MQ4CPP
Message Queuing For C++

Riccardo Pompeo

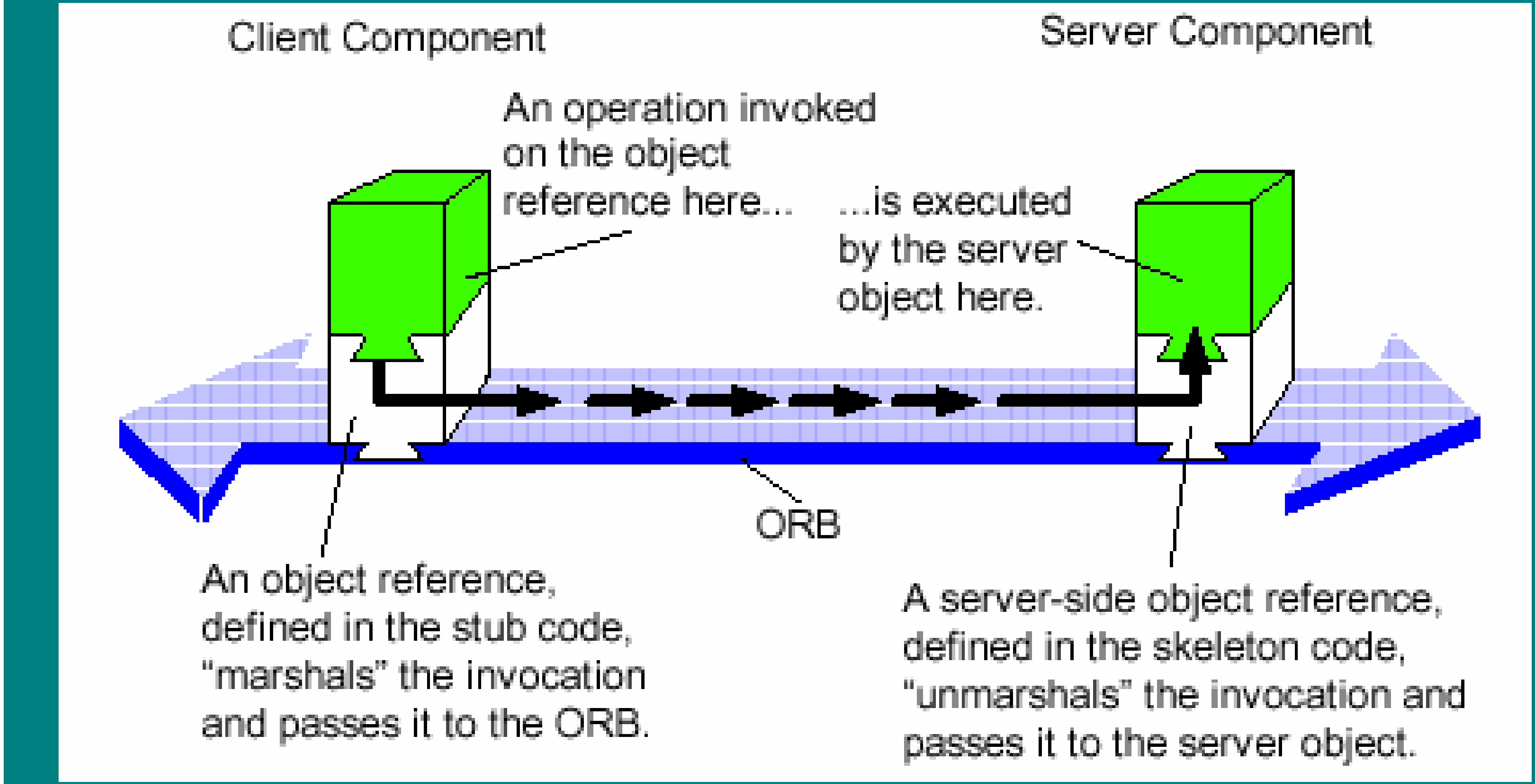
What is?

- MQ4CPP is a Message-Oriented Middleware (MOM) and implements the following messaging paradigms:
  - Direct/Indirect messaging (local)
  - Unsolicited messaging (remote)
  - Request/Reply (remote)
  - Conversation (remote)
  - Broadcast (local/remote)
  - Publish/Subscribe
  - Store & Forward
  - Memory Channel
  - File Transfer
  - Distributed Lock Manager

- Support of:
  - Multithreading (pthread, Win Thread)
  - Sockets (berkley , Win Sock2)
  - Cluster (failover, session replication)
  - Encription (Rijndael 128/256)
  - Compression
  - Service lookup (local/remote)
  - Message routing

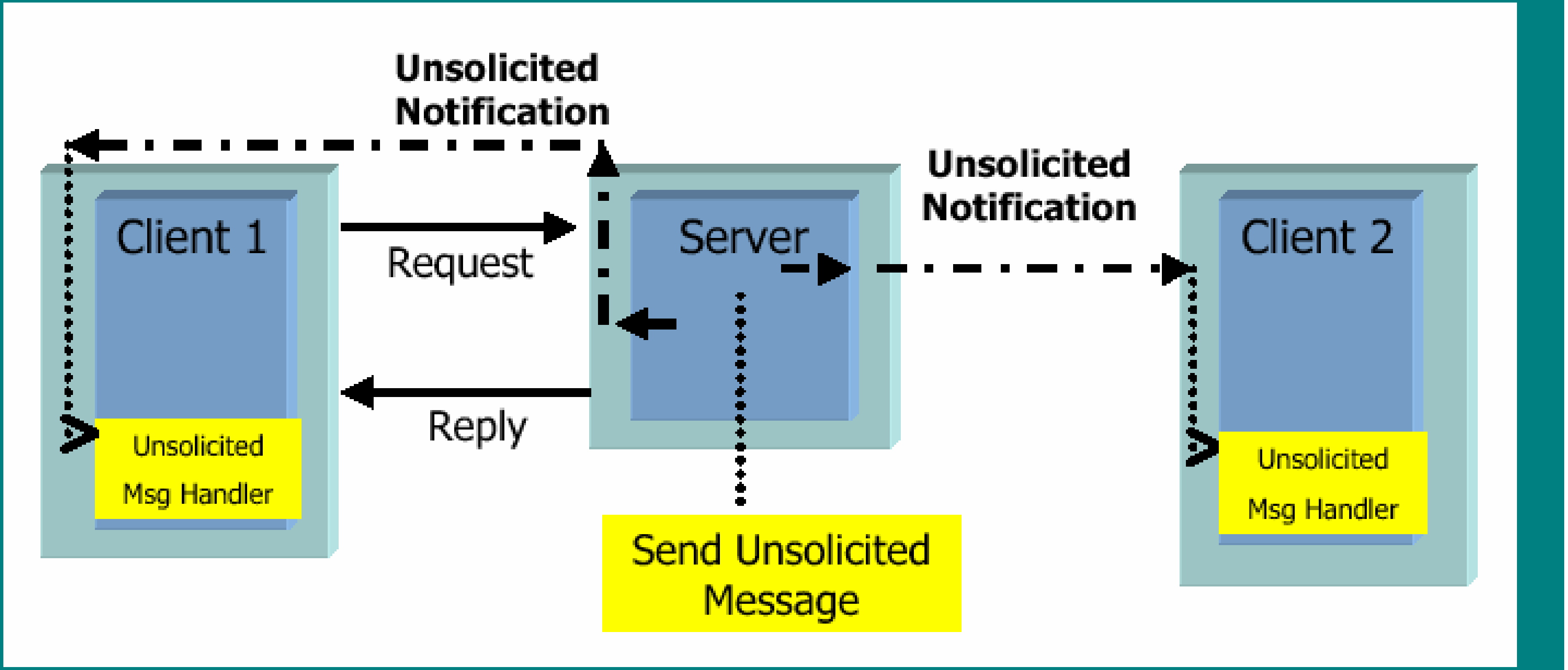
- Tested platforms:
  - Linux (x86, IA64) POSIX
  - Windows (x86, IA64) SDK

Object Request Broker Paradigm

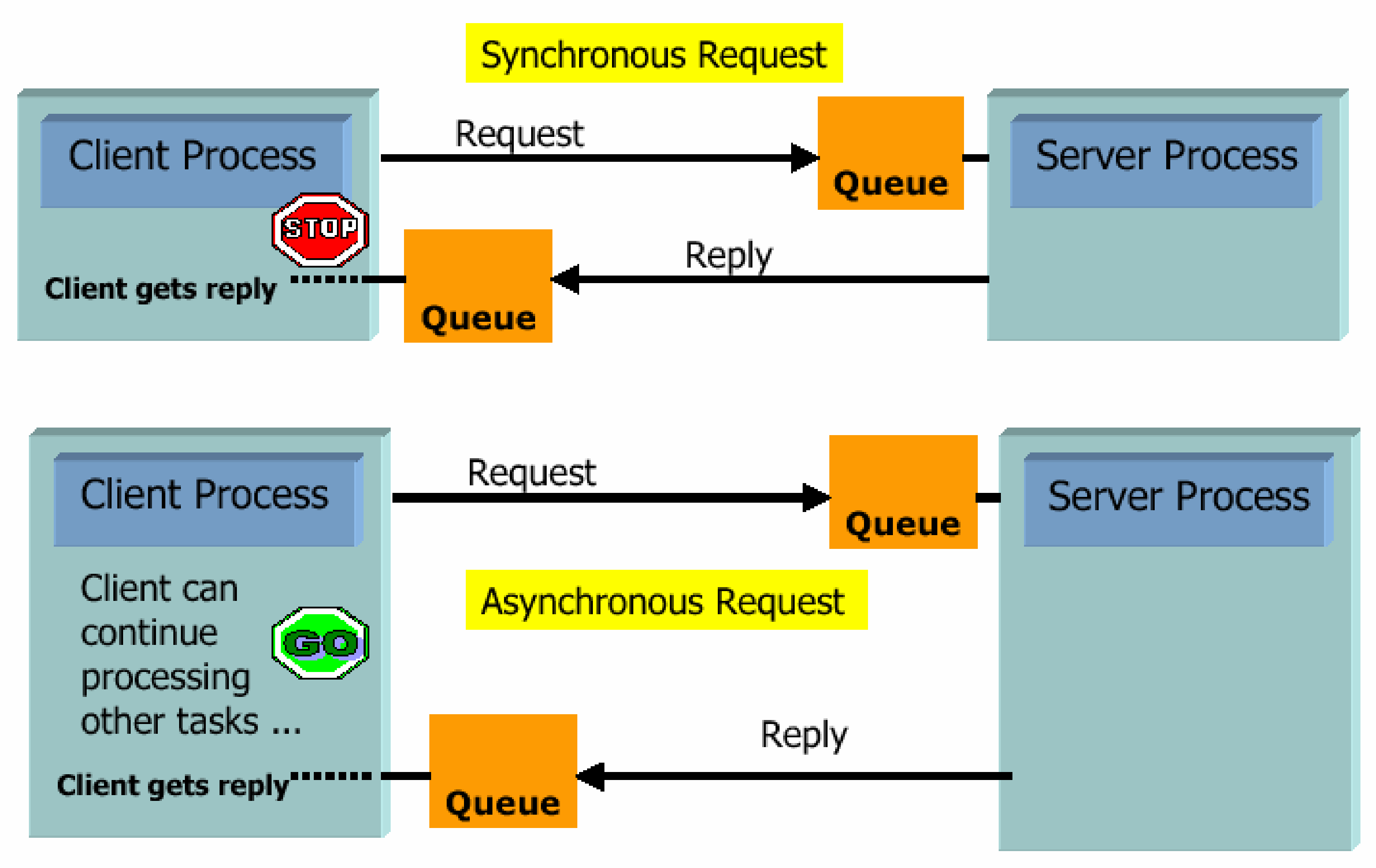


Client Component
Server Component
An operation invoked on the object reference here...
...is executed by the server object here.
ORB
An object reference, defined in the stub code, “marshals” the invocation and passes it to the ORB.
A server-side object reference, defined in the skeleton code, “unmarshals” the invocation and passes it to the server object.

