFO`), according to a sequence number (`BufferAscendingSequenceNumber`), or according to a key that is generated for each added message (`BufferSorting`).

2.3 Incoming handlers

Incoming Handlers provide a `in-push` interface and are responsible to add received messages in the buffer. They implements a specific overflow policy:

- `IncomingHandlerBlocking` blocks until the message can be added in the buffer.
- `IncomingHandlerDropping` drops incoming message if the buffer is full.
- `IncomingHandlerExceptionThrowing` throws an exception if the buffer is full.
- `IncomingHandlerRemoving` removes a message from the buffer if it is full.

2.4 Outgoing handlers

Outgoing Handlers provide a `out-pull` interface. When the pull method is called, a message is removed from the buffer and returned. *Outgoing Handlers* provide a specific policy when no message is available in the buffer (the buffer may not be empty if it implements an oderring):

- `OutgoingHandlerBlocking` blocks until a message is available for removing.
- `OutgoingHandlerNonBlocking` returns `null` if no message is available in the buffer.
- `OutgoingHandlerAggregating` returns a message containing every available messages in the buffer.

2.5 Queue personalities

A queue personality is an assembly of a buffer, two handlers and a pump if any (for instance: Push/Pull, FIFO , blocking input and non-blocking output). DREAM provides an easy way to use queue personalities in ADL.

The `org.objectweb.dream.queue.PushPullQueue` ADL describes a Push/Pull queue component containing a *Buffer*, an *Incoming Handler*, an *Outgoing Handler* and a shared *Message Manager*. It takes four arguments:

- the first one specifies the capacity of the *Buffer*;
- the second one specifies the *Incoming Handler* policy (`Bocking`, `Dropping`, `ExceptionThrowing`, or `Removing`);
- the third one specifies the *Outgoing Handler* policy (`Aggregating`, `Blocking`, or `NonBlocking`);
- the last one specifies the *Buffer* implementation (`FIFO`, `LIFO`, `AscendingSequenceNumber`, ...).

So the follwing ADL describe a composite component containing a Push/Pull queue with a capacity of 10 messages, FIFO ordering, blocking input and non blocking output.

```
<definition>
  ...
  <component name="Queue" definition=
    "org.objectweb.dream.queue.PushPullQueue(10,Blocking,NonBlocking,
FIFO)">
    <component name="MessageManager" definition="./MessageManager"/>
  </component>
  ...
</definition>
```

The ADL `org.objectweb.dream.queue.PushPushQueueActive` describes a Push/Push queue component containing a *Buffer*, an *Incoming Handler*, an *Outgoing Handler*, a *Pump*, a shared *Message Manager* and a shared *Activity Manager*. It takes four arguments:

- the first one specifies the capacity of the *Buffer*;
- the second one specifies the *Incoming Handler* policy (`Bocking`, `Dropping`, `ExceptionThrowing`, or `Removing`);
- the third one specifies the *Buffer* implementation (`FIFO`, `LIFO`, `AscendingSequenceNumber`, ...).
- the last one specifies the number of thread in the *Pump*.

So the follwing ADL describe a composite component containing a Push/Push queue with a capacity of 10 messages, FIFO ordering, blocking input and 1 thread.

```
<definition>
  ...
  <component name="Queue" definition=
    "org.objectweb.dream.queue.PushPushQueue(10,Blocking,FIFO,1)">
    <component name="ActivityManager" definition="./ActivityManager"/>
    <component name="MessageManager" definition="./MessageManager"/>
  </component>
  ...
</definition>
```

2.6 Add hoc Queues

DREAM provides some specific monolithic queue implementations. They provides usual policies and have better performance than their generic equivalent (see dreamlib performance results).

- `PushPullQueueFastImpl` is a Push/Pull FIFO queue without capacity limitation and with a blocking output. It is functionally equivalent to `PushPullQueue(0,Blocking,Blocking,FIFO)`.
- `PushPullQueueNotSynchronizedImpl` is a Push/Pull FIFO queue without capacity limitation and with non-blocking output. Moreover the implementation is **not** thread safe. It is almost functionally equivalent to `PushPullQueue(0,NonBlocking,NonBlocking,FIFO)`.

3 Aggregators

This section describes the aggregator components provided by the DREAM library. These aggregators are located in the `org.objectweb.dream.aggregator` package.

An aggregator allows to aggregate multiple messages into a single on. Various implementations are provided, the differences between these implementations are the way that messages are retrieves and how the aggregated message is produced.

The DREAM library provides three aggregators and one disaggregator.

- `PullPullAggregatorFixedNumber` aggregates a fixed number of messages. It provides an `out-pull` interface and as a `in-pull` client interface. When a message is pulled on the `out-pull` interface, the aggregator pulls a fixed number of message on its `in-pull` interface and returns an aggregated message. The number of pulled messages is specified as a fractal attribute.
- `PullPullAggregatorWhileNotNull` as the same type as the previous aggregator but rather than pulling a fixed number of message it pulls messages until `null` is returned. A policy specifies if the component must returns `null` or an empty message when the first pull call returns `null`. This policy is specified as a boolean fractal attribute
- `PullPullAggregatorCollectionInput` collectes messages on its client collection interface. It provides a `out-pull` interface and as a `in-pull` client collection interface. When a message is pulled on the `out-pull` interface, the aggregator pulls a message on each `in-pull` client binding and returns an aggregated message. Messages are pulled sequentially in a non predictable order.
- `PushPushDisaggregator` receives aggregated message on its `in-push` interface and pushed sequentially the sub messages it contains on its `out-push` client interface.

4 Protocols

Protocols components provides abstractions inspired by the JONATHAN framework. These abstractions are mainly based on the Export/Bind pattern. A protocol *exports* interfaces and returns identifiers (`exportId`). Other protocols instances can bind explicitly or silently to the exported interface using the associated `exportId`.

Protocols component implementations provides different way to exchange messages and various non functional properties. The main message exchange paradigms are **Message Channel** and **Message Passing**.

4.1 Message Channel protocols

A message channel is a two way, point to point communication channel (typically a TCP socket). Message Channel protocols provides the following interface:

```
public interface ChannelProtocol {
  ExportIdentifier export(ChannelFactory factory, Map hints);
  OutgoingPush bind(ExportIdentifier id, IncomingPush toClientPush);

  ExportIdentifier createExportId(Map info);
}
```

The `export` method exports a `ChannelFactory` interface and returns an `exportId` which uniquely identify the exported channel factory.