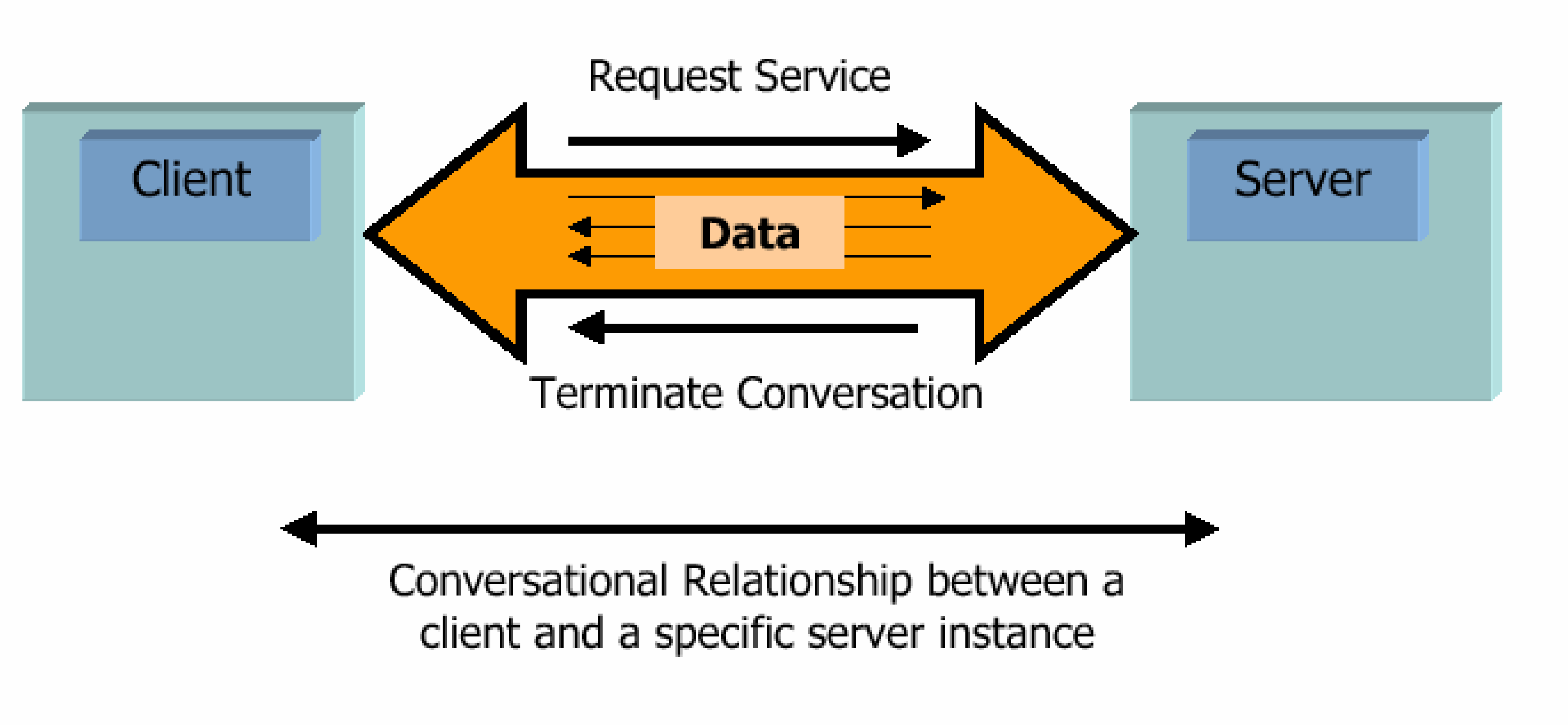
Unsolicited Messaging Paradigm



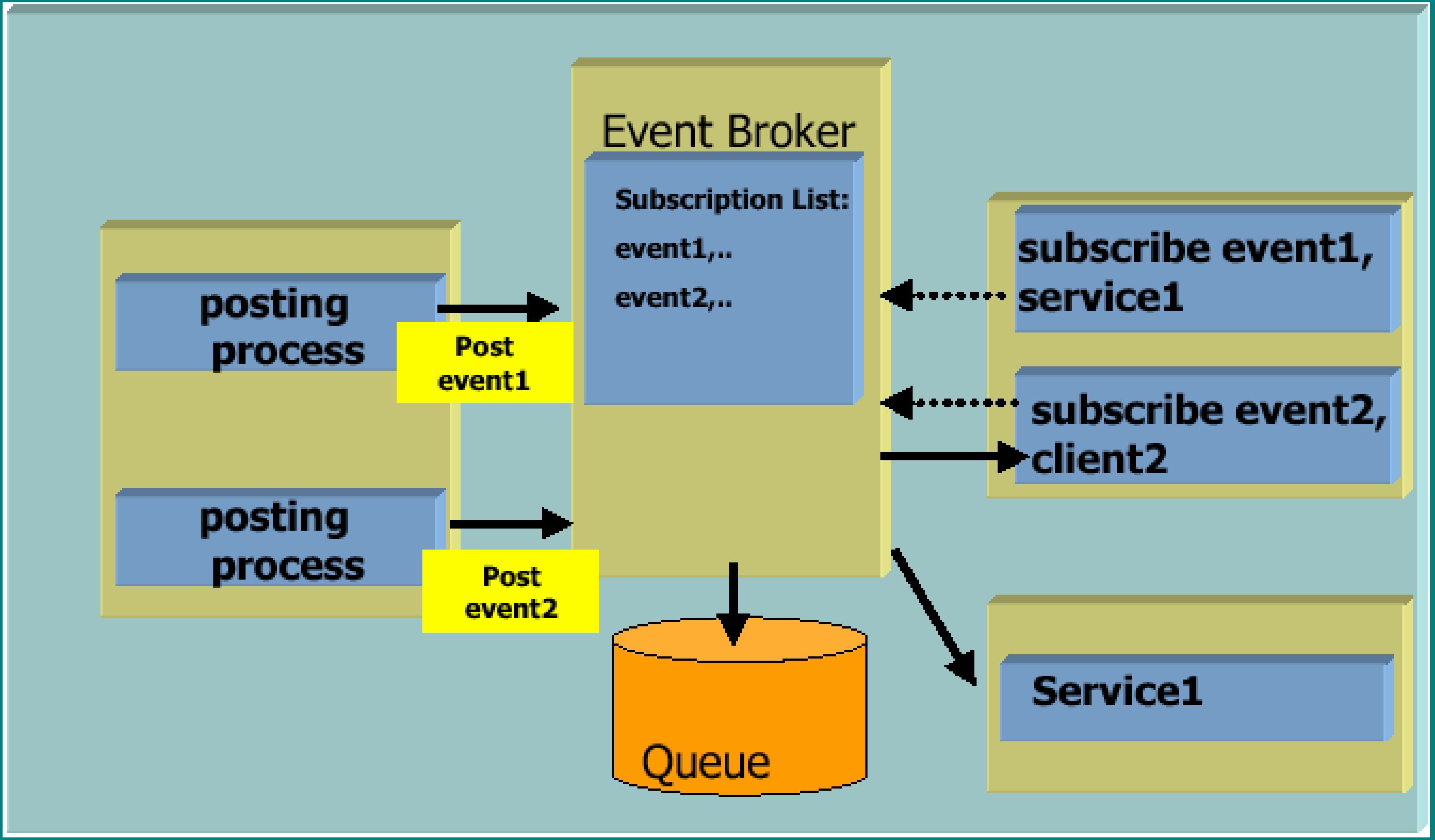
Request/Reply Paradigm



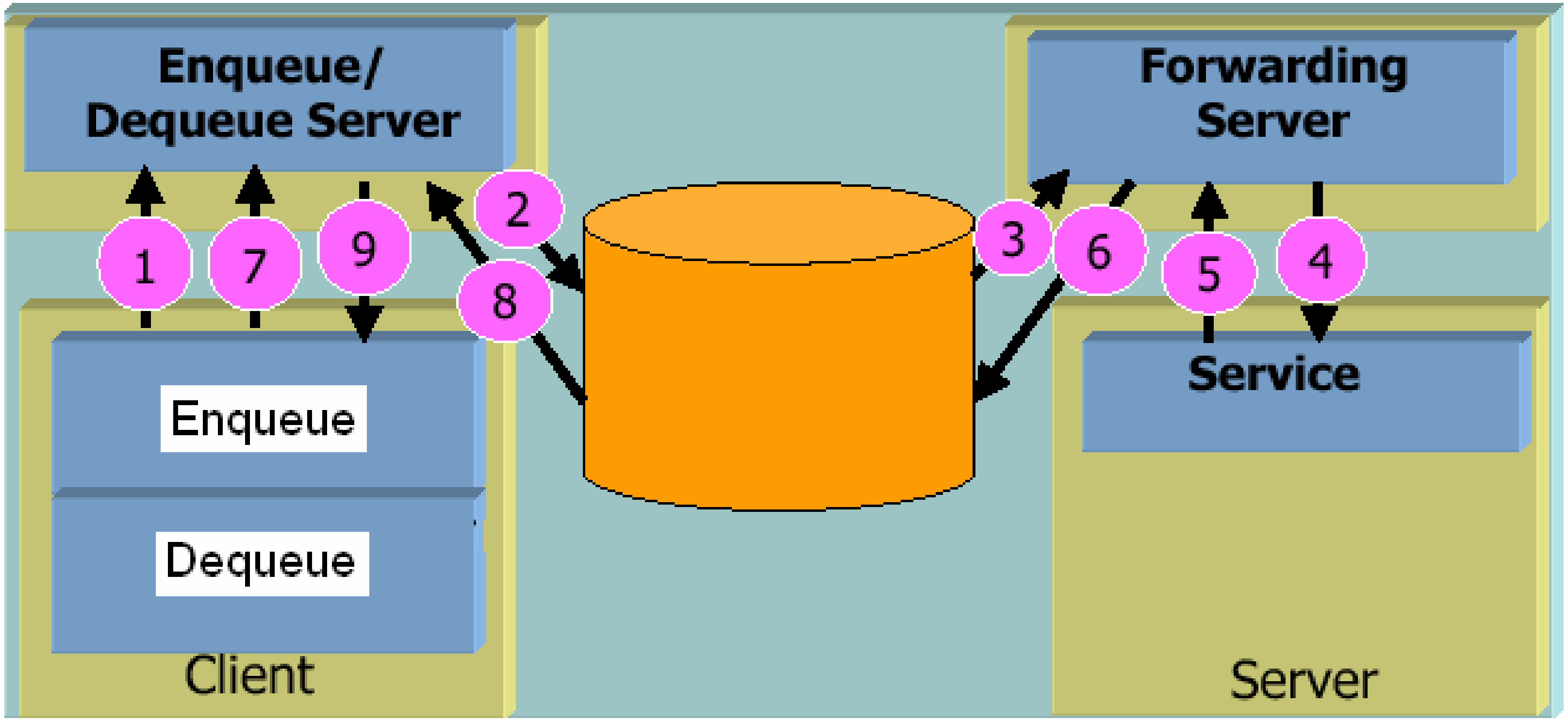
Conversation Paradigm


Request Service
Client
Data
Server
Terminate Conversation
Conversational Relationship between a client and a specific server instance

Publish & Subscribe Paradigm



Store & Forward paradigm

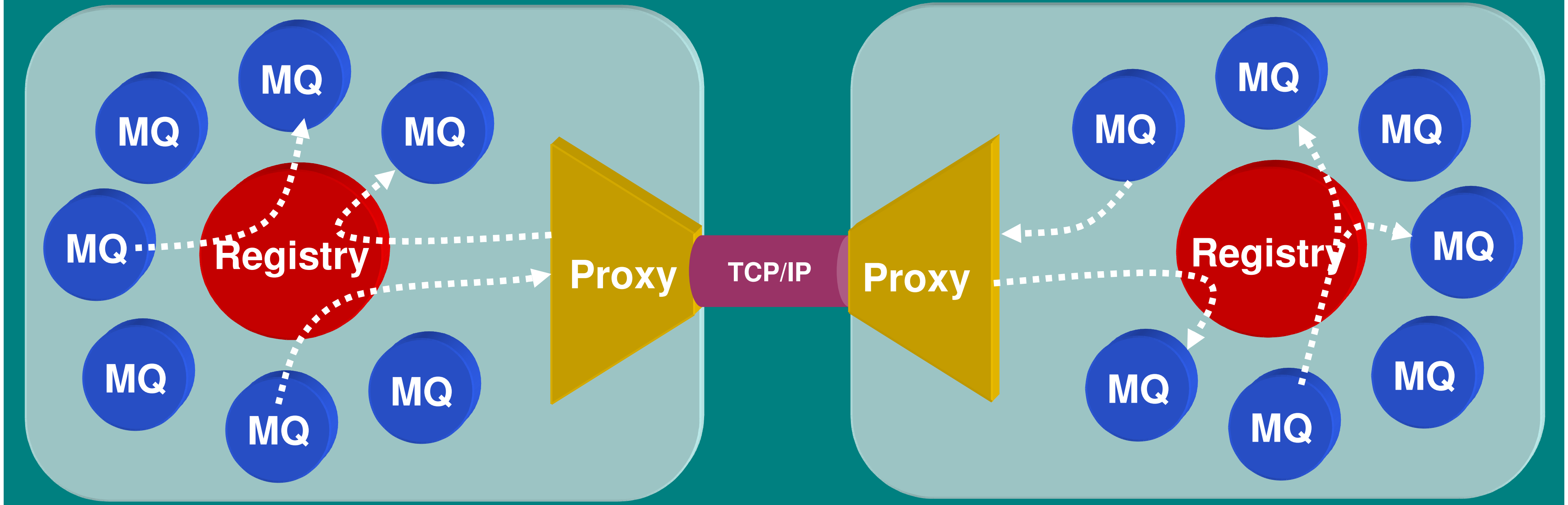


1-Client Requests Enqueue
2-E/D Server Writes Request
3-Fwd Server reads Request

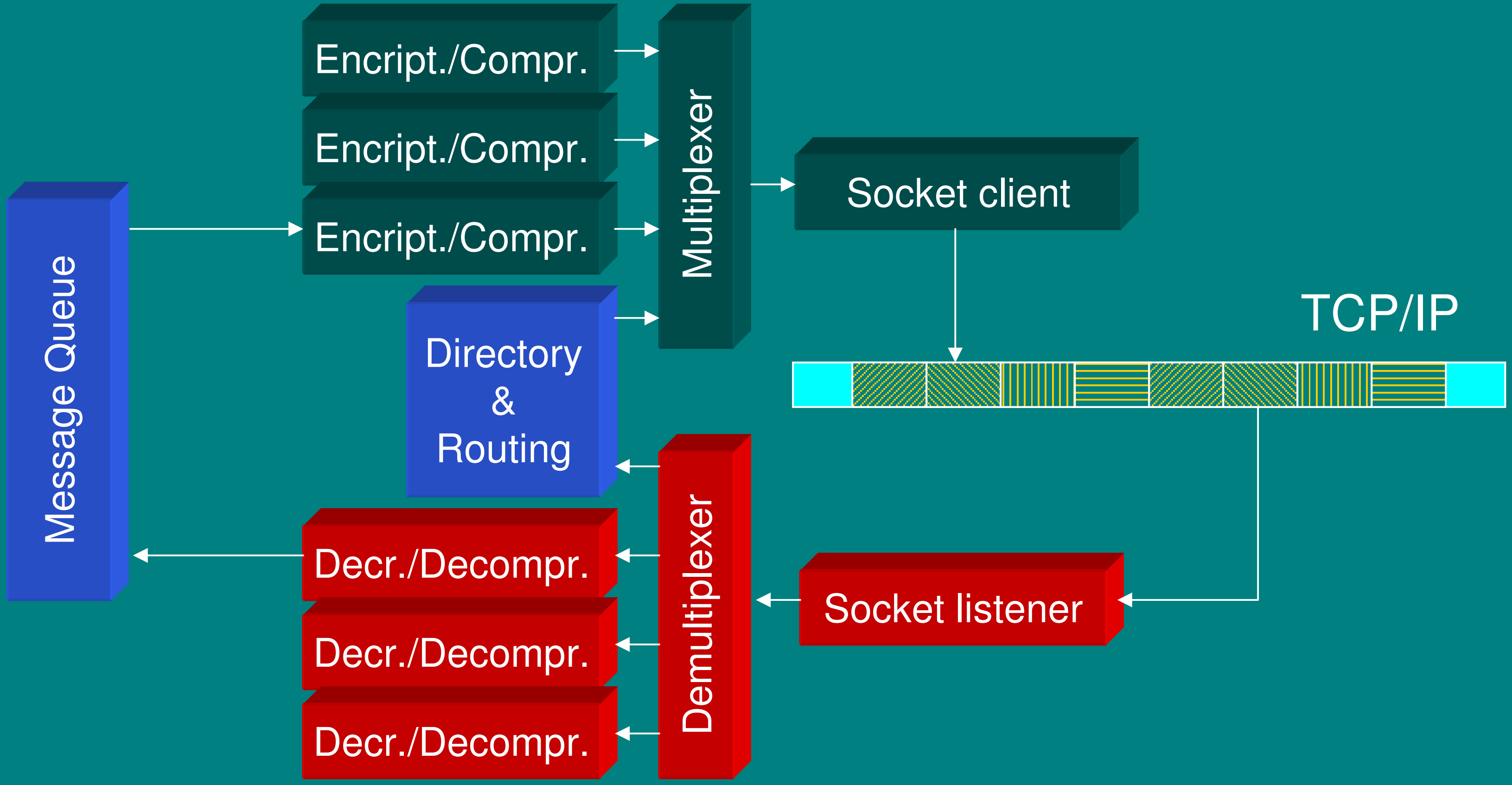
4-Fwd Server Calls Service
5-Service Sends Reply
6-Fwd Server Writes Reply

7-Client Requests Dequeue
8-E/D Server Reads Reply
9-E/D Server Returns Reply

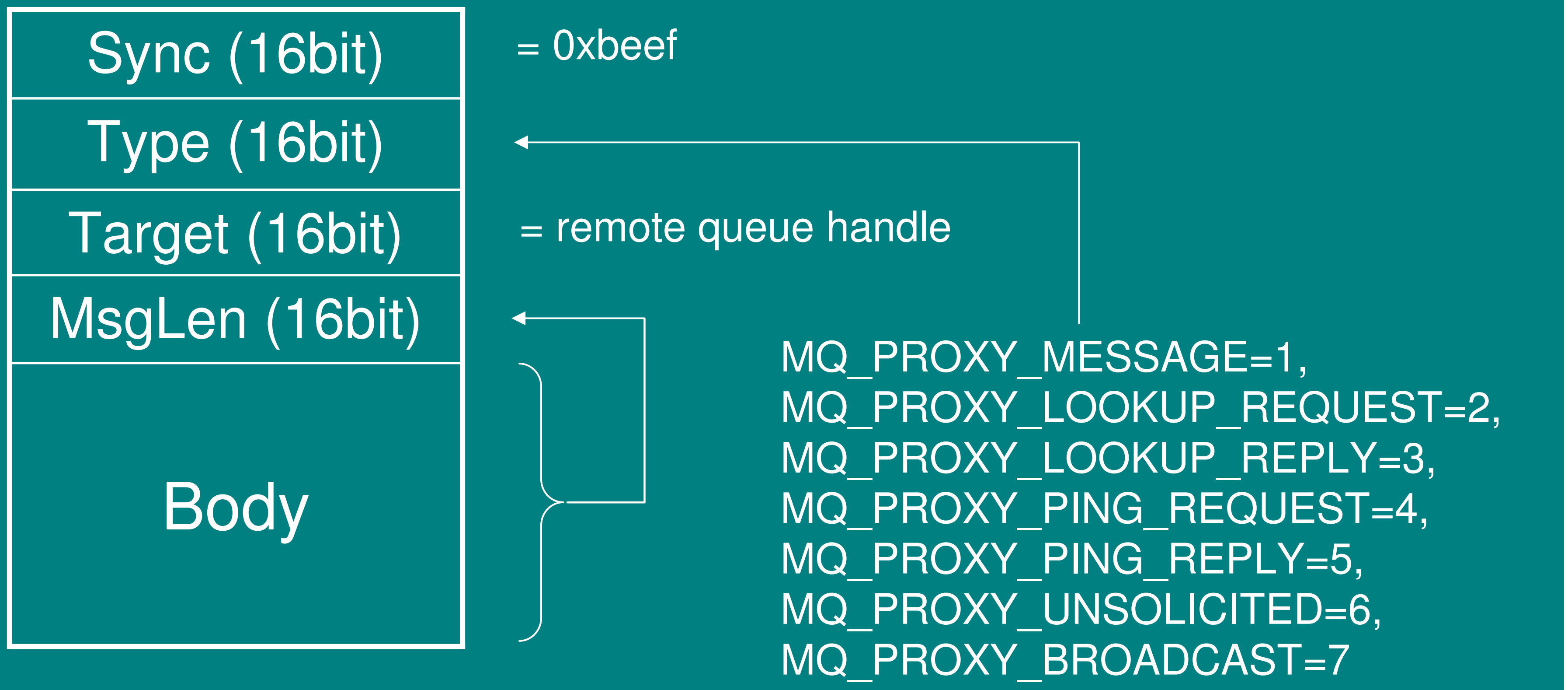
MQ4CPP logical architecture



MQ4CPP networking architecture



MQ4CPP Protocol



Sync (16bit)

Type (16bit)

Target (16bit)

MsgLen (16bit)

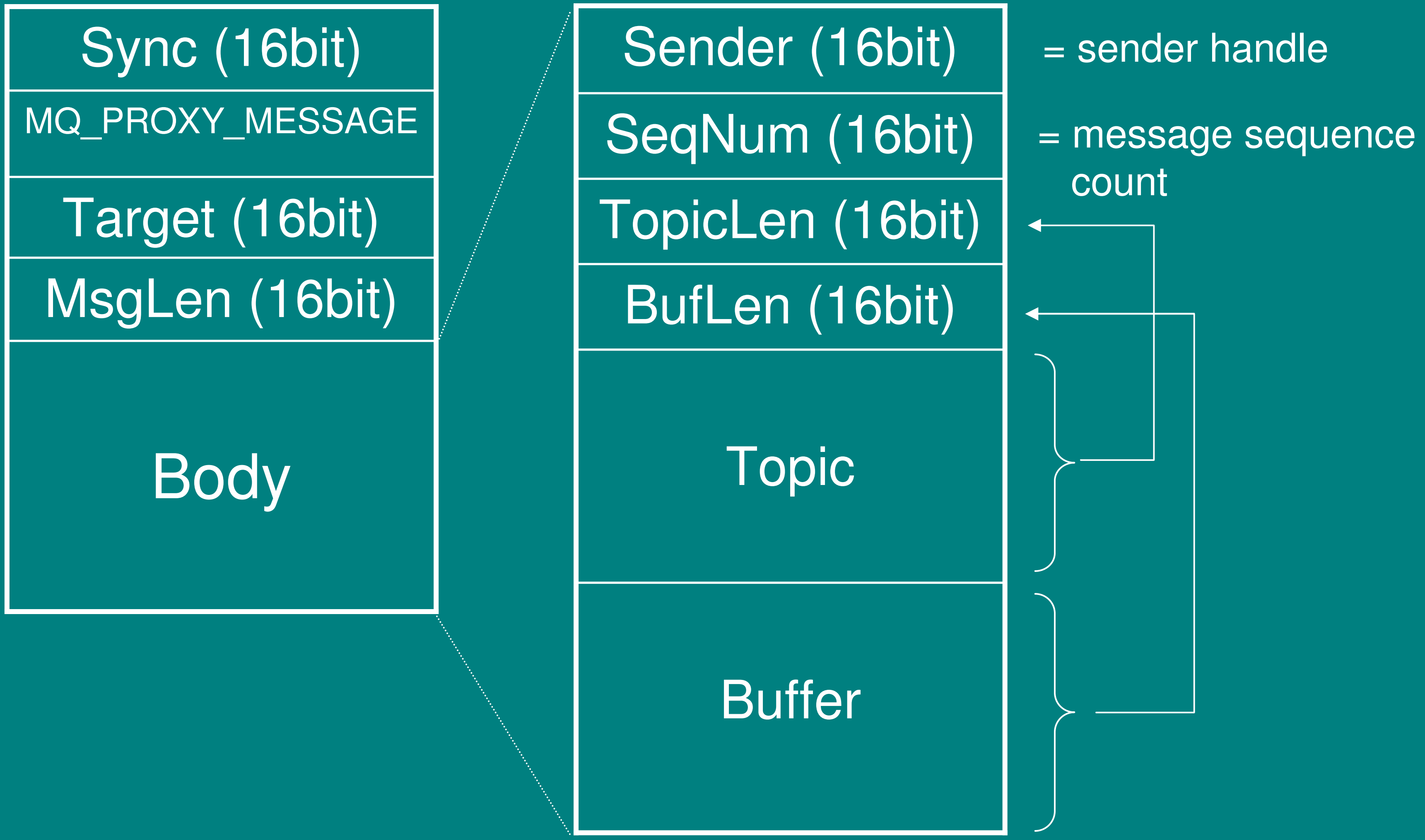
Body

= 0xbeef

= remote queue handle

MQ_PROXY_MESSAGE=1,
MQ_PROXY_LOOKUP_REQUEST=2,
MQ_PROXY_LOOKUP_REPLY=3,
MQ_PROXY_PING_REQUEST=4,
MQ_PROXY_PING_REPLY=5,
MQ_PROXY_UNSOLICITED=6,
MQ_PROXY_BROADCAST=7

Network message structure



Sync (16bit)

MQ_PROXY_MESSAGE

Target (16bit)

MsgLen (16bit)