The `ChannelFactory` interface has the following definition:

```
public interface ChannelFactory {
  IncomingPush instantiate(OugoingPush toClientPush);
}
```

The `bind` method of the `ChannelProtocol` interface creates a new communication channel between the client (the component which call the `bind` method) and the server (the component which has exported the `ChannelFactory` interface). This method takes as parameter the export identifier of the exported channel factory and the `IncomingPush` interface on which the client wants to receive messages sent by the server. The method returns a `OutgoingPush` interface on which the client can send messages to the server.

On the server side when the binding is established, the `instantiate` method of the exported `ChannelFactory` interface is called. The given `OutgoingPush` interface can be used to send message to the client. The method returns the a `IncomingPush` interface on which messages coming from the client will be passed.

Protocol components may handle multiple communication channels at a time, each of these are implemented by *session* components.

Session components have `OutgoingPush` interfaces to send messages down the protocol stack, and `IncomingPush` interfaces for messages going up the protocol stack.

On server side, *Session factory* components are responsible to create session components. These factory components are instantiated at export time (a session factory is associated with an exported `ChannelFactory` interface). On client side, session components are created at bind time by the protocol itself.

4.1.1 The TCP protocol

The TCP/IP protocol component uses TCP sockets to implement communication channel. The export operation creates a `ServerSocket` wrapped in a session factory component, allocates a task that listen for incoming connections, and returns an export identifier containing the local host name and the listening port of the server socket.

- On client side, the bind operation open a socket using information contained in the given export identifier, creates a session component that wraps the socket, binds this component to the provided `IncomingPush` interfaces and returns the `OutgoingPush` interface the session provides.
- On server side, when an incoming connection is opened on the server socket, the session factory creates a session component, calls the `instantiate` method of the exported `ChannelFactory` interface giving as `OutgoingPush` interface the interface provided by the session component. Finally the session is bound to the `IncomingPush` interface returned by the `instantiate` method.

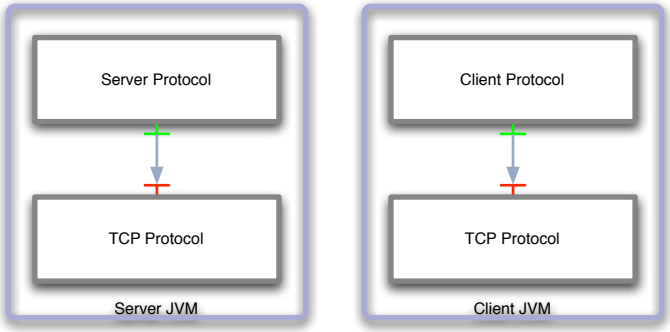

Figure 2: TCP protocol use case

Figure3 is a sequence diagram of the simple use case depicted in figure2. This exmaple is a client/server distributed application where the *Server Protocol* exports a `ChannelFactory` and the *Client Protocol* binds to it and sends a message. These three steps are shown in figure3 by the orange numbers.

1. The *Server Protocol* creates a *Channel Factory* and exports it through the TCP protocol. The TCP protocol creates a *Session Factory* component which wraps a `ServerSocket`,

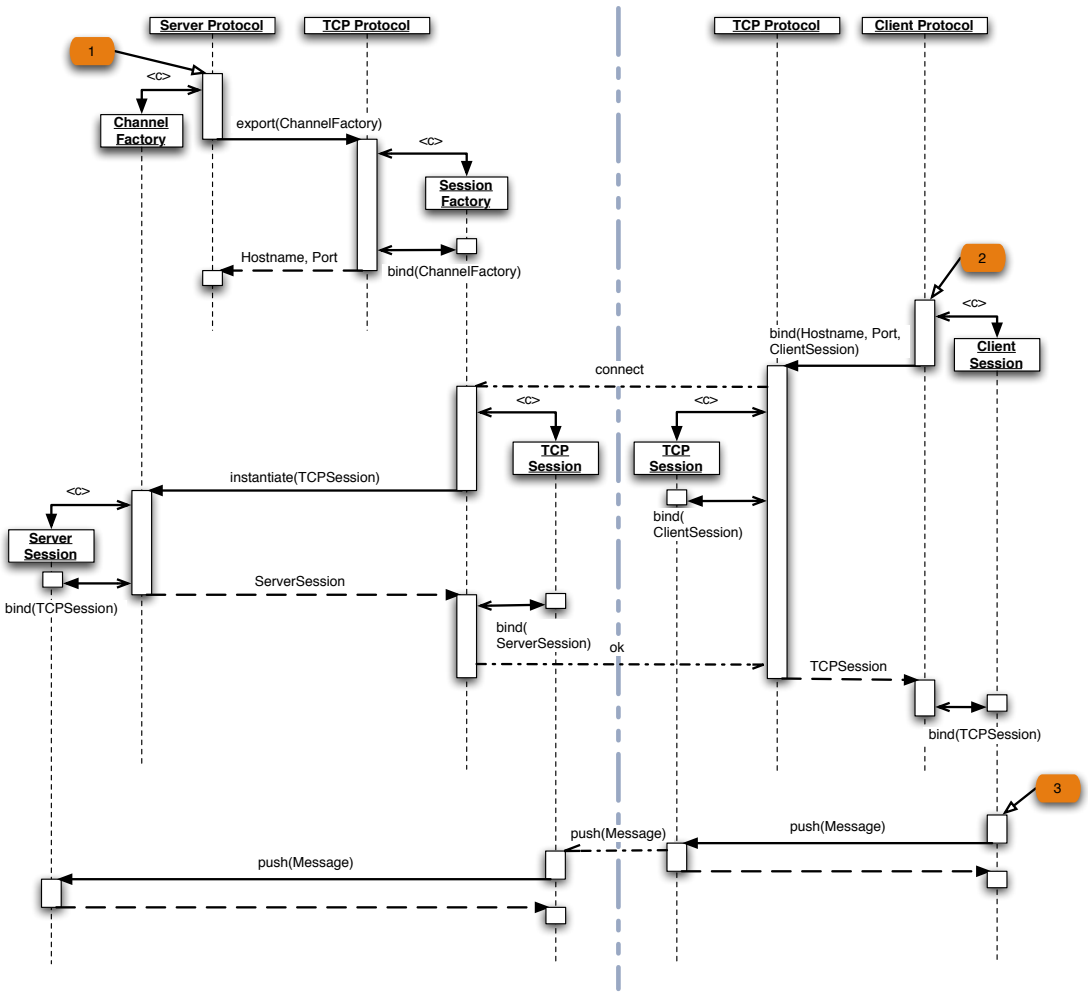

Figure 3: TCP protocol sequence diagram

binds it to the channel factory it is exporting and returns an export identifier containing the local host name and the listening port of the server socket.