Body

Sender (16bit) = sender handle

SeqNum (16bit) = message sequence count

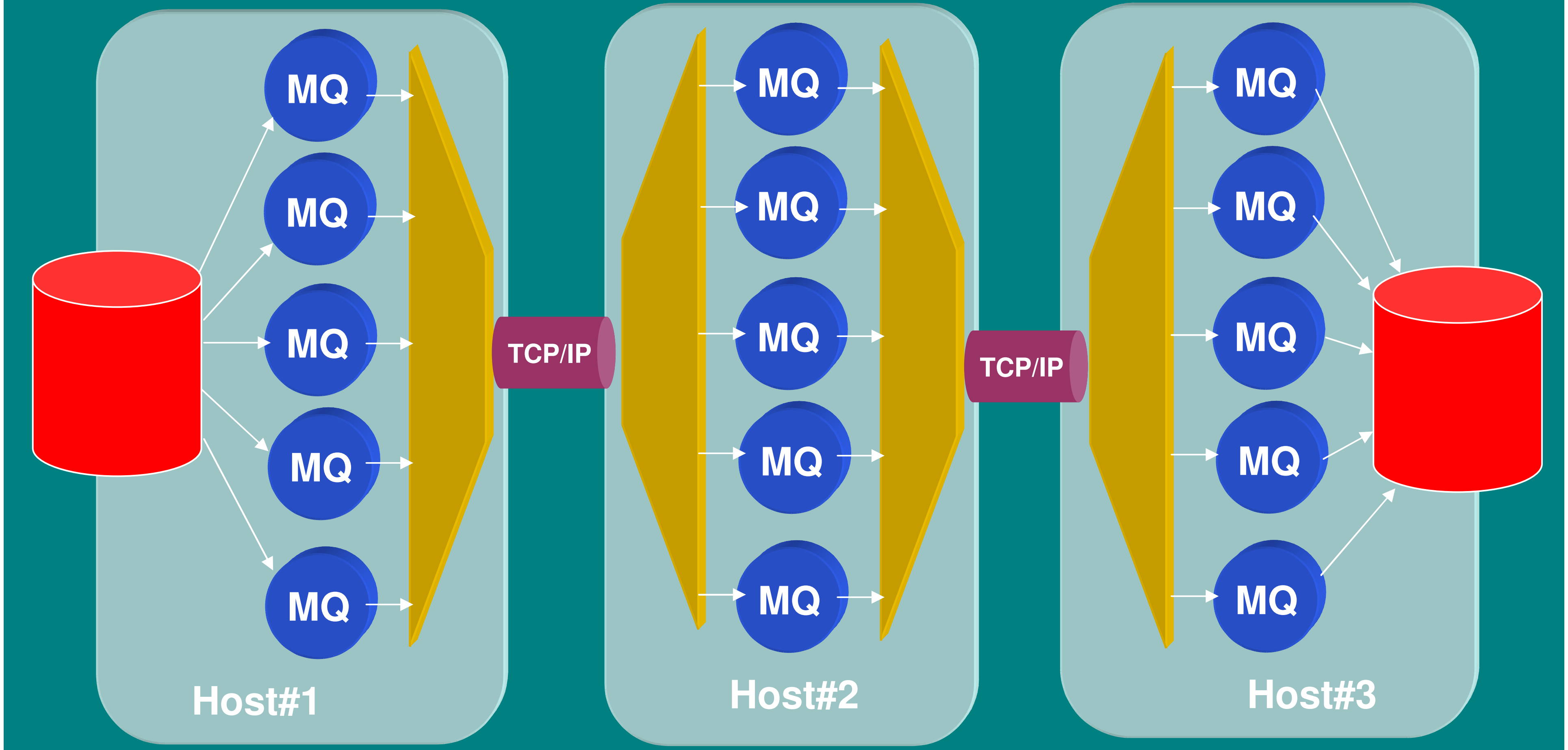
TopicLen (16bit)

BufLen (16bit)

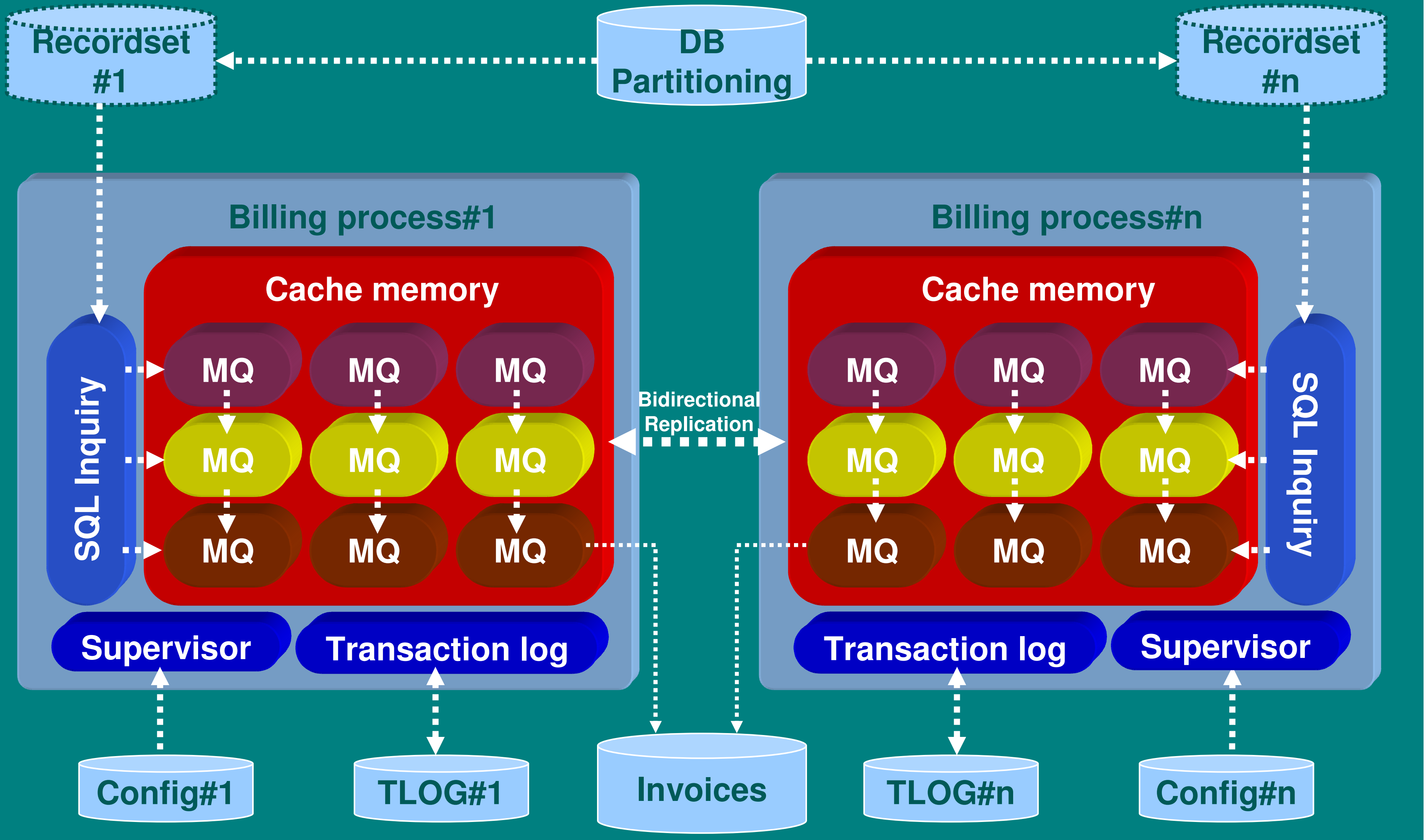
Topic

Buffer

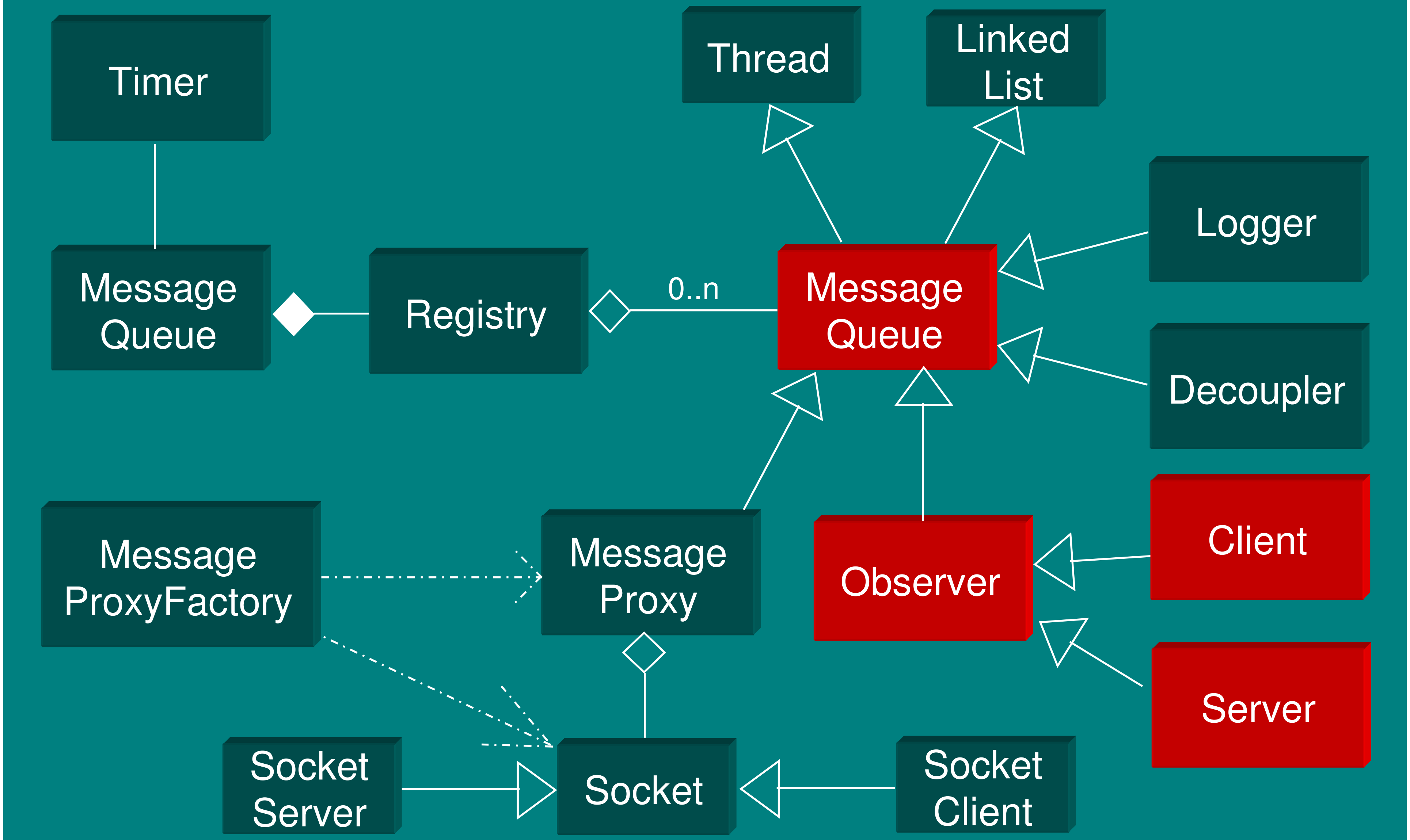
High-perfomance computing model



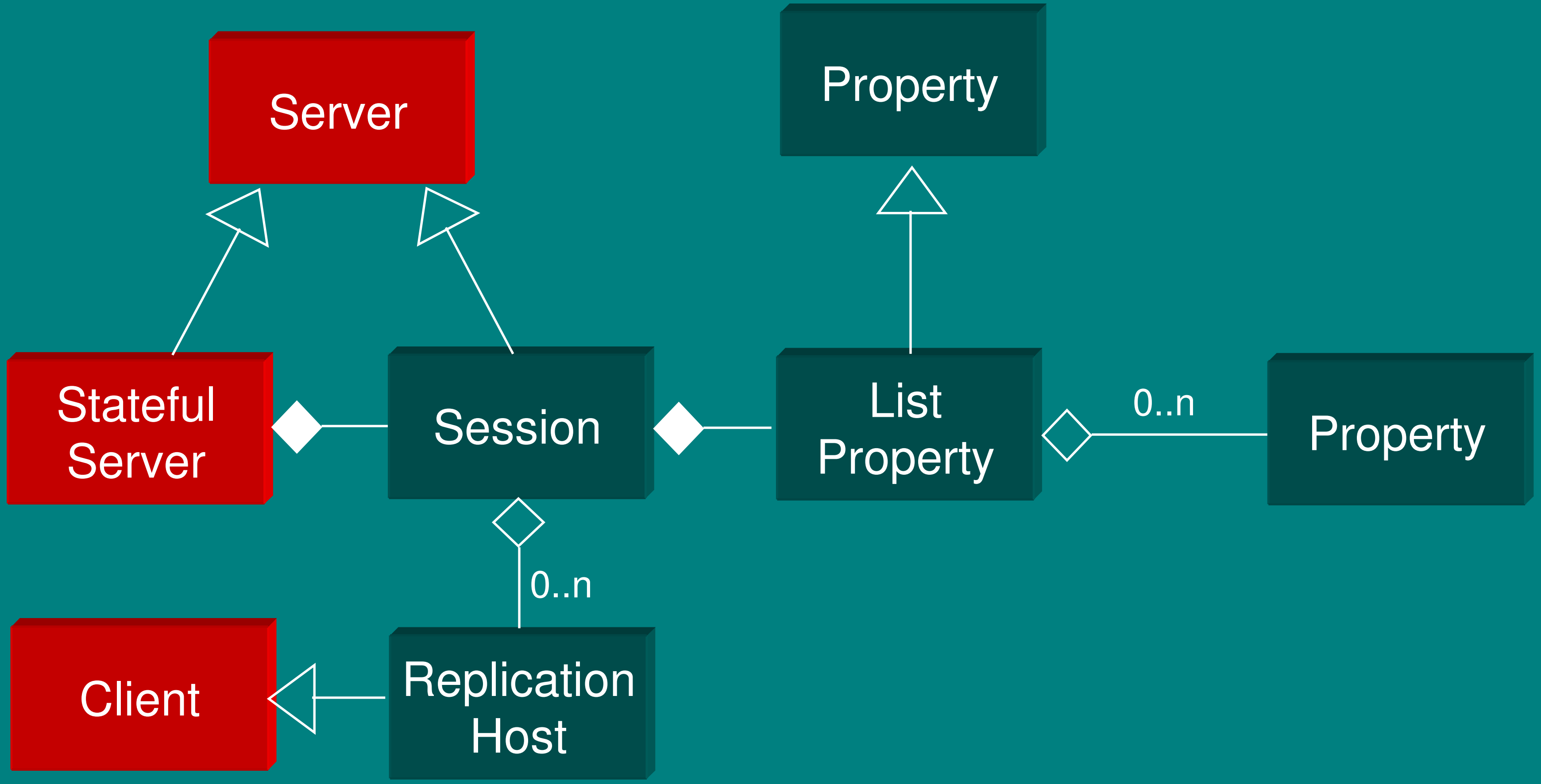
Parallel processing model



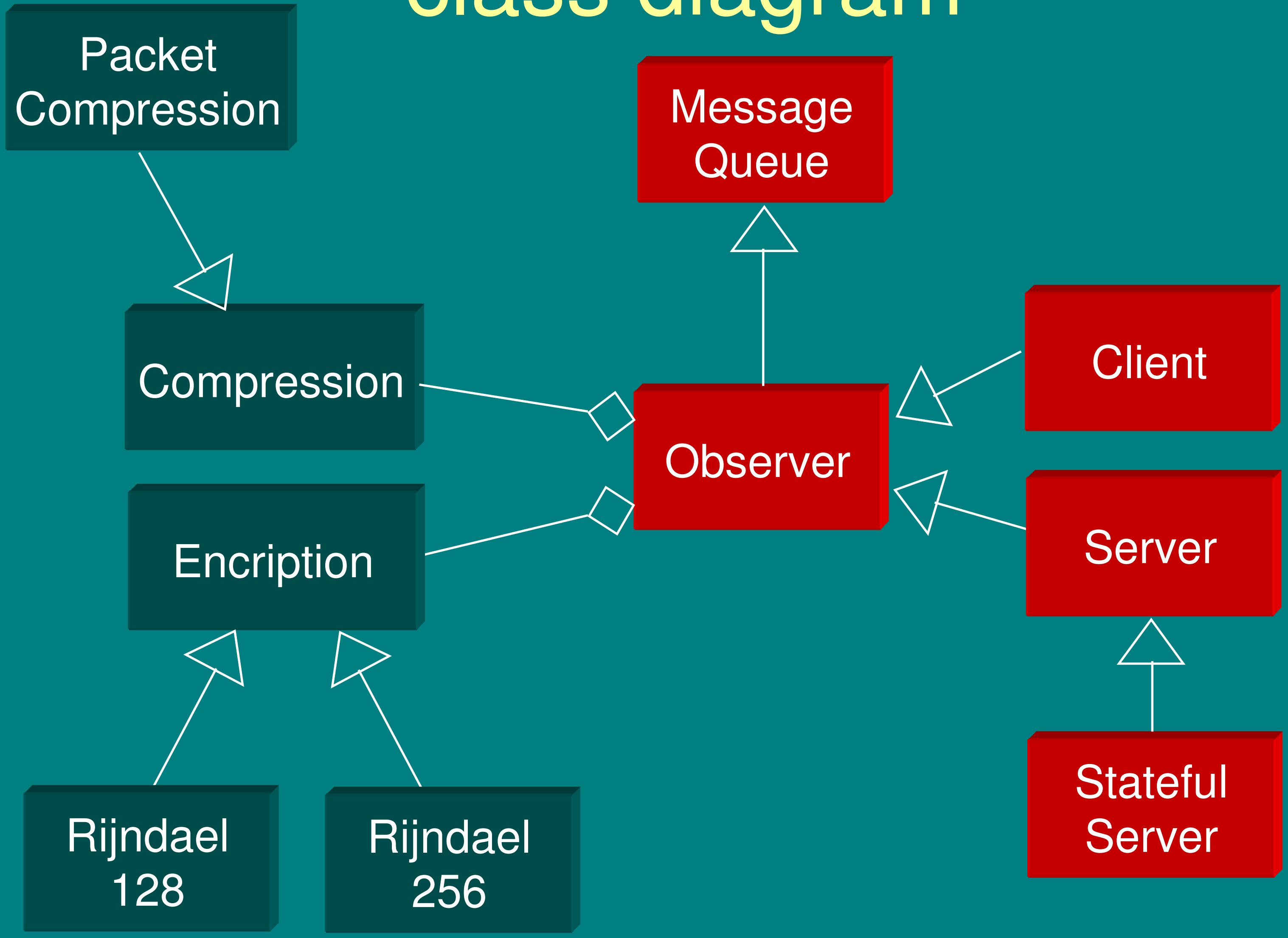
Main class diagram



Session management class diagram



Encription & Compression class diagram



Basic thread skeleton

```
class MyThread : public Thread
{
public:
        MyThread(const char* theName) : Thread(theName)
        {
            start();
            setAffinity(1);
        };

        virtual ~MyThread ()
        {
            stop(false);
        };

protected:
        void run() { … };
};
```

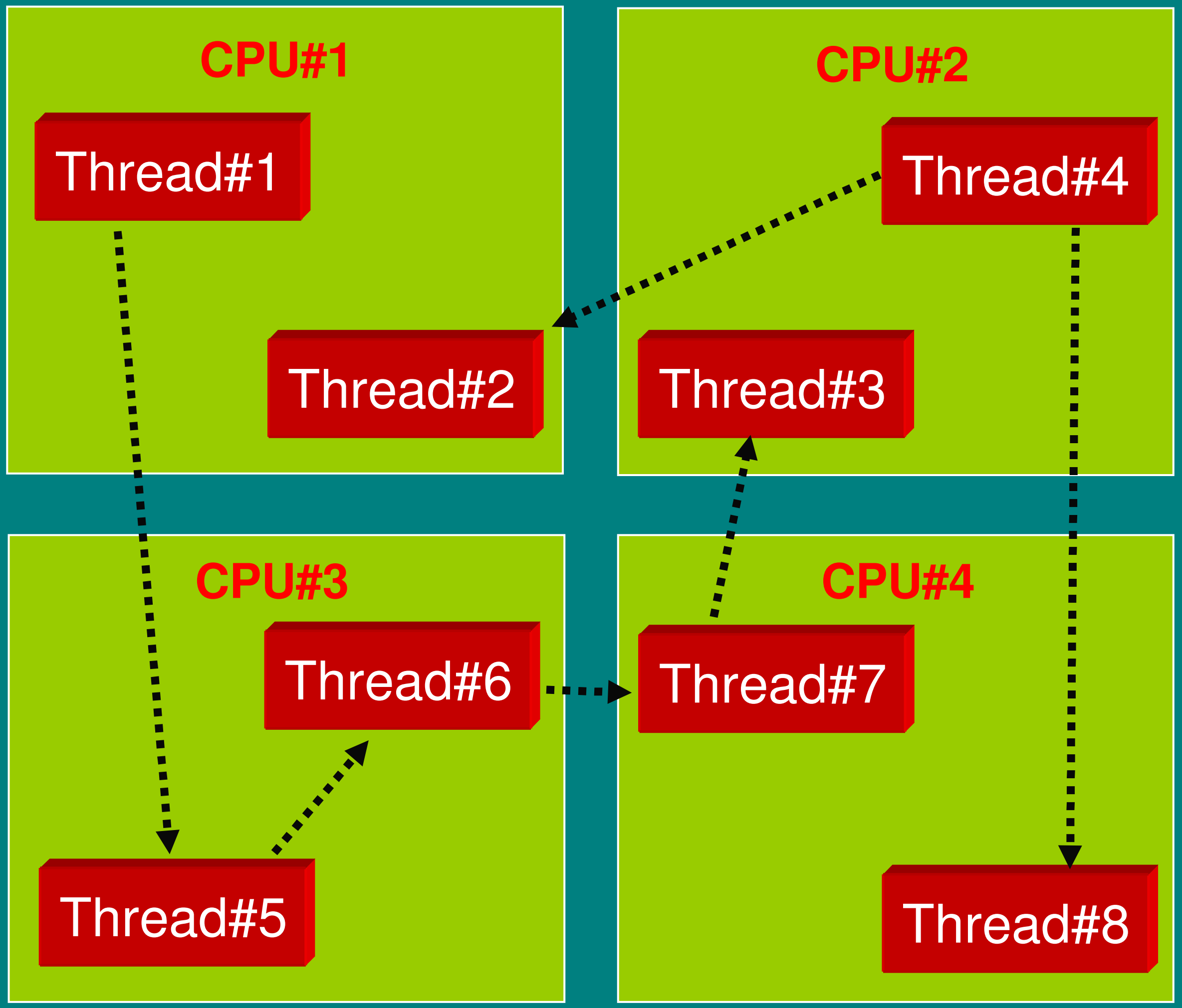
Start this thread

Set cpu affinity for this thread

Stop this thread

Execute this thread

Thread affinity



CPU#1

Thread#1

Thread#2

CPU#2

Thread#4

Thread#3

CPU#3

Thread#6

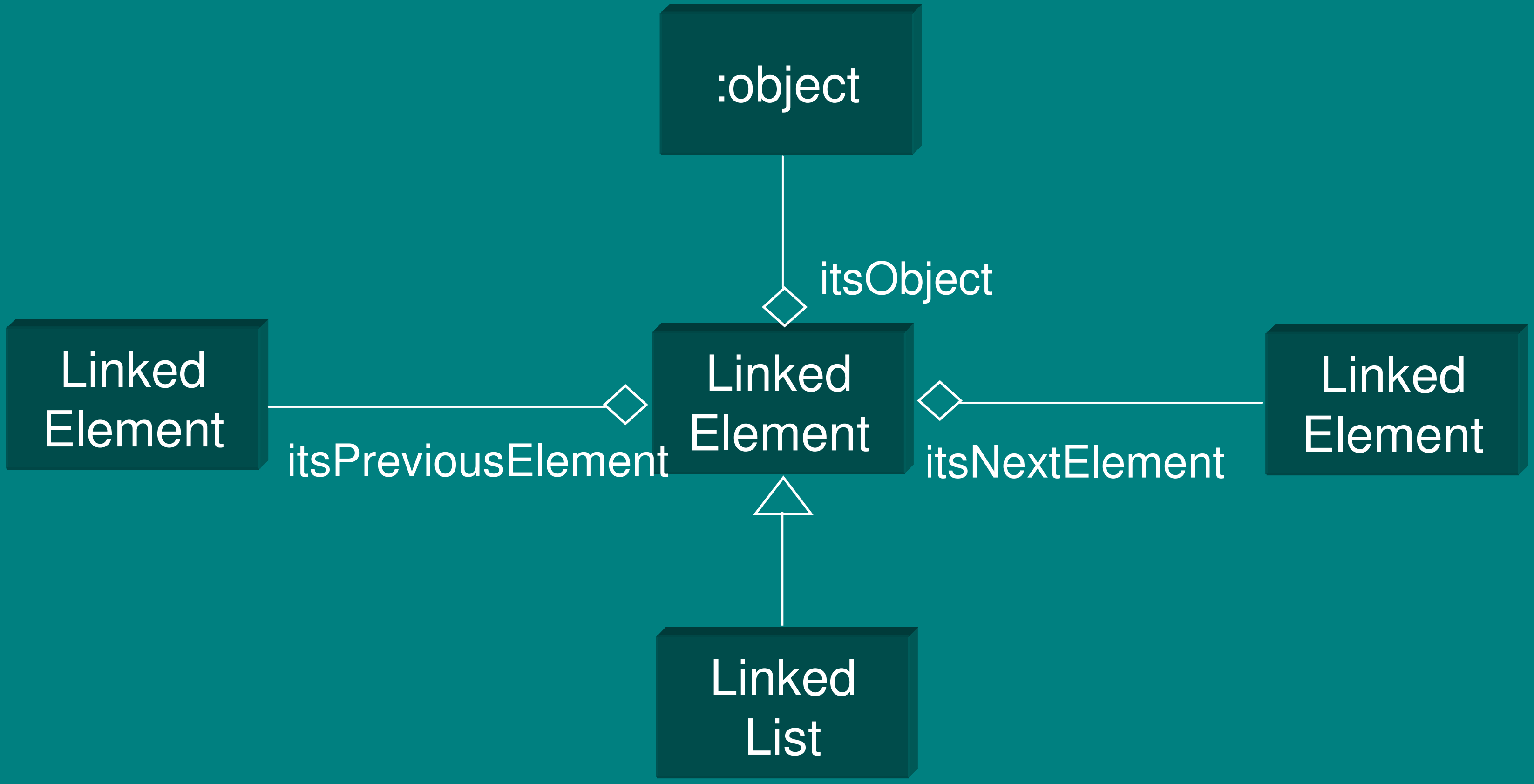
Thread#5

CPU#4

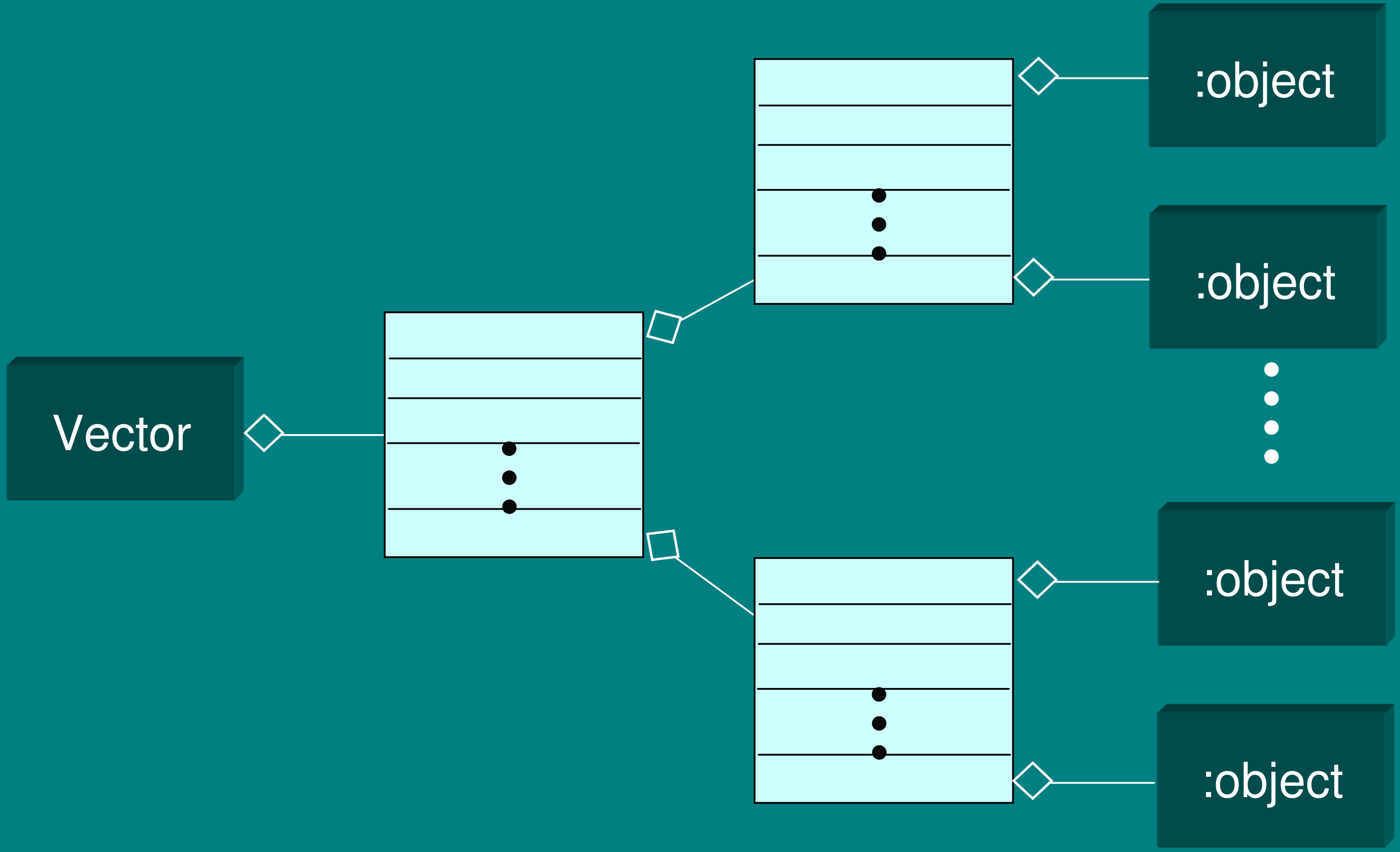
Thread#7

Thread#8

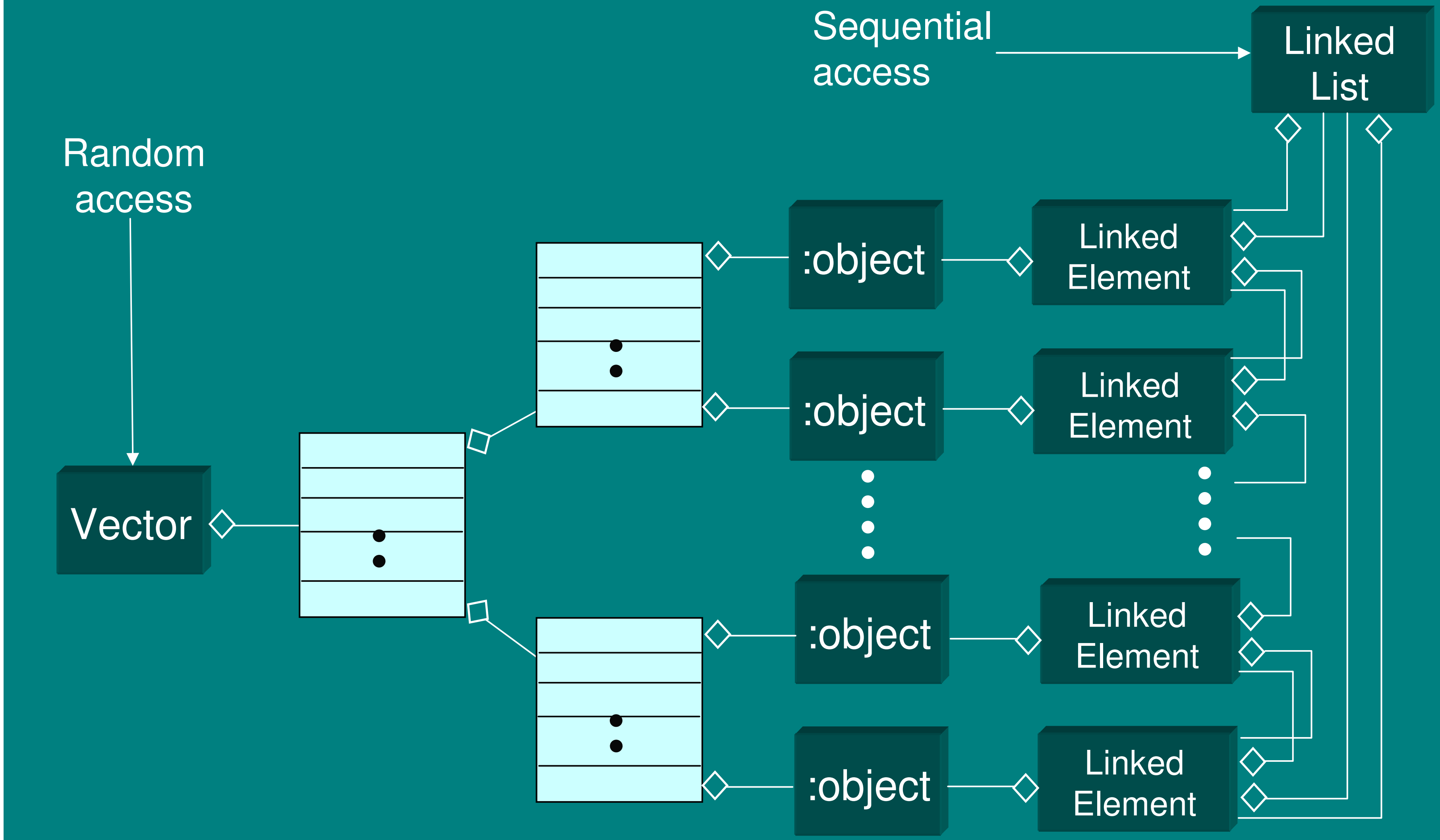
LinkedList


:object
itsObject
Linked Element
itsPreviousElement
Linked Element
itsNextElement
Linked Element
Linked List

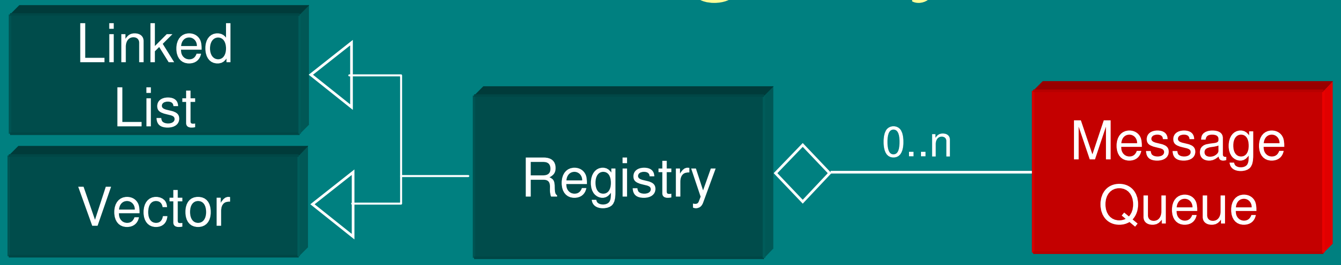
Vector



Vector + LinkedList



Registry



- Registry is the only owner of all message queues
- You can instantiate a message queue and forget to deallocate.
- Registry on shutdown remove by itself all message queues.
- To remove a message queue do not use ‘delete’.
- Use instead MessageQueue::shutdown() to shutdown gracefully the thread
- Registry has a garbage collector process to detect and remove all stopped message queues.

Startup & Shutdown procedure

```
STARTLOGGER(“messages.log”)
LocalhostRouter* aRouter=new LocalhostRouter();


….


Thread::shutdownInProgress();
STOPLOGGER()
STOPTIMER()
STOPREGISTRY()
```

Tracing

- #include “Trace.h” to use tracing
- TRACE displays debug messages on stdout or Microsoft WinDbg debugger
- TRACE(string) displays a string
- TRACE(“Value=“ << aValue) to display values
- DUMP(description, buffer, length) displays a buffer in ASCII and hexadecimal formats
- ‘#define SILENT’ before ‘#include Trace.h’ disables trace messages
- DISPLAY(string) display a string but it cannot be disabled by SILENT definition

Logging

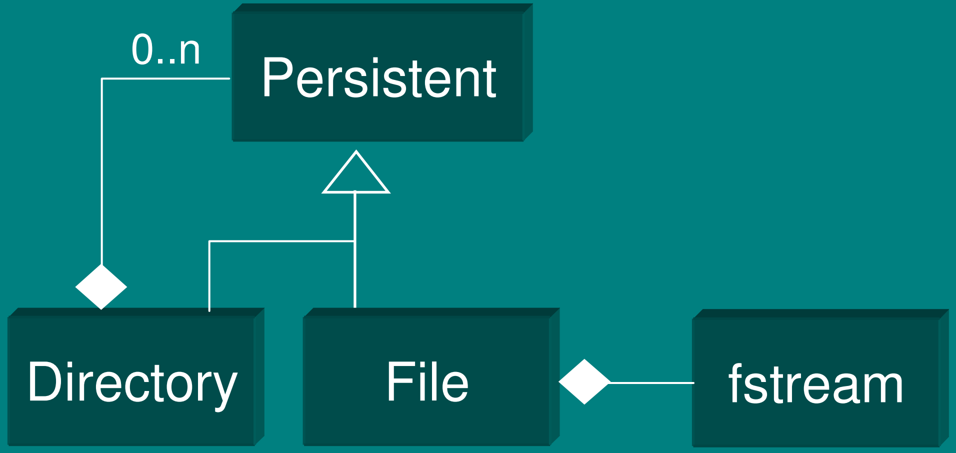
- MQ4CPP include a thread safe logger
- #include “Logger.h” to use logging
- STARTLOGGER(filename) starts the logger thread using the specified filename
- STOPLOGGER stops and flush the logger thread
- BUFFER(address,length) logs a buffer
- LOG(string) logs a string marked as [INFO]
- WARNING(string) logs a string marked as [WARN]
- CRITICAL(string) logs a string marked as [CRIT]
- DEBUG(string) logs a string marked as [DEBG]

Timer

- #include “Logger.h” to use timers
- SCHEDULE(queue, time) schedules a self-repetitive timer for the specified queue
- STOPTIMER stop the Timer thread on shutdown
- TIMEPOINT mark a start point to compute an elapse time
- TRACE_ELAPSE displays the elapsed time (it can be disabled using ‘#define SILENT’)
- DISPLAY_ELAPSE displays the elapsed time

File system

- #include “FileSystem.h” to use file system helper classes
- Persistent objects can represent a File or a Directory
- File class handles the access to a single file
- Directory class handles the access to a single directory
- Directory is the only owner of all Persistent objects created during a search in the directory itself.