2. The *Client Protocol* creates a *Client Session* and binds it providing the export identifier of the previously exported *Channel Factory* and the .`IncomingPush` interface provided by the *Client Session*

   - On client side, the TCP protocol opens a TCP socket using the host name and port contained in the export identifier, and creates a *TCP Session* which wraps this socket.
   - On server side, when the *Session Factory* receives the new connection, it creates a *TCP Session* which wraps the new socket, then call the `instantiate` method of the *Channel Factory* component, giving the `OutgoingPush` interface provided by the *TCP session* as parameter.

- The *Channel Factory* creates a *Server Session* component, binds it to the *TCP Session* and returns the `IncomingPush` interface the *Server Session* provides.
- The *Session Factory* binds the *TCP Session* to the *Server Session* and sends a message to the client to inform it that the channel has been correctly instantiated (this message may contains the exception thrown by the `instantiate` method).
- When the *TCP Protocol*, on client side, receives this message, it binds the *TCP Session* to the *Client Session*, and returns the `OutgoingPush` interface the *TCP Session* provides.
- Finally the *Client Protocol* binds the *Client Session* to the *TCP Session*.

A distributed message channel is now instantiated between the *Client Session* and the *Server Session*.

3. The *Client Session* pushes a message to the *Server Session* using the *TCP Session* it is bound to. The message is serialized and send to the remote *TCP Session* which de-serializes it and pushes it to the *Server Session*.

4.1.2 Multiplexing exports

Export multiplexer protocol allows to export multiple channel factory over a single lower level export. This allows for instance to share a single server socket over multiple protocol components.

The first time the `export` method of the export multiplexer protocol is called, the protocol exports its own channel factory to the lower level protocol. Then for each latter exports, the protocol allocates a unique number to the channel factory it is exporting, and returns an export identifier containing the unique number and the export identifier of its own exported channel factory (called the lower level export identifier).

The `bind` method calls the `bind` method of the lower level protocol giving the lower level export identifier. Then it sends as first message, the unique number contained in the export identifier. At server side, when the `instantiate` method of the channel factory exported by the export multiplexer protocol is called, a new session is created. When the session receives the first message, it retrieves the unique number it contains, and call the `instantiate` method of the channel factory associated with this number.

Figure5 is a sequence diagram of the simple use case depicted in figure4. This exmaple is a client/server distributed application where the two diffrent *Server Protocols* exports a `ChannelFactory` and the *Client Protocol* binds to one of them and sends a message. These four steps are shown by the orange numbers.

1. The *Server Protocol 1* creates and exports the *Channel Factory 1*. The *Multiplex Export Protocol* creates a *Channel Factory MultiExp*, exports it through the *Lower Protocol* and returns an export identifier containing a unique number (here 1) and the lower export identifier (which identify the *Channel Factory MultiExp*).
2. The *Server Protocol 2* creates and exports the *Channel Factory 2*. Since the *Multiplex Export Protocol* as already exported its *Channel Factory MultiExp* through the *Lower*

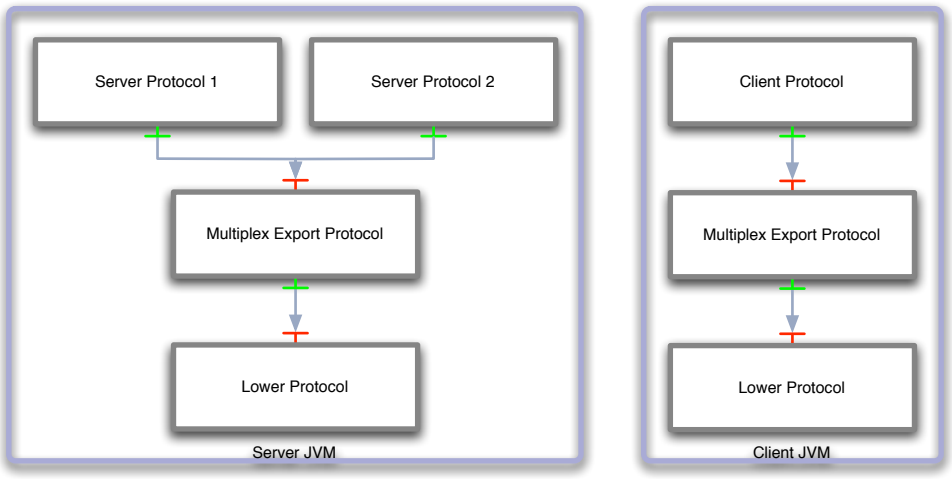

Figure 4: Export multiplexer protocol use case

Protocol, it simply returns an export identifier containing a unique number (here 2) and the lower export identifier.

3. The *Client Protocol* component creates a *Client Session 1* and binds it providing the export identifier of the *Channel Factory 1* and the `IncomingPush` interface the *Client Session* provides.

   - The client side *Multiplex Export Protocol*, creates a *MultiExp Session 1* and binds it to the *Channel Factory MultiExp* through the *Lower Protocol*.
   - On server side , the `instantiate` method of the *Channel Factory MultiExp* is called by the *Lower Protocol*. The *Channel Factory MultiExp* creates, binds and returns the *MultiExp Session 1*.

   At this time a message channel is instantiated between the *MultiExp Session 1* on client side and the *MultiExp Session 1* on server side.

   - On client side, the *Multiplex Export Protocol* pushes through this message channel a message containing the number of the *Channel Factory* it want to be bound.
   - The message is received by the server side *MultiExp Session 1*. It retrieves the number the message contains and call the `instantiate` method of the corresponding *Channel Factory* (in our case *Channel Factory 1*).
   - The *Channel Factory 1* creates, binds and returns the *Server Session 1*.
   - The server side *MultiExp Session 1* sends a reply message to inform that the message channel has been instantiated correctly.
   - When the client side *MultiExp Session 1* receives the reply, it returns itself to the *Client Protocol*. A distributed message channel is now instantiated between the *Client Session 1* and the *Server Session 1*.