Hashing

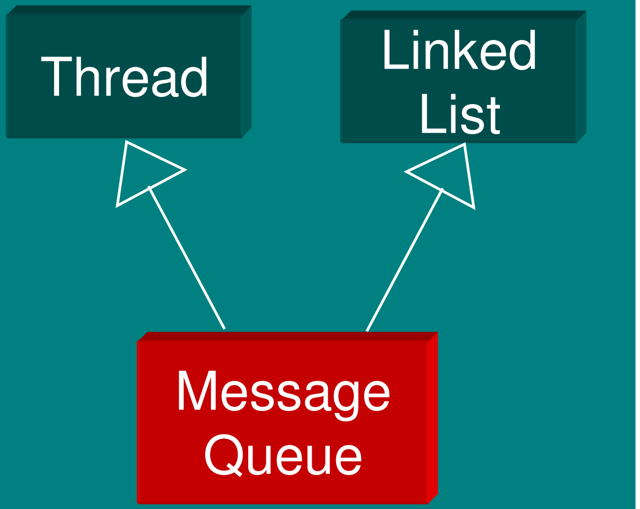
- MQ4CPP include a set of hashing algorithms.
- #include “GeneralHashingFunction.h” to use hashing functions
- **RS Hash Function**: a simple hash function from Robert Sedgwicks Algorithms in C book.
- **JS Hash Function**: a bitwise hash function written by Justin Sobel
- **PJW Hash Function**: this hash algorithm is based on work by Peter J. Weinberger of AT&T Bell Labs.
- **ELF Hash Function**: similar to the PJW Hash function, but tweaked for 32-bit processors. Its the hash function widely used on most UNIX systems.
- **BKDR Hash Function**: this hash function comes from Brian Kernighan and Dennis Ritchie's book "The C Programming Language". It is a simple hash function using a strange set of possible seeds which all constitute a pattern of 31....31...31 etc, it seems to be very similar to the DJB hash function.
- **SDBM Hash Function**: this is the algorithm of choice which is used in the open source SDBM project. The hash function seems to have a good over-all distribution for many different data sets. It seems to work well in situations where there is a high variance in the MSBs of the elements in a data set.
- **DJB Hash Function**: an algorithm produced by Daniel J. Bernstein and shown first to the world on the comp.lang.c newsgroup. Its efficient as far as processing is concerned.
- **DEK Hash Function**: an algorithm proposed by Donald E. Knuth in The Art Of Computer Programming Volume 3, under the topic of sorting and search chapter 6.4.
- **AP Hash Function**: an algorithm produced by Arash Partow. It is based on a hybrid rotative and additive hash function algorithm based around four primes 3,5,7 and 11.

Sorting

- MQ4CPP include a merge sort algorithm.
- #include “MergeSort.h” to use it
- It is an algorithmic design paradigm that contains the following steps:
  - Divide: Break the problem into smaller sub-problems
  - Recur: Solve each of the sub-problems recursively
  - Conquer: Combine the solutions of each of the sub-problems to form the solution of the problem
- MergeSort works in this way:
  - If the input sequence has only one element returns
  - Partition the input sequence into two halves
  - Sort the two subsequences using the same algorithm
  - Merge the two sorted subsequences to form the output sequence
- MergeSort uses vectors of pair<T1,T2> where T1 is used to compare
- MergeSort is a f(N * log N) algorithm

Basic MessageQueue skeleton

```
class MyClient : protected MessageQueue
{
protected:

public:
    MyClient()
            : MessageQueue("MyClient")  { …. };

    virtual ~MyClient() { …. };

protected:
    virtual void onMessage(Message* theMessage) { …
};
```



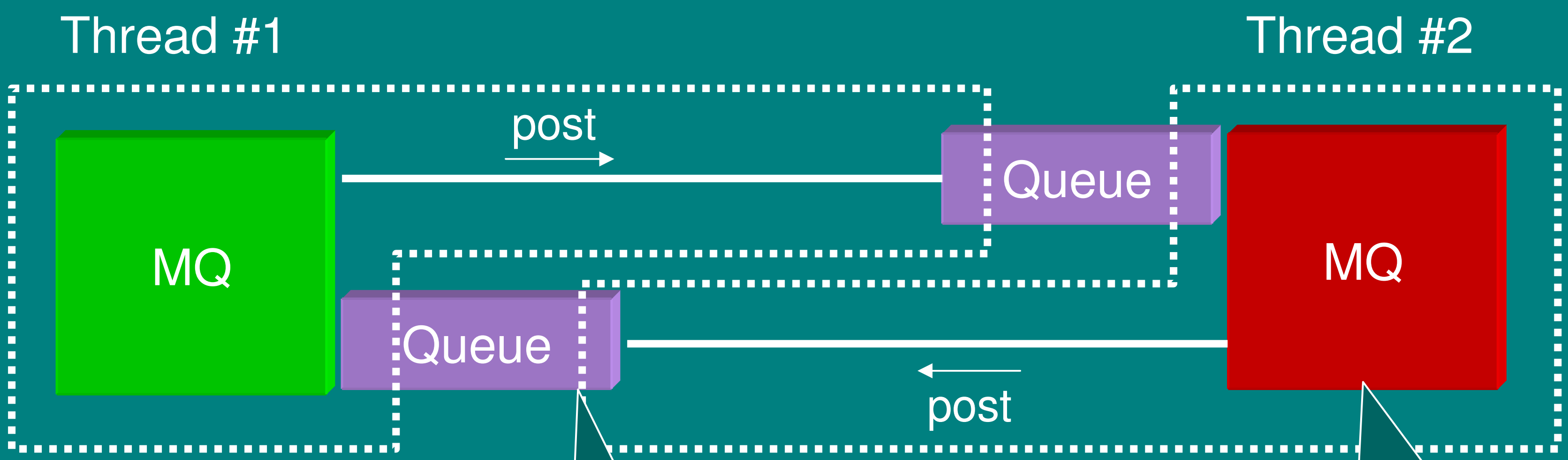
Thread

Linked List

Message Queue

Each thread has a name and a message queue handle assigned by Registry (MQHANDLE) during object creation.

MQ4CPP threads decoupling



Threads are synchronized using:

Thread::wait()

….

Thread::release()

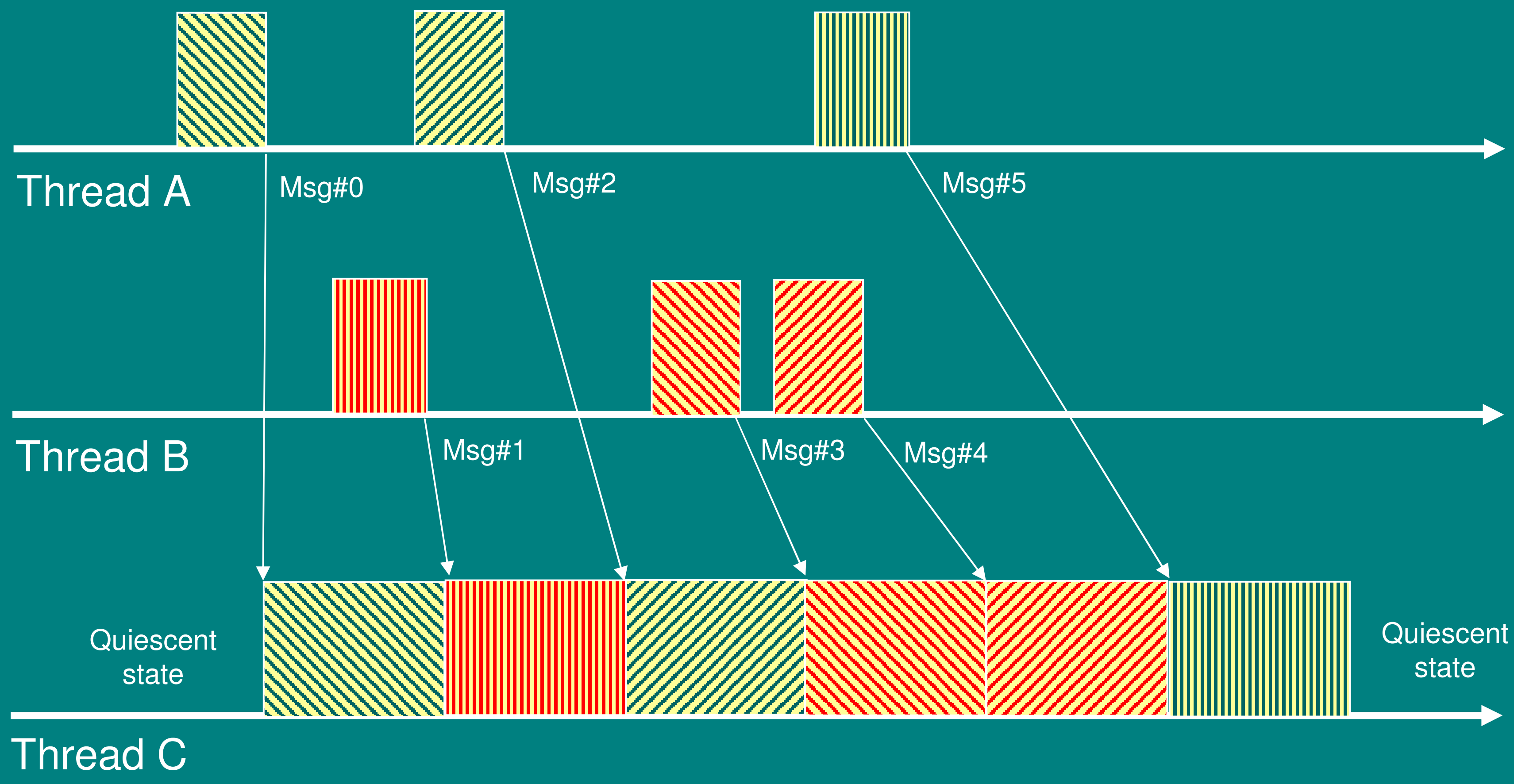
A thread without messages in queue will be suspended, until a new message is posted, using:

Thread::suspend()

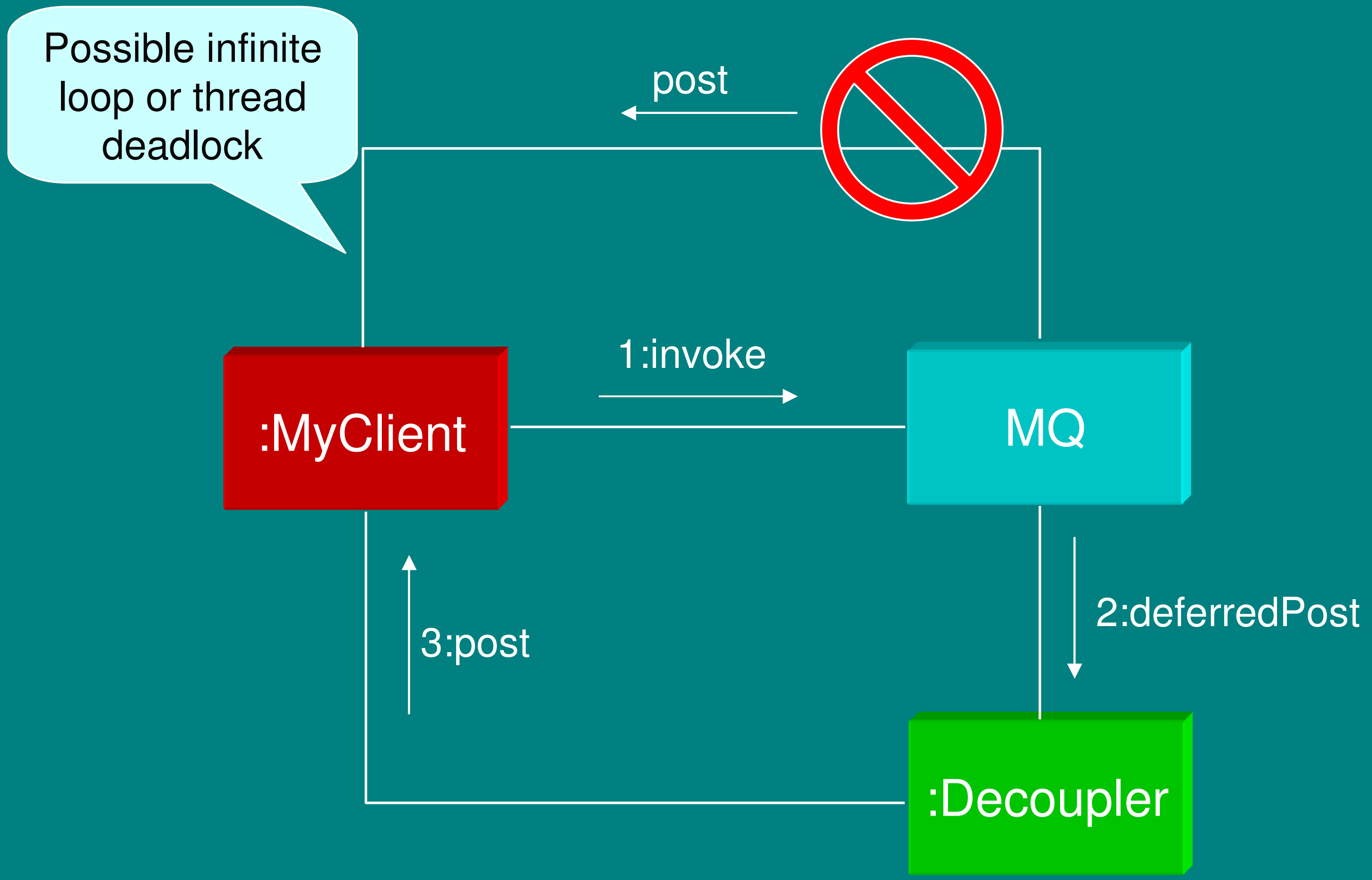
….

Thread::resume()

CPU resources usage

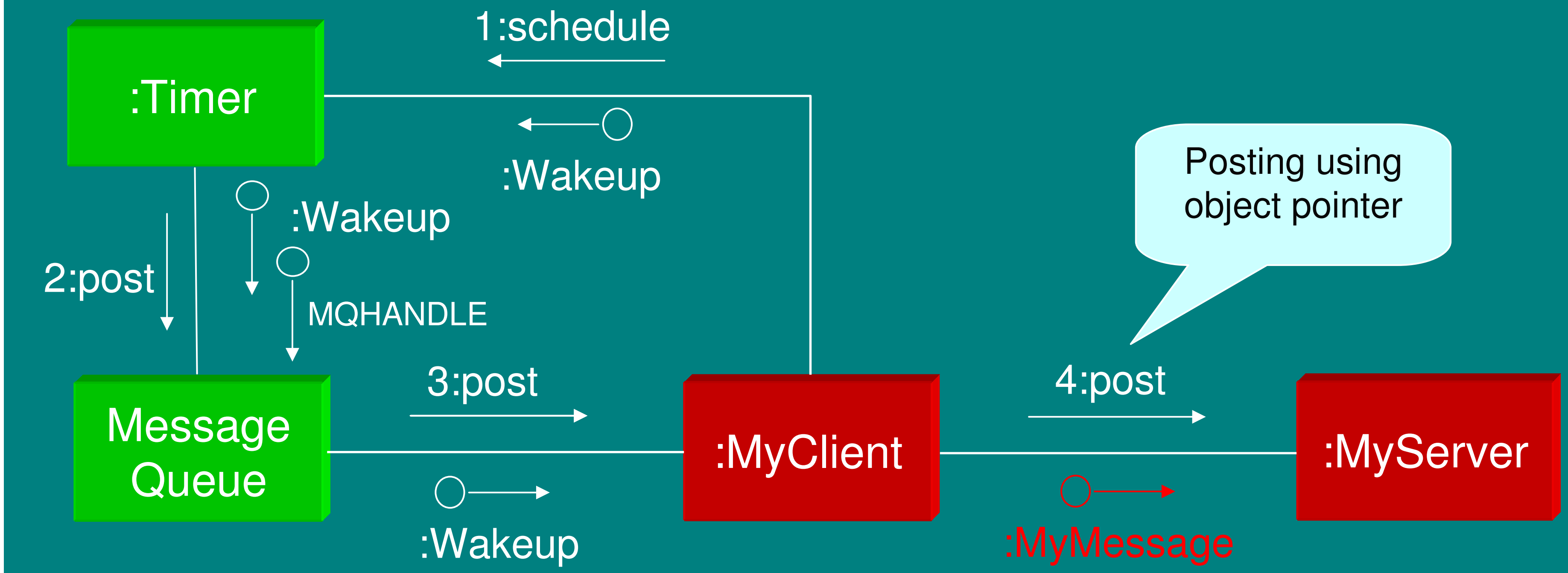


Deferred messaging

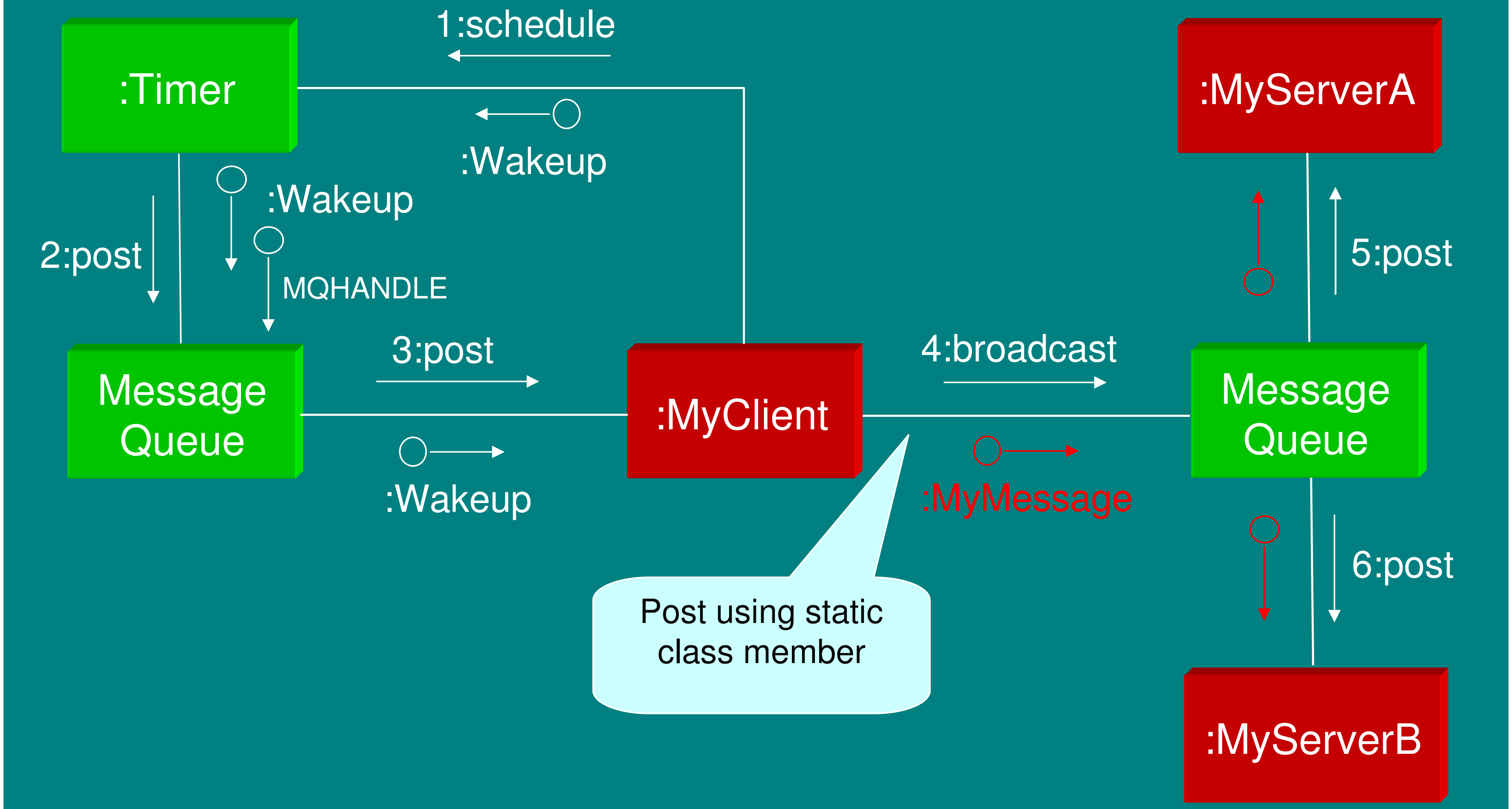


Possible infinite loop or thread deadlock

Direct messaging (example1.cpp)

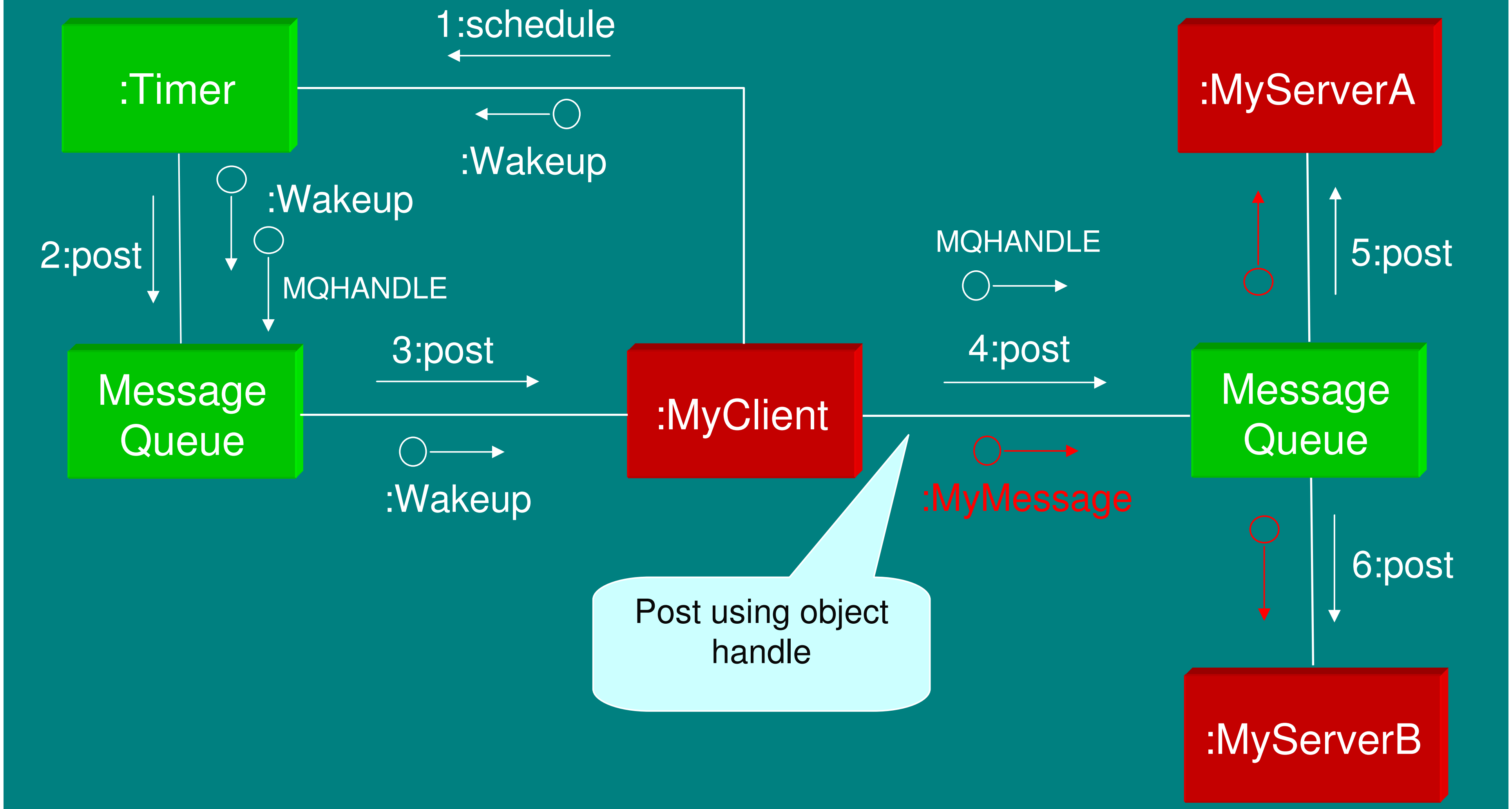


Broadcast (example2.cpp)



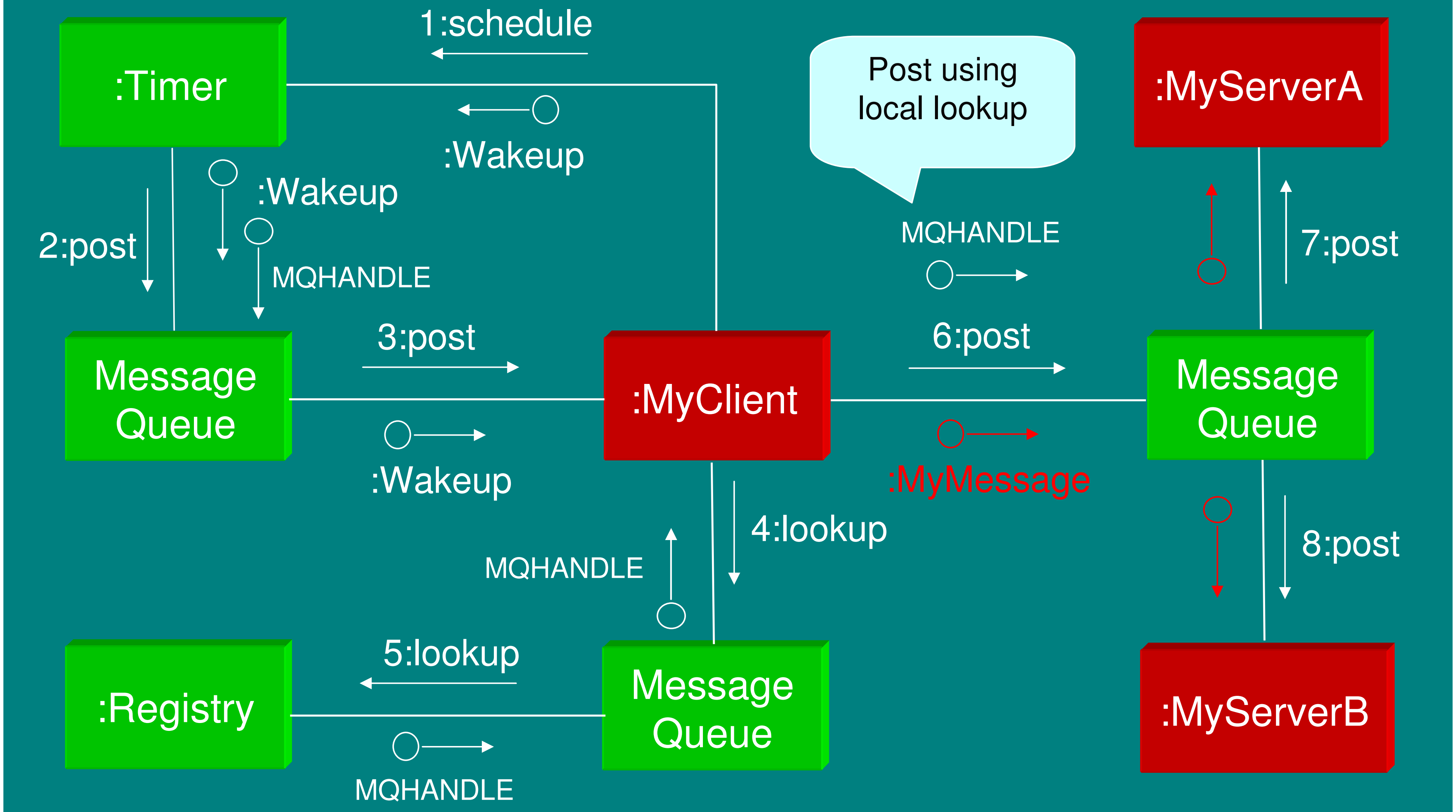
Post using static class member

Indirect messaging (example3.cpp)



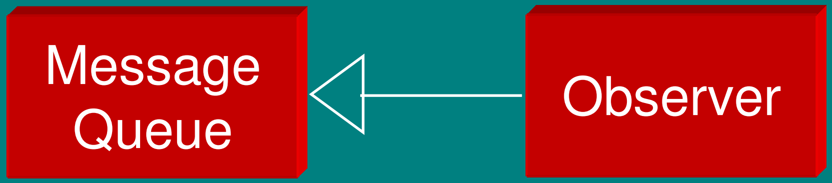
Post using object handle

Local lookup (example4.cpp)



Basic Observer skeleton

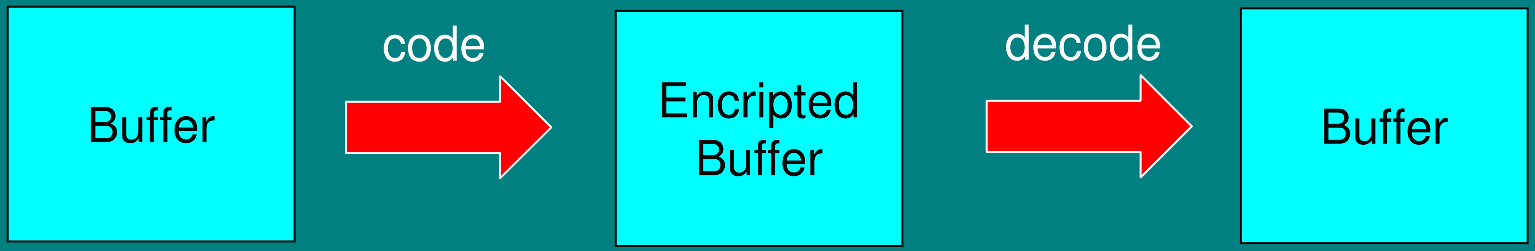
```
class MyClient : public Observer
{
protected:

public:
     MyClient(const char* theName) : Observer(theName) { … };
     virtual ~MyClient() { … };

protected:
     virtual void onWakeup(Wakeup* theMessage) { … };
     virtual void onPing(PingReplyMessage* theMessage) { … };
     virtual void onLookup(LookupReplyMessage* theMessage) { … };
     virtual void onBroadcast(NetworkMessage* theMessage) { … };
     virtual void onUnsolicited(NetworkMessage* theMessage) { … };
     virtual NetworkMessage* onRequest(NetworkMessage* theMessage) { … };
     virtual void onLocal(Message* theMessage) { … };
};
```



Encription

- MQ4CPP include MCRIPT Rijndael 128 and 256 bit encryption algorithms.
- Rijndael is a block ciphers algorithm with:
  - 128 bits of key and 128 bit of block size
  - 256 bits of key and 256 bit of block size
- Encription class include a 128 and 256 key generator
- To use encryption use:

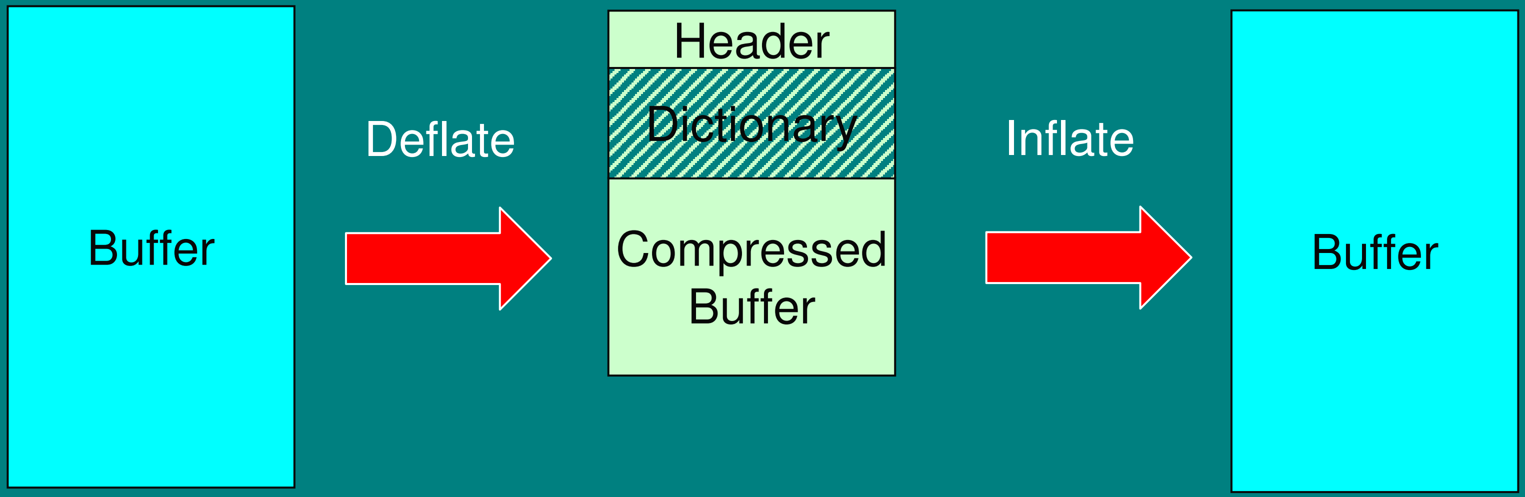
```
Observer::setEncription(new Rijndael128(Encription::generateKey128(“pass1")))
Observer::setEncription(new Rijndael256(Encription::generateKey256(“pass2")))
```



Compression

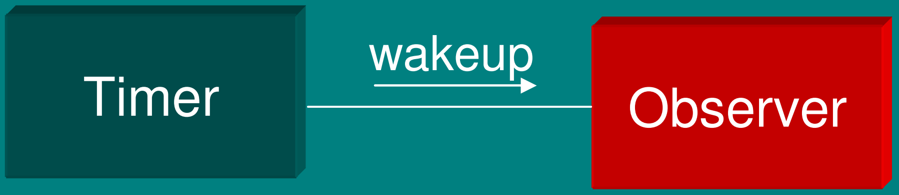
- MQ4CPP include a lossless data compressor/decompressor based on a dictionary coder algorithm
- Compression works using also a cache mechanism to avoid to send dictionary information and reducing the bandwidth needed. This mechanism works only in a peer-to-peer transmition.
- Compression is a cpu-consuming process. Use only if you have a low bandwidth connectivity with your peer.
- To use compression:

```
Observer::setCompression(new PacketCompression())
```



Start a timer

```
class MyClient : public Observer
{
private:

public:
    MyClient(const char* theName) : Observer(theName)
    {
         setEncription(new Rijndael128()); // Optional
         setCompression(new PacketCompression()); // Optional
         ...
         SCHEDULE(this,200); // Set a timer of 200 ms
    };

   virtual ~MyClient() {};

protected:
   virtual void onWakeup(Wakeup* theMessage)
   {
         // Each 200 ms
   };
};
```



Client/Server

```
class MyClient : public Client
{
public:
        MyClient(const char* theName, char* theHost,int thePort, const char* theTarget)
        : Client(theName,theHost,thePort,theTarget)
        {
                setEncription(new Rijndael256(Encription::generateKey256("MyVerySecretPassword")));
                setCompression(new PacketCompression(false));
                setTopic(MessageProxyFactory::getUniqueNetID());
                send(….);
        };

        virtual ~MyClient() {};

protected:
        void success(string theBuffer) { … };
        void fail(string theError) { … };
};

class MyServer : public Server
{
public:
        MyServer(const char* theName) : Server(theName)
        {
                setEncription(new Rijndael256(Encription::generateKey256("MyVerySecretPassword")));
                setCompression(new PacketCompression());
        };

        virtual ~MyServer() {};

protected:
        string service(string theBuffer) { … };
};
```