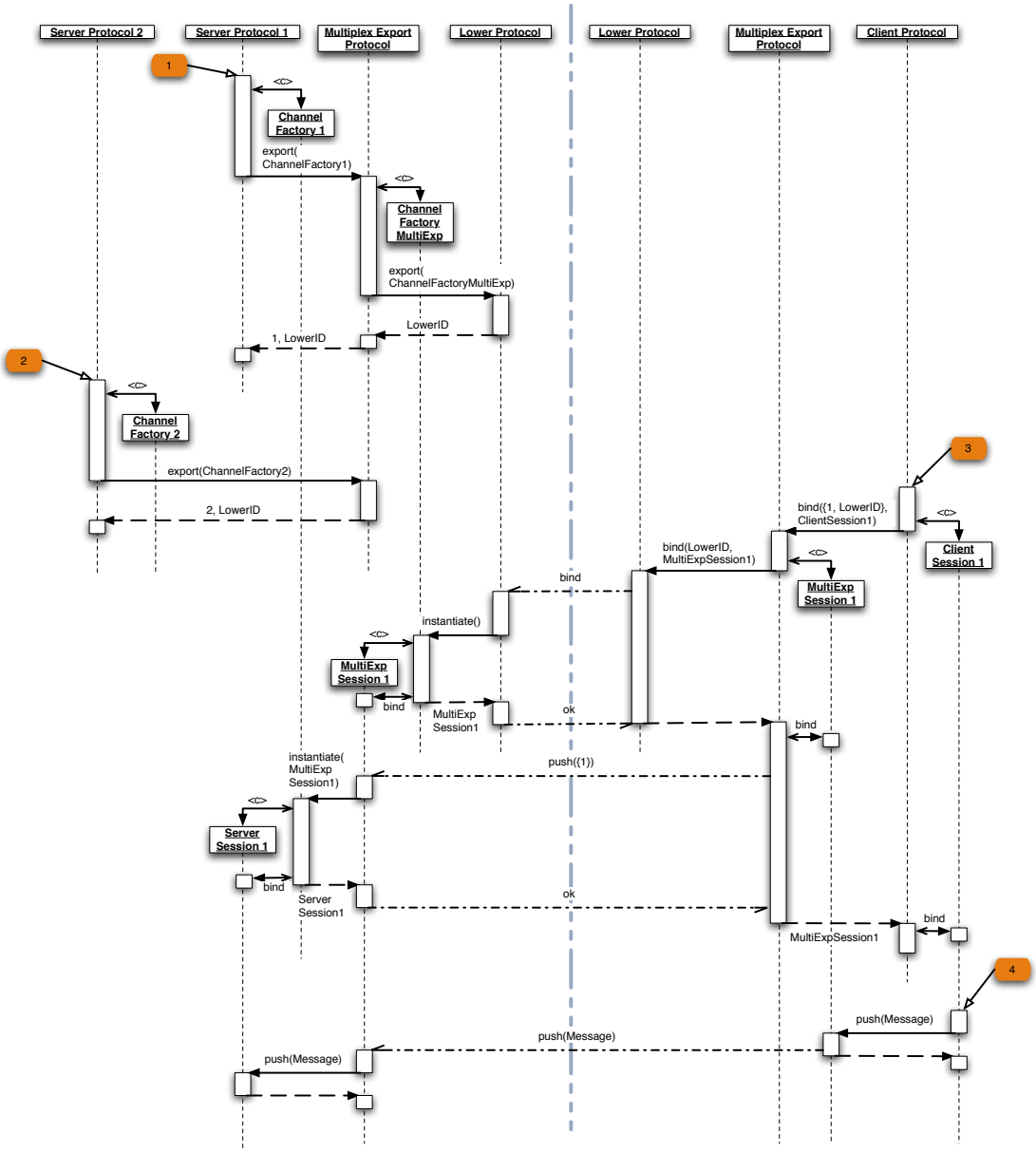

Figure 5: Export multiplexer protocol sequence diagram

- The *Client Session 1* pushes a message to the *Server Session 1* using the *MultiExp Session 1* it is bound to. The message is passed down to the message channel and received by the remote *MultiExp Session 2* which pushes it to the *Server Session 1*.

4.1.3 Multiplexing bindings

Binding multiplexer protocol allows to multiplex multiple communication channels over a single

one. This is useful when multiple clients on the same JVM want to open communication channels to the same channel factory.

The `export` method simply creates a session factory which is exported using the lower level protocol. The returned export identifer is the export identifier returned by the lower level protocol.

The `bind` method looks if a channel has already been opened with the same export identifier. If it is not the case, a new session is created and bound through the lower level protocol. If a channel already exists, a special `bind` message is sent on it. When this message is received by the session on the server side, the `instantiate` method of the channel factory is called. Since sessions component multiplex multiple channels over a single one, a chunk containing an identifer is put in every outgoing messages allowing the remote session to deliver messages to the correct upper channel.

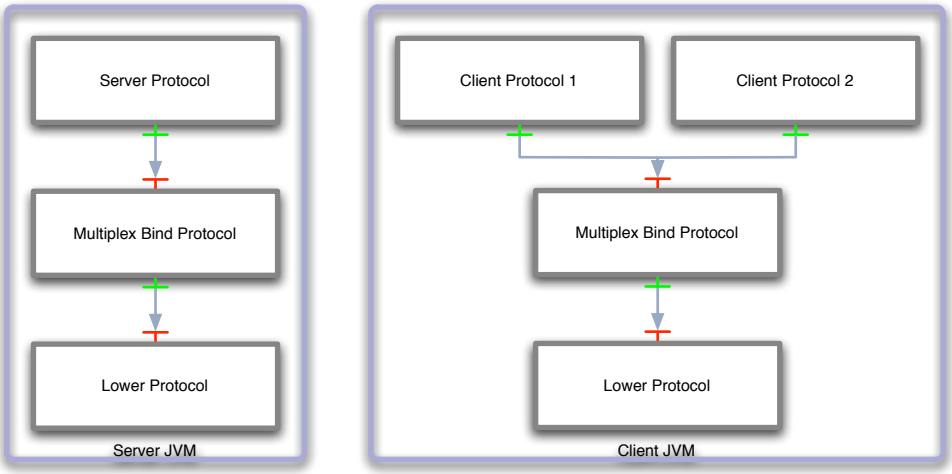

Figure 6: Bind multiplexer protocol use case

Figure7 is a sequence diagram of the simple use case depicted in figure6. This exmaple is a client/server distributed application where the *Server Protocol* exports a `ChannelFactory` and two different *Client Protocols* bind to it and one of them sends a message. These four steps are shown by the orange numbers.

1. The *Server Protocol* creates and exports the *Channel Factory*. The *Multiplex Bind Protocol* creates a *Channel Factory MultiBind*, exports it through the *Lower Protocol* and returns the lower export identifier.

2. The *Client Protocol 1* component creates a *Client Session 1* and binds it providing the export identifier of the *Channel Factory* and the `IncomingPush` interface the *Client Session 1* provides.

   - The *Multiplex Bind Protocol* creates a *Demux* component and call the `bind` method on it.