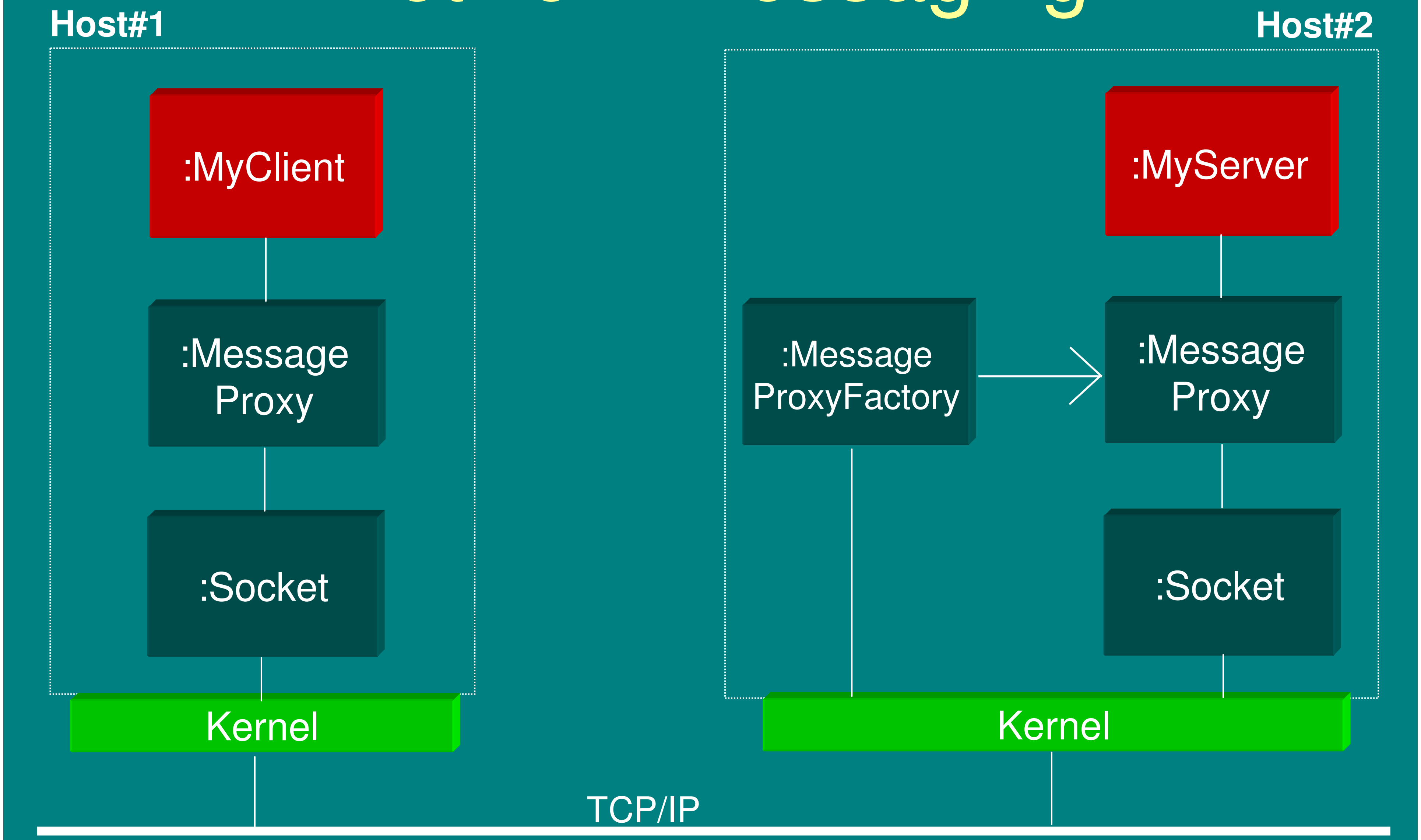
Set topic for remote switching

Service invocation

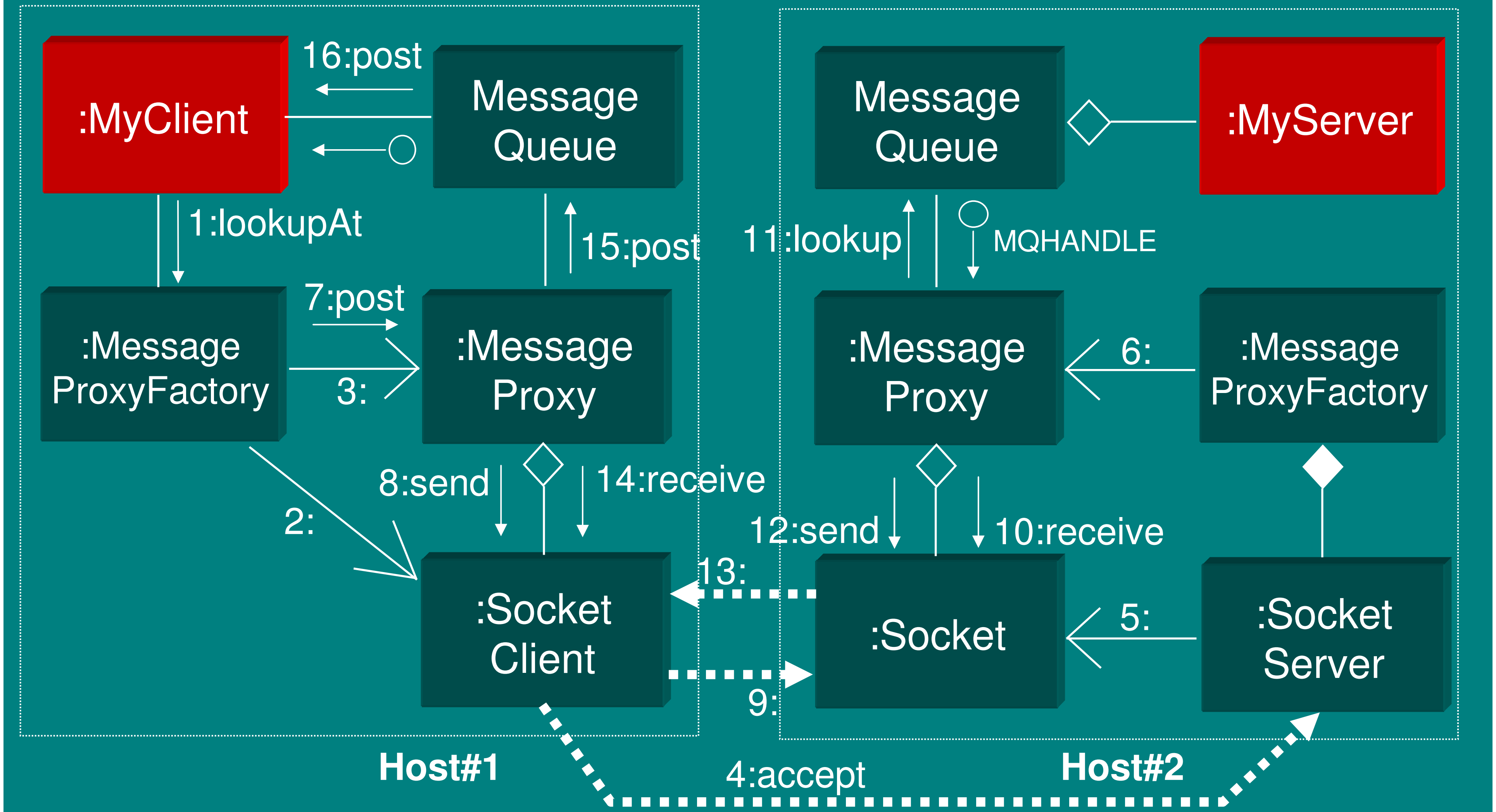
Service result

Service implementation

Network messaging

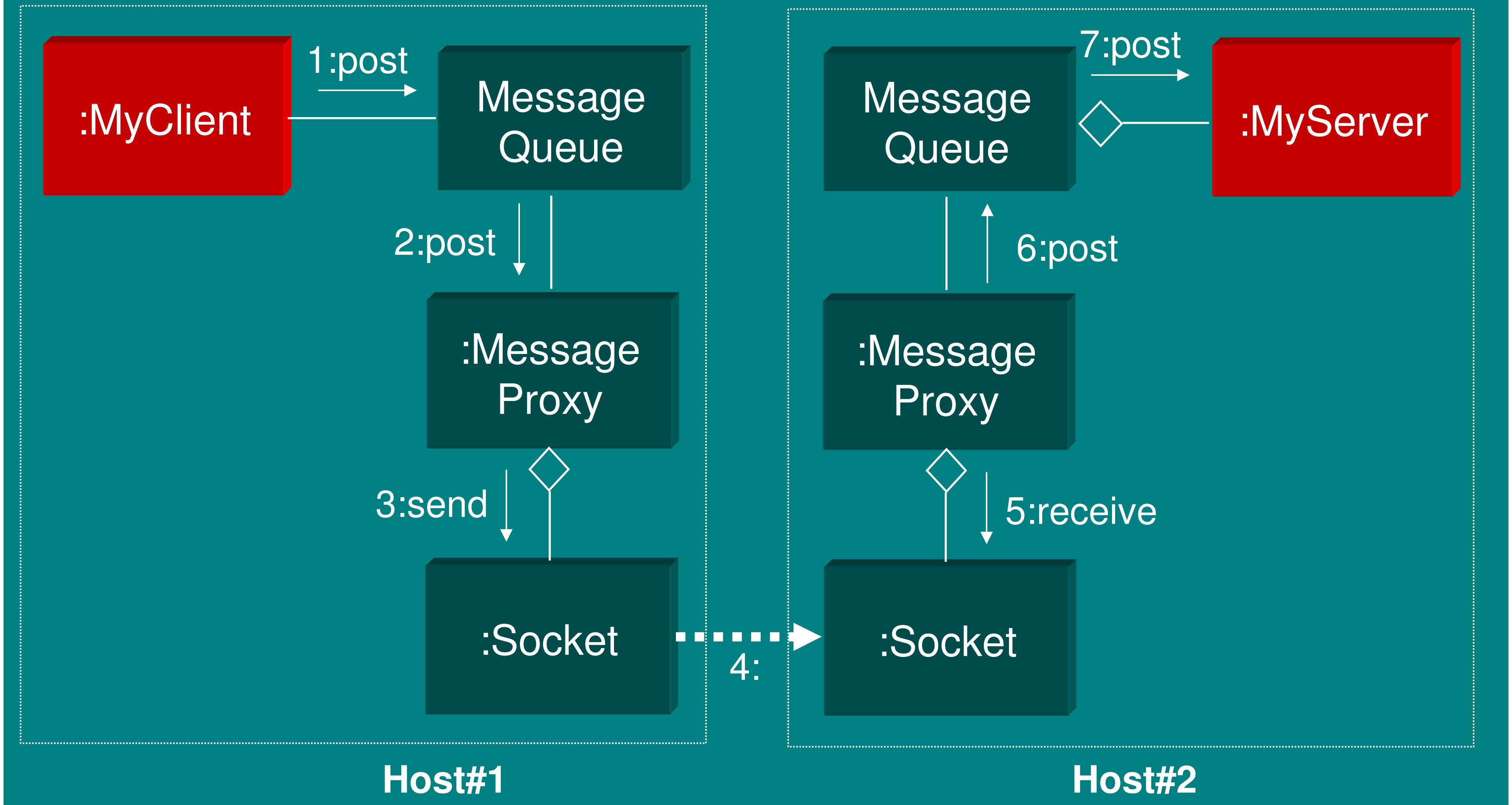


Remote lookup

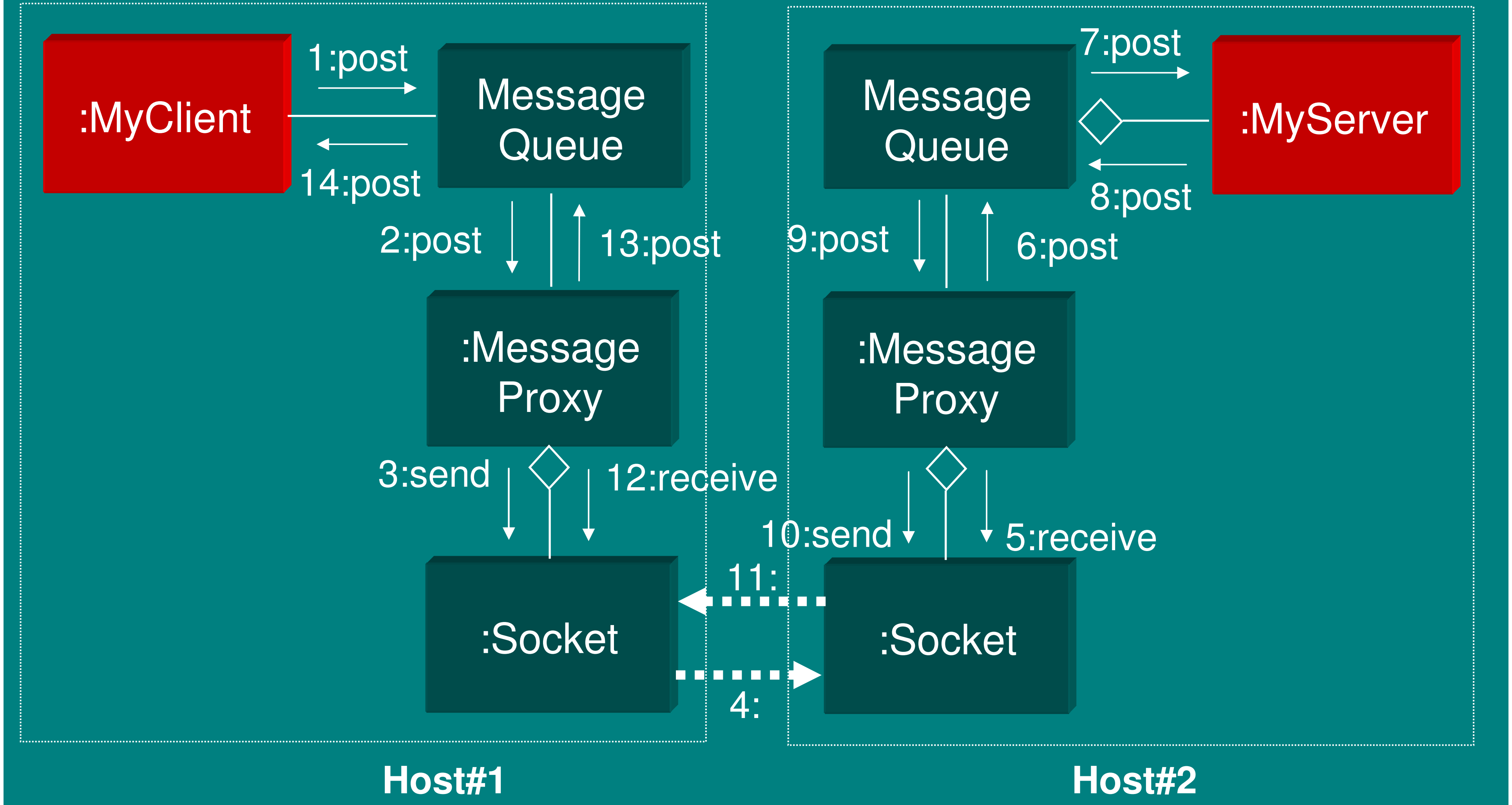


Riccardo Pompeo – http://www.sixtyfourbit.org – mailto:riccardo.pompeo@sixtyfourbit.org

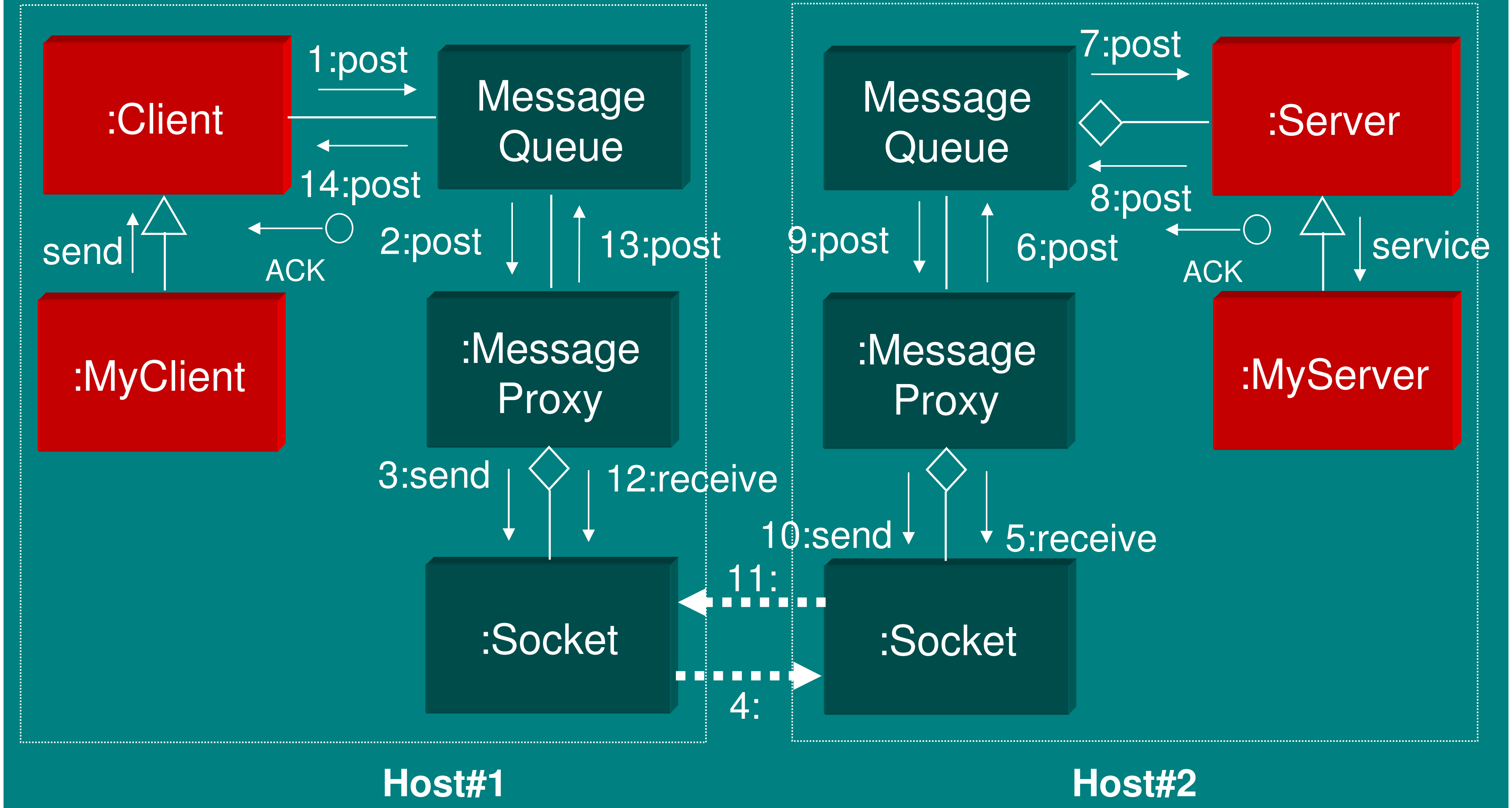
Unsolicited messaging (example5.cp)