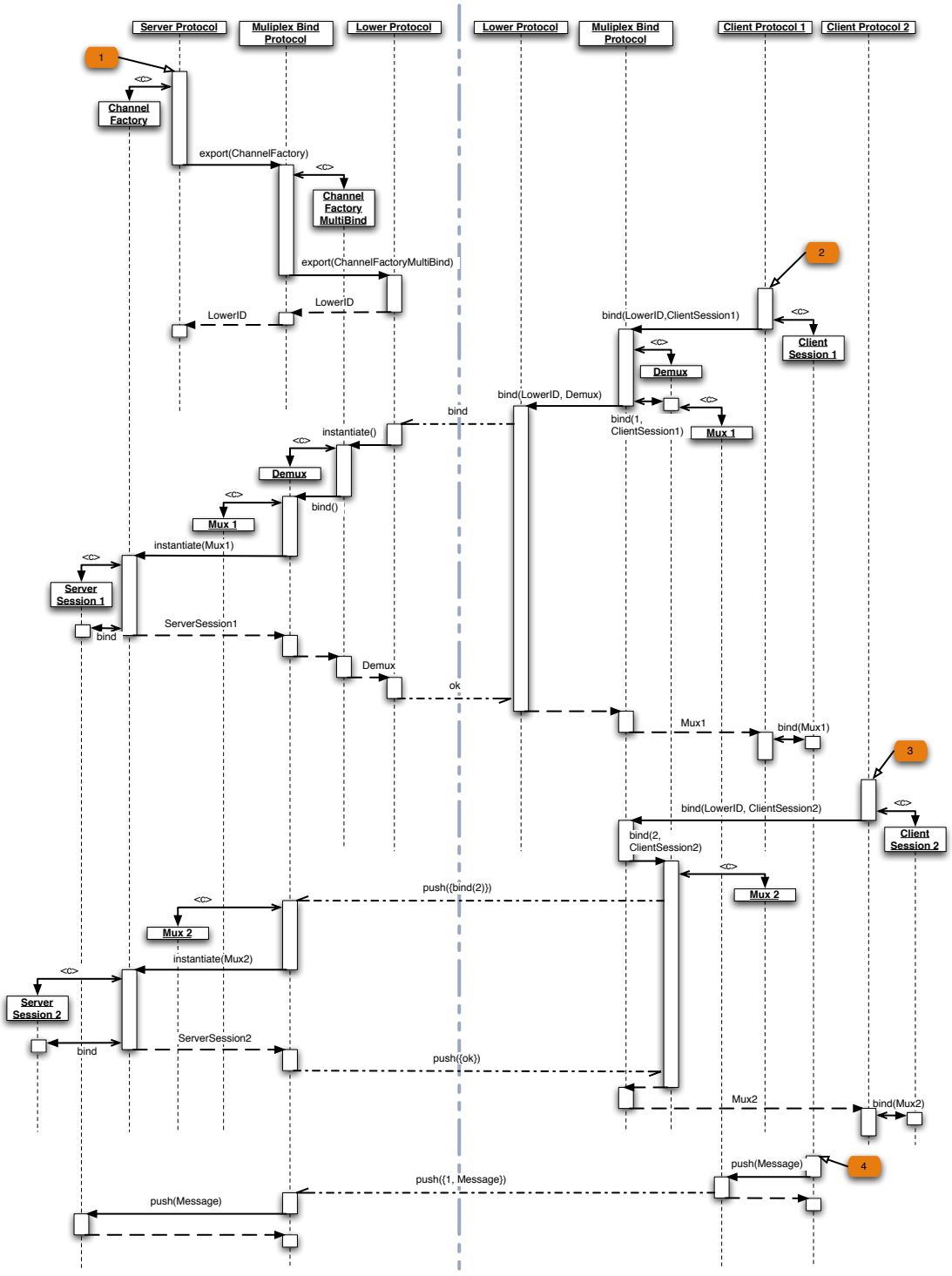

Figure 7: Bind multiplexer protocol sequence diagram

- The `bind` method creates the *Mux 1* component. [1]

[1] The *Mux* component is responsible to add to outgoing messages a MultiplexChunk containing a number

- The *Multiplex Bind Protocol* binds the *Demux* component to the *Channel Factory MultiBind* through the *Lower Protocol*.
- On server side , the `instantiate` method of the *Channel Factory MultiBind* is called by the *Lower Protocol*. The *Channel Factory MultiBind* creates a *Demux* component and call the `bind` method on it.
- The `bind` method creates the *Mux 1* component and calls the `instantiate` method of the *Channel Factory* giving the *Mux 1* as `OutgoingPush`.
- The *Channel Factory* creates, binds and returns the *Server Session 1*.
- The server side *Multiplex Bind Protocol* sends a message to the client to inform it that the channel has been correctly instantiated.
- On the client side, the message is received and the *Multiplex Bind Protocol* returns the *Mux 1* component as `OutgoingPush` interface that can be used to send messages to *Server Session 1*.

3. The *Client Protocol 2* component creates a *Client Session 2* and binds it providing the export identifier of the *Channel Factory*.

   - The *Multiplex Bind Protocol* finds that a message channel already exists for the given export identifier, it simply call the `bind` method on corresponding *Demux* component.
   - The `bind` method creates the *Mux 1* component and pushes a message containing a `BindChunk` through the lower message channel.
   - The message is received by the server side `Demux` component. It retrieves the `BindChunk`, creates a `Mux 2` component and calls the `instantiate` method of the *Channel Factory* giving the *Mux 2* as `OutgoingPush`.
   - The *Channel Factory* creates, binds and returns the *Server Session 2*.
   - The server side *Demux* sends a message to the client side *Demux* to inform it that the channel has been correctly instantiated.
   - On the client side, the message is received and the *Multiplex Bind Protocol* returns the *Mux 2* component.

   At this time two message channels have been instantiated, the first one betweeen *Client Session 1* and *Server Session 1* and the second one betweeen *Client Session 2* and *Server Session 2*. The organization of the various session components involved in these message channels is depicted in figure8.

4. The *Client Session 1* pushes a message to the *Server Session 1* using the *Mux 1* it is bound to.

   - The *Mux 1* component adds a MultiplexChunk containing the number of the message channel (in our case 1), and pushes down the message through the lower message channel.
   - The message is received by the server side *Demux* component. It retrieves the MultiplexChunk and pushes the message to the *Server Session 1*

identifying the channel on which they are sent. The *Demux* component retrieves this MultiplexChunk in incoming messages and forward it on the right channel (see figure8).

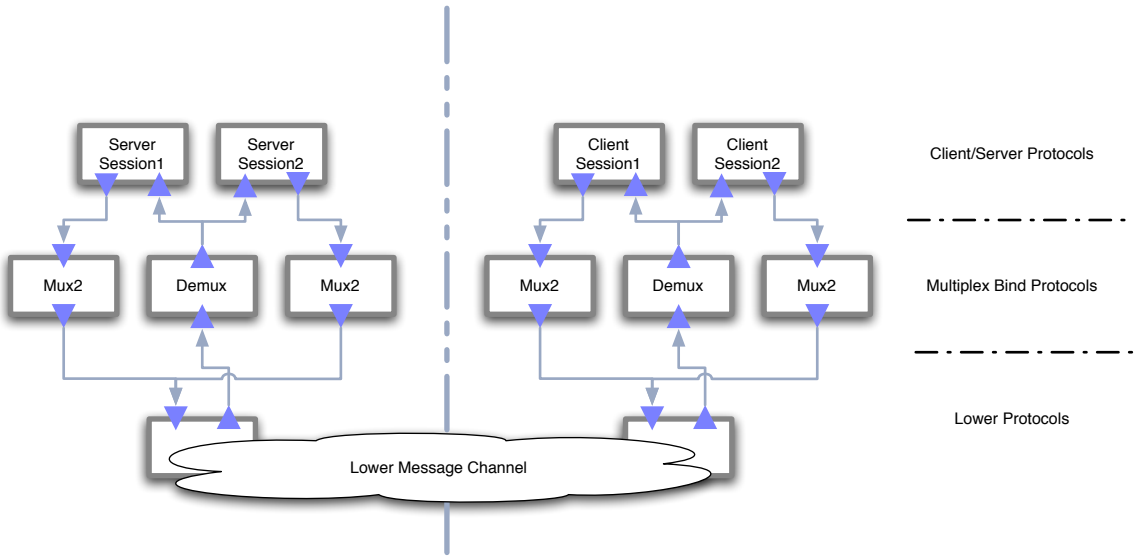

Figure 8: Bind multiplexer protocol sessions

4.2 Message Passing protocols

Message passing protocols allows to open access points to a message bus, and to send or receive messages on/from it. These protocols in opposite to channel protocol are not connection oriented. This means that they do not provide a `bind` operation.

Message Channel protocols provides the following interface:

```
public interface MessagePassingProtocol {
  MessagePassingOutgoingPush export(IncomingPush incomingPush,
      Map hints);

  ExportIdentifier createExportId(Map info);
}
```

The `export` method opens a local access point and returns a `MessagePassingOutgoingPush` interface that can be used to send messages over the message bus. The given `IncomingPush` interface is the interface on which incoming messages will be passed.

The `MessagePassingOutgoingPush` interface as the following signature:

```
public interface MessagePassingOutgoingPush {

  ExportIdentifier getLocalExpotId();
  void outgoingPush(Message message, ExportIdentifier to);

  void outgoingClose();
}
```

The `outgoingPush` can be used to send messages to remote sites identified by the given export identifier.

The `getLocalExpotId` method returns the identifier of the local access point. More precisely, if, on a remote site, a message is sent with this export identifier as destination, the message will be received by the local access point.

4.2.1 The UDP protocol

The UDP protocol component uses UDP socket to implement message bus access point.

The `export` operation creates a `DatagramSocket` wrapped in a session component, allocates a task that retrieves and pushes incoming messages (if the given `IncomingPush` interface is not `null`), and returns the `MessagePassingOutgoingPush` interface the session component provides.

The export identifiers this protocol handles, contain hostname and port of datagram sockets. So the `getLocalExpotId` method returns an export identifier containing the local hostname an the local port of the datagram sockets. The `outgoingPush` takes as destination an export identifier containing the hostname and the port of the remote datagram sockets.

The UPD Protocol implementation can also add to incoming messages an `ExportIdentifierChunk` containing the identifier of the sender of the message (see `MessagePassingProtocol#FROM_CHUNK_NAME` and `UDPProtocol.export`).

4.2.2 Message passing over channel protocol

This protocol is a bridge between the message passing abstraction an message channel. It provides a message passing protocol interface and uses a message channel protocol as lower level protocol.

This protocol allows for instance to use a message passing protocol over reliable TCP channels rather than UDP sockets.

The protocol manage a cache of channels. These channels are opened and closed transparently by the protocol.

The `export` operation creates a *Session Manager* compnent which provides the `MessagePassingOutgoingPush` interface and the `ChannelFactory` interface. This *Session Manager* is exported through the lower message channel protocol, the returned export identifier will be used as the local access point identifier.

The `outgoingPush` method of the *Session Manager* try to find if a channel is already open for the given destination.

- If a channel is found, the message is pushed through this channel.
- If no channel can be found, the *Session Manager* calls the `bind` method of the lower level protocol giving the destination as identifier, adds the channel in the cache, finally the message is pushed through this new channel.

The `instantiate` method of the *Session Manager* is called when a remote protocol has a message to send to this one. In this case the remote *Session Manager* sends (as first message) a message containing its local identifier. This way, the local *Session Manager* knows who is at the other end of the channel and can put it in the cache of opened channels.

Moreover, the size of the channels cache is limited, so if a channel must be opened to send a message, and the cache is full, a channel must be closed before opening a new one. The implementation manages a LRU list in which channels are moved to the head of this list each time they are used (for sending or receiving a message). When a channel must be closed, the *Session Manager* closes the channel at the tail of the list.

In some case two channels may be opened for the same destination. This may happend if two protocols exchange two messages at the same time, while no channel is already opened between these two protocols.

Lets take the example of two protocols (`P1` and `P2`). Each one has opened an access point using its respective *Message passing over channel* protocol, these two access points will be called `SM1` and `SM2` (`SM` as *Session Manager*). At the same time, `P1` sends a message to `P2` and `P2` sends a message to `P1`.

Since `SM1` has no opened channel for `SM2`, it opens a new one (called `C1`). For the same reason, `SM2` opens a channel to `SM1` (called `C2`). In fact the `instantiate` method of `SM2` may have already been called (since `SM1` is opening a channel to `SM2`) but we suppose that it has not yet received the message containing the identifier of this new channel, so `SM2` does not know that `C1` is a channel for `SM1` (this kind of channels is called anonymous channel).

At this time: for `SM1`, `C1` is a channel to `SM2` and `C2` is an anonymous channel; and for `SM2`, `C2` is a channel to `SM1` and `C1` is an anonymous channel.

`SM1` pushes through `C1` a message containing its identifier and the message `P1` is sending. `SM2` pushes through `C2` a message containing its identifier and the message of `P2`.

When `SM1` receives through `C2` the identifier of `SM2`, it finds that it has already an opened channel for `SM2` in its cache. Idem for `SM2`. `SM1` and `SM2` must choose a channel to close and they must be sure that no message is being transmited by the channel they will close.

Each one computes a hash of their identifiers and compares them. If the local identifier is lower than the remote one, the channel in the cache must be closed and replaced by the anonymous channel.

Lets supposes that the hash of the identifier of `SM1` is lower than the hash of the identifier of `SM2`. So `C1` must be closed and replaced by `C2` in the cache of `SM1`. But to be sure that no message is being transmited in `C1`, `SM1` does not close `C1` directly. It sends through it a *close message*. When `SM2` receives this message, it closes `C1`.

Finally `SM1` and `SM2` has a unique channel (`C2`) between them, which can be used to send messages from `SM1` to `SM2` and messages from `SM2` to `SM1`.

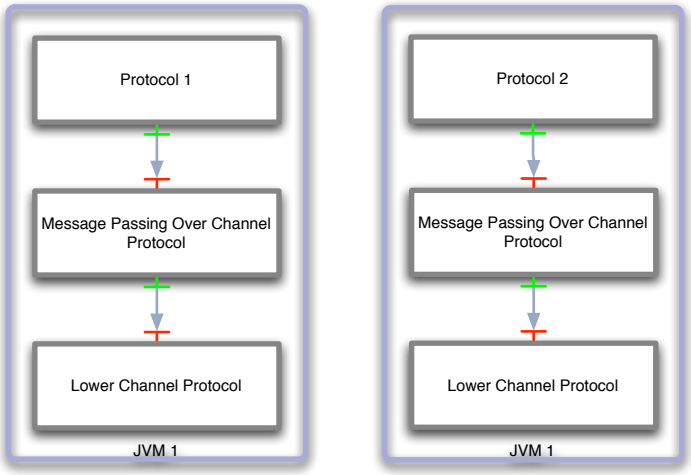

Figure 9: Message passing over channel protocol use case

Figure10 is a sequence diagram of the simple use case depicted in figure9. This exmaple is a peer-to-peer distributed application where the *Protocol* components export an access point and exchange two messages. These different steps are shown by the orange numbers.

1. *Protocol 1* and *Protocol 2* create and export a *Session* component. The *Message Passing Over Channel Protocol* creates a *Session Manager* and exports the `ChannelFactory` interface it provides through the *Lower Channel Protocol*. The returned export identifier is set as the local export identifier.
2. *Session1* pushes a message to *Session2*.
   - *Session Manager 1* does not find a channel in the cache for the given destination. It creates the *Session to ID2* component and binds it to *Session Manager 2* using *Lower Channel Protocol 1*.
   - The `instantiate` method of *Session Manager 2* is called. It creates and returns a new session component.
   - The *Session Manager 1* calls the `bind` method of *Session to ID2*. This method pushes a message containing the local export identifier.
   - This message is received by the session created by *Session Manager 2*. It retrieves the export identifier *ID1* and adds itself in the cache of *Session Manager 2* as *Session to ID1*.
   - The *Session Manager 1* finally calls the `push` method of *Session to ID2*. The message is received by *Session to ID1* and pushed to *Session 2*.
3. *Session 2* pushes a message to *Session 1*. *Session Manager 2* finds an opened channel for the given destination, pushes the message through it. It is received by *Session to ID2* and pushed to *Session 1*.

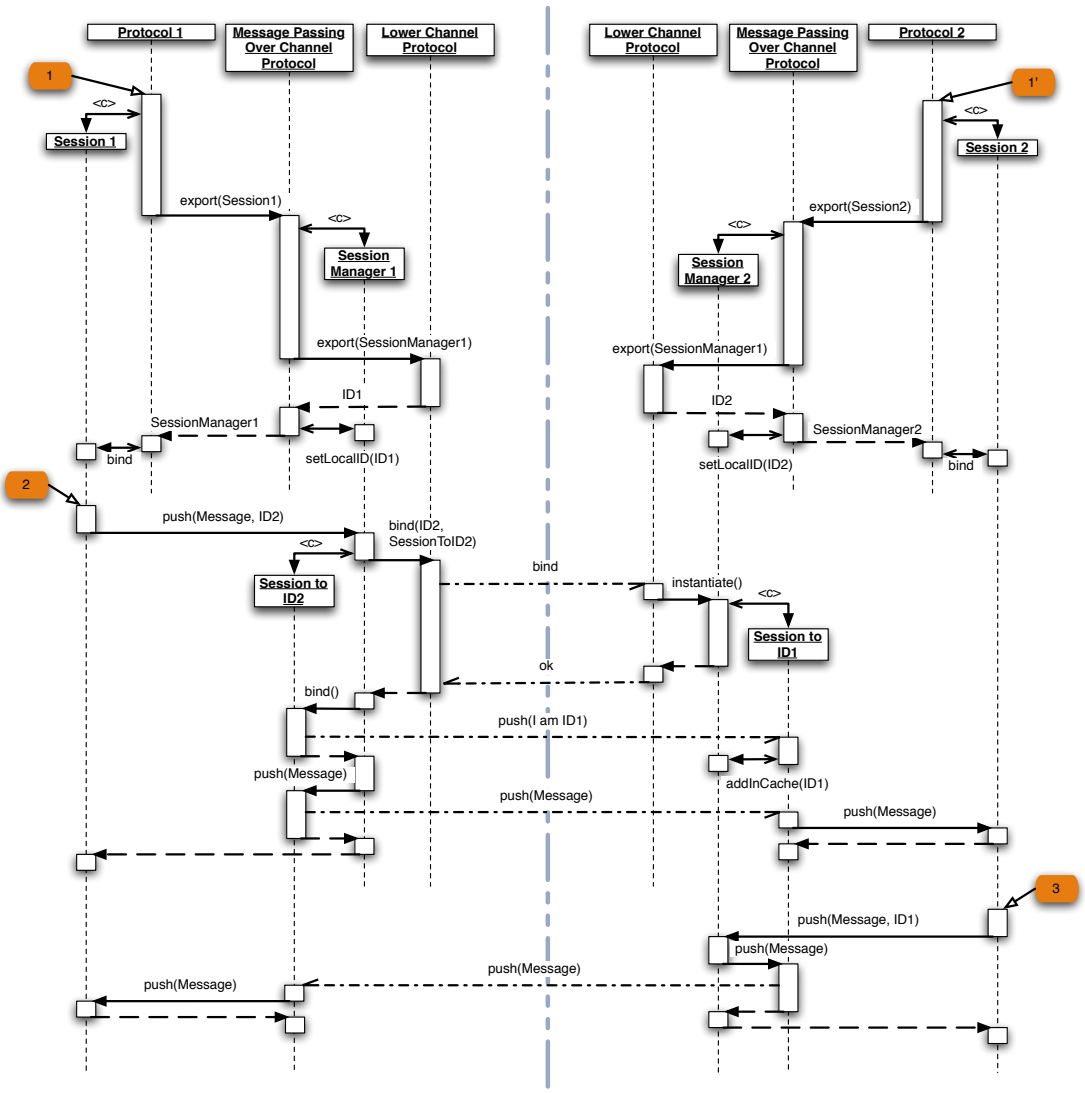

Figure 10: Message passing over channel protocol sequence diagram

4.2.3 Buffering protocols

Buffering protocols allow to decouple execution flow from message flow. They can decouple incoming or outgoing messages. Decoupled messages are put in a waiting list and sent by an independent task.

4.2.4 Fragmentation protocol

Fragmentation protocol serializes fragments outgoing messages in order to push byte array messages with a maximum size. Incoming messages fragments are gathered and deserialized.

4.3 RPC protocol

we implement a RPC protocol over channel protocol. This implementation has been made for experimentation, and is not yet usable.

- export : export a couple stub/skeleton
- bind : retrieve a stub

Bindings to the same skeleton are multiplexed by the RPC protocol to avoid unnecessary creation of communication channel for the same skeleton.

5 Channels

Channels components make the use of protocol components easy! They allows to exchange messages from different address spaces. *Channel* components uses *Message Passing Protocol* to send and receives messages. When a *Channel* component is started, it exports itself through the lower *Message Passing Protocol*.

A *Channel* provides an `in-push` interface on which it receives messages that must be sent to another address space. The destination of the message (the `export identifier` of the remote access point) is resolved by a *Destination Resolver* component.

A *Channel* may also have a `out-push` client interface on which messages received from other address spaces will be pushed. *Channels* that has an `out-push` interface are named *Channel In/Out*, those who do not are named *Channel Out*.

Figure11 depicts an example of a channel component which uses *TCP Protocol* and *Message Passing Over Channel Protocol*.

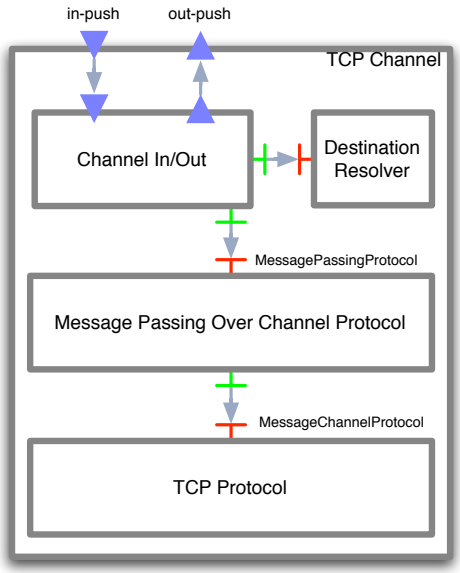

Figure 11: TCP channel

The *Destination Resolver* implementation provided by DREAM retrieves the destination of each message in an `ExportIdentifierChunk`. A fractal attributes can be used to specify the name of the chunk, and if it should be removed before the message is sent.

Moreover, *Channel In/Out* implementation provides an attribute that allows to specify if an `ExportIdentifierChunk` should be added to incoming messages containing the identifier of the *Channel* that emits it.

DREAM provides the following channel ADL:

- `orb.objectweb.dream.channel.ChannelInOutTCPStack`: A *Channel In/Out* using the

TCP Protocol and the *Message Passing Over Channel Protocol* (as depicted in figure 11).

- `orb.objectweb.dream.channel.ChannelOutTCPStack`: As the previous one but can only emits messages.
- `orb.objectweb.dream.channel.ChannelInOutUDPStack`: A *Channel In/Out* using the *UDP Protocol*
- `orb.objectweb.dream.channel.ChannelOutUDPStack`: As the previous one but can only emits messages.

6 Pumps

A *Pump* component is a bridge between `Pull` and `Push`. It contains a task and has two client interfaces `in-pull` and `out-push`. The task pulls a message on the `in-pull` interface and pushes it on the `out-push` interface. The task may be register in the *Task Manager* as mono threaded, multi threaded, or periodic.

Moreover a pump may be synchronized using a *Mutex* component (the *Pump* component has an optional `mutex` client interface). The mutex is locked before a message is pulled and released after it is pushed.

A pump has three policies that can be specified as fractal attributes:

- `pushNullPolicy` indicates whether `null` messages must be pushed or ignored.
- `stopOnPushExceptionPolicy` indicates whether the task must be stopped when a `PushException` occurs.
- `stopOnPullExceptionPolicy` indicates whether the task must be stopped when a `PullException` occurs.

7 Routers

Router components have one `Push` input and multiple `Push` outputs, for each incoming message, they choose an output and forward the message through it. The DREAM library provides three simple *Router* implementations in the `org.objectweb.dream.router` package.

- `RouterChunkName` has two outputs, it chooses the output depending on the presence in the message of a chunk with a given name. This name is specified as a fractal attribute. Messages that contain the chunk, are pushed on the output named `out-push-with-chunk`; those that do not, are pushed on the output `out-push-without-chunk`.
- RouterRandom has a collection of outputs. For each incoming message, the output is chosen randomly.
- `RouterRoundRobin` has a collection of outputs. It implements a round-robin choice of its outputs.

8 Serialization/De-serialization components

This section describes the *Serialization/De-serialization* components available in the DREAM library. These components are located in the `org.objectweb.dream.serializator` package. They allow to easily transforms a message into a byte array or restore a message from a byte array.

- The *Serializator* component provides the following interface:

```
public interface Serializator {
  byte[] serialize(Message message) throws IOException;
}
```

  It uses a *Message Codec* to encode the message and returns a byte array representation of it. This components also uses an *Object Pool* of `CodecInputOutput`

- The *De-Serializator* component provides the following `interface`:

```
public interface DeSerializator {
  Message deserialize(byte[] byteArray) throws IOException;
}
```

  It uses a *Message Codec* to decode the byte array.

- The *Push Push Serializator* uses a *Serializator* to encode incoming messages and pushes a new message with a `ByteArrayChunk` containing the byte array representation of the incoming message. The component can preserve a chunk of the incoming message from being serialized. The chunk is removed before the message is encoded and added in the created message in addition to the `ByteArrayChunk`. The name of this preserved chunk can be specified as a fractal attribute.

- The *Push Push De-Serializator* does the reverse operation of the previous component. It receives incoming message containing a `ByteArrayChunk`, decodes its content using a *De-Serializator*, transfers additional chunks in the decoded message and forwards it.

References

[1] The Fractal Component Model. `http://fractal.objectweb.org`.

[2] M. Leclercq, V. Quema, and J.-B. Stefani. The Dream Framework Core. `http://dream.objectweb.org/dreamcore/`.