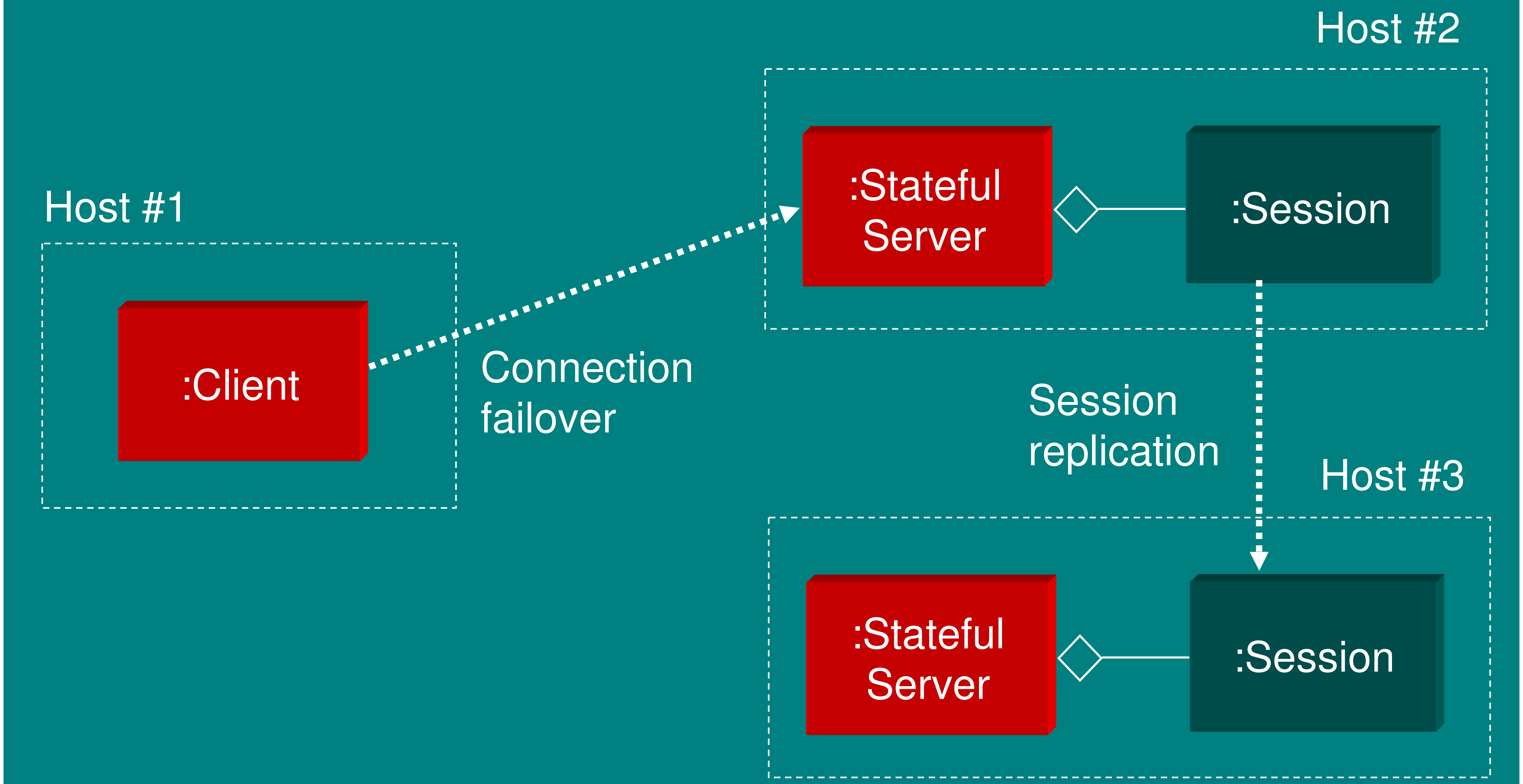
Conversation (example6.cpp)



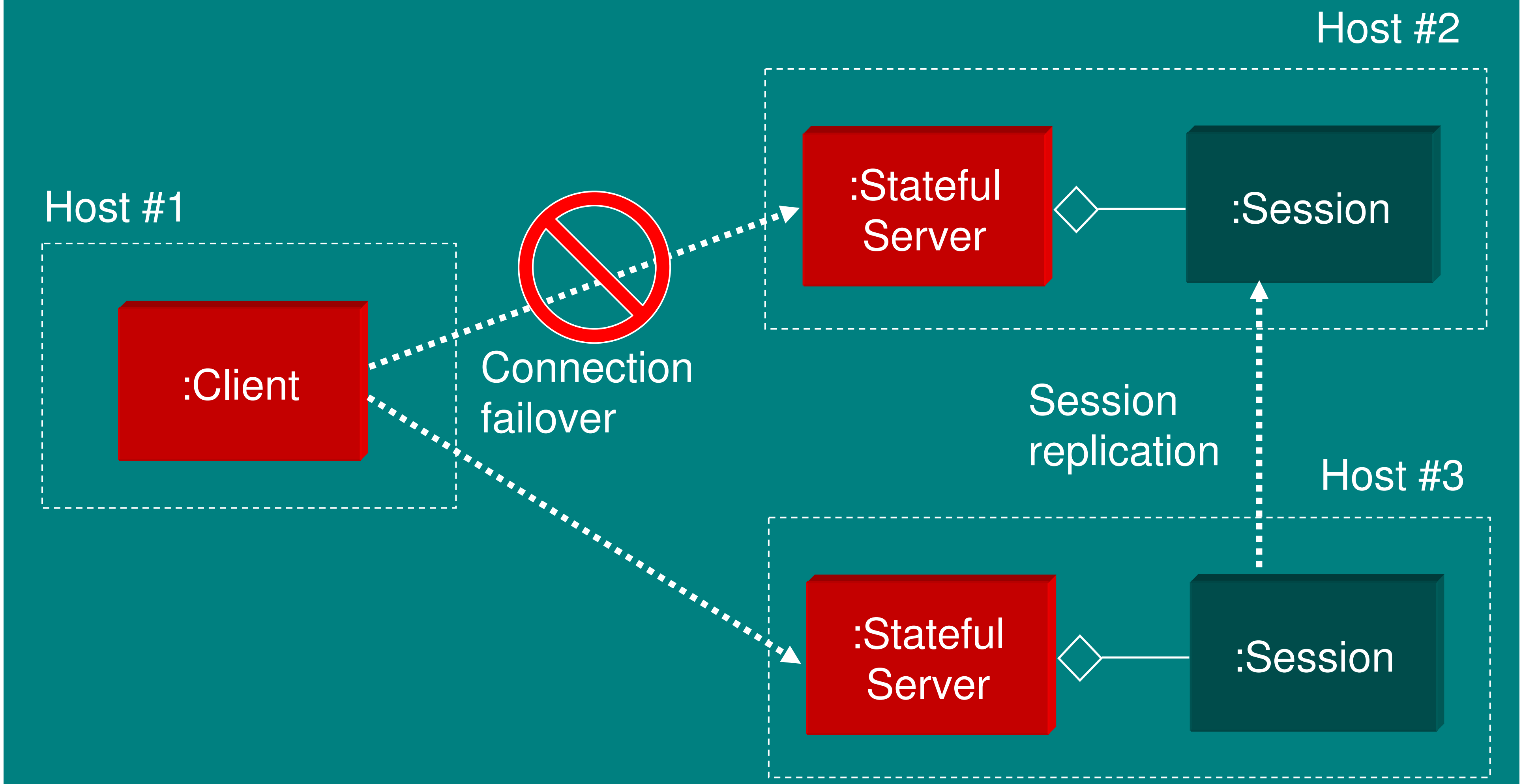
Reliable Request/Reply (example7.cpp)



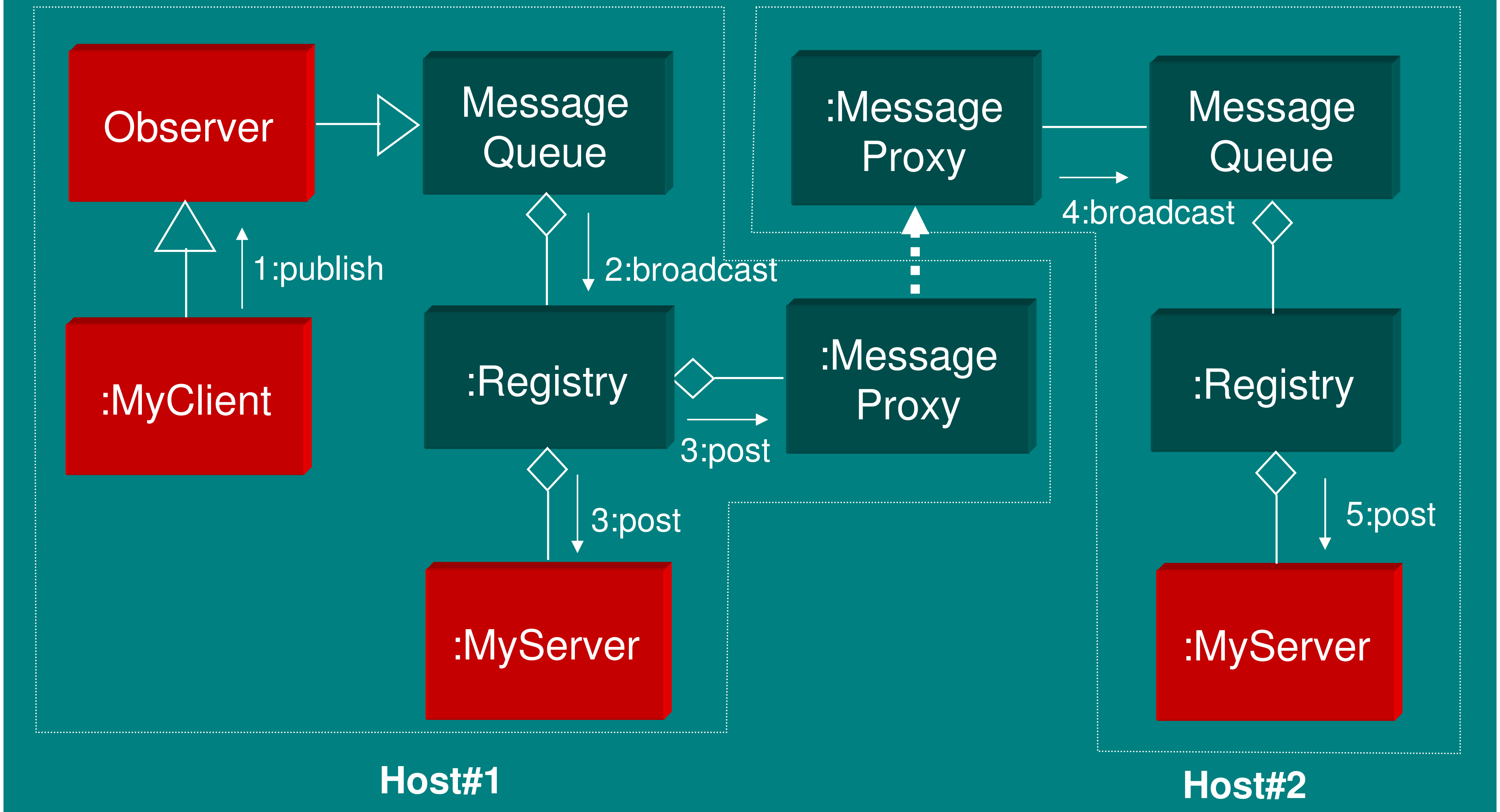
Session replication (example8.cpp)



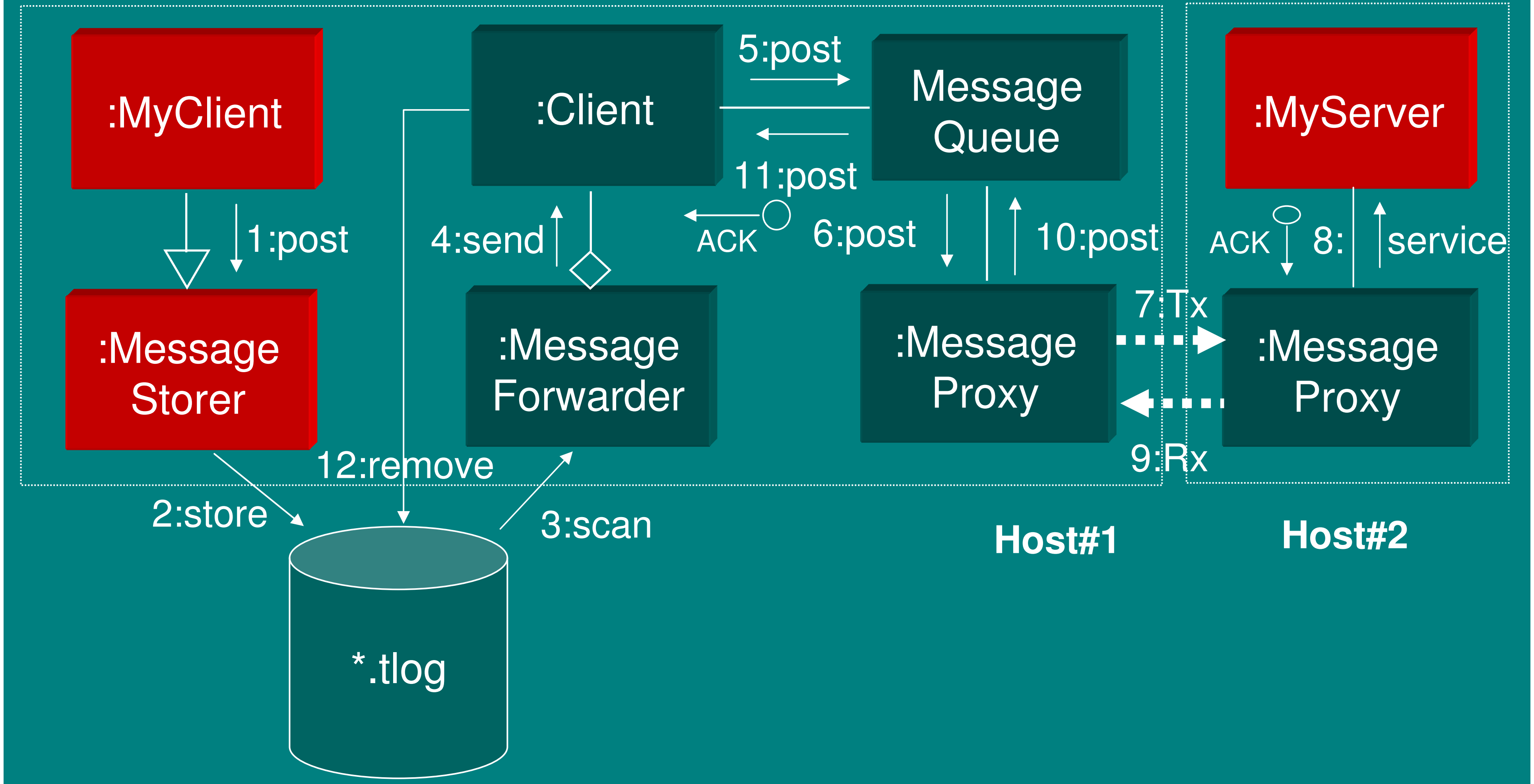
Failover (example8.cpp)



Publish/Subscribe (example9.cpp)

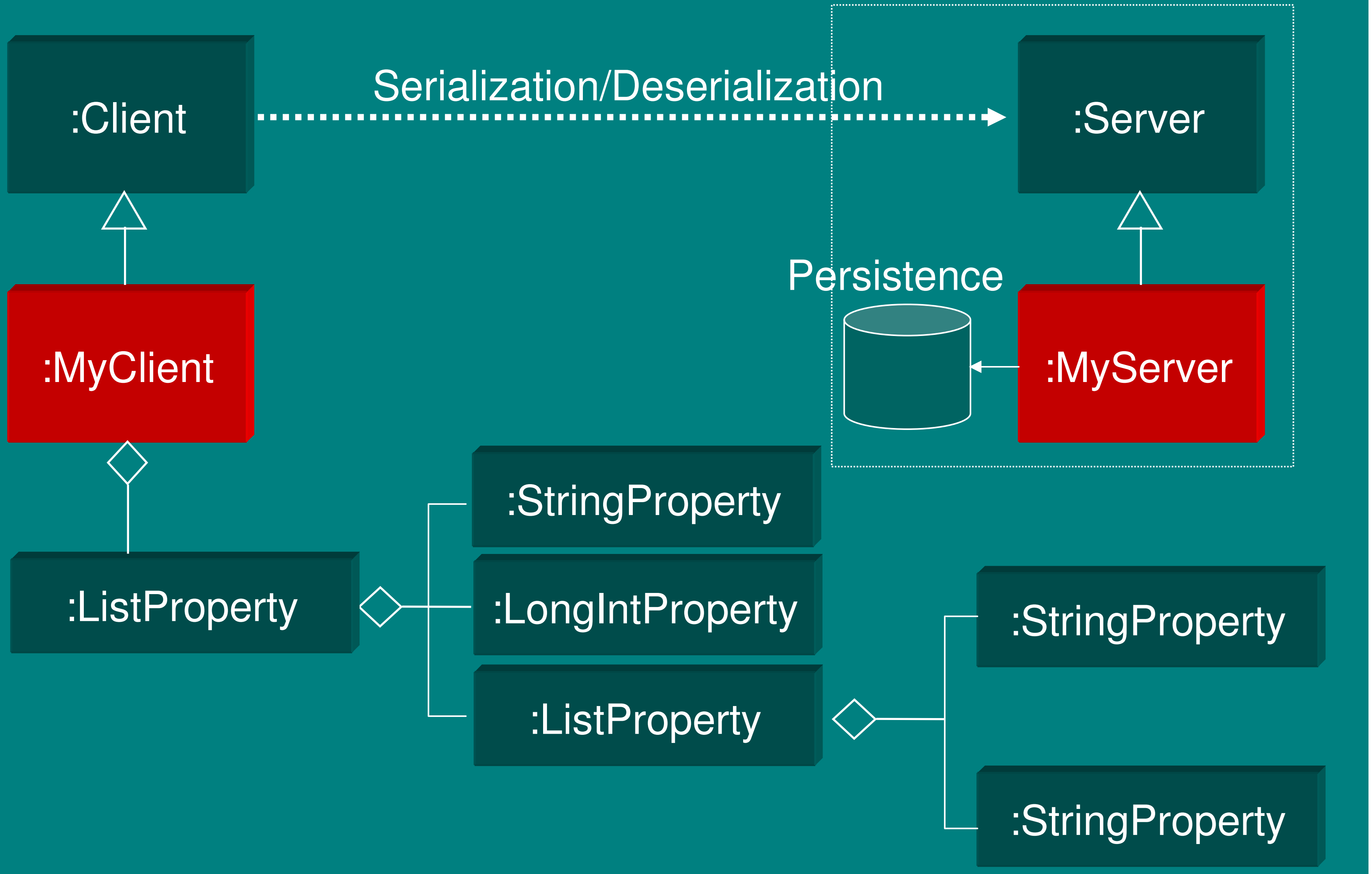


Store & Forward (example10.cpp)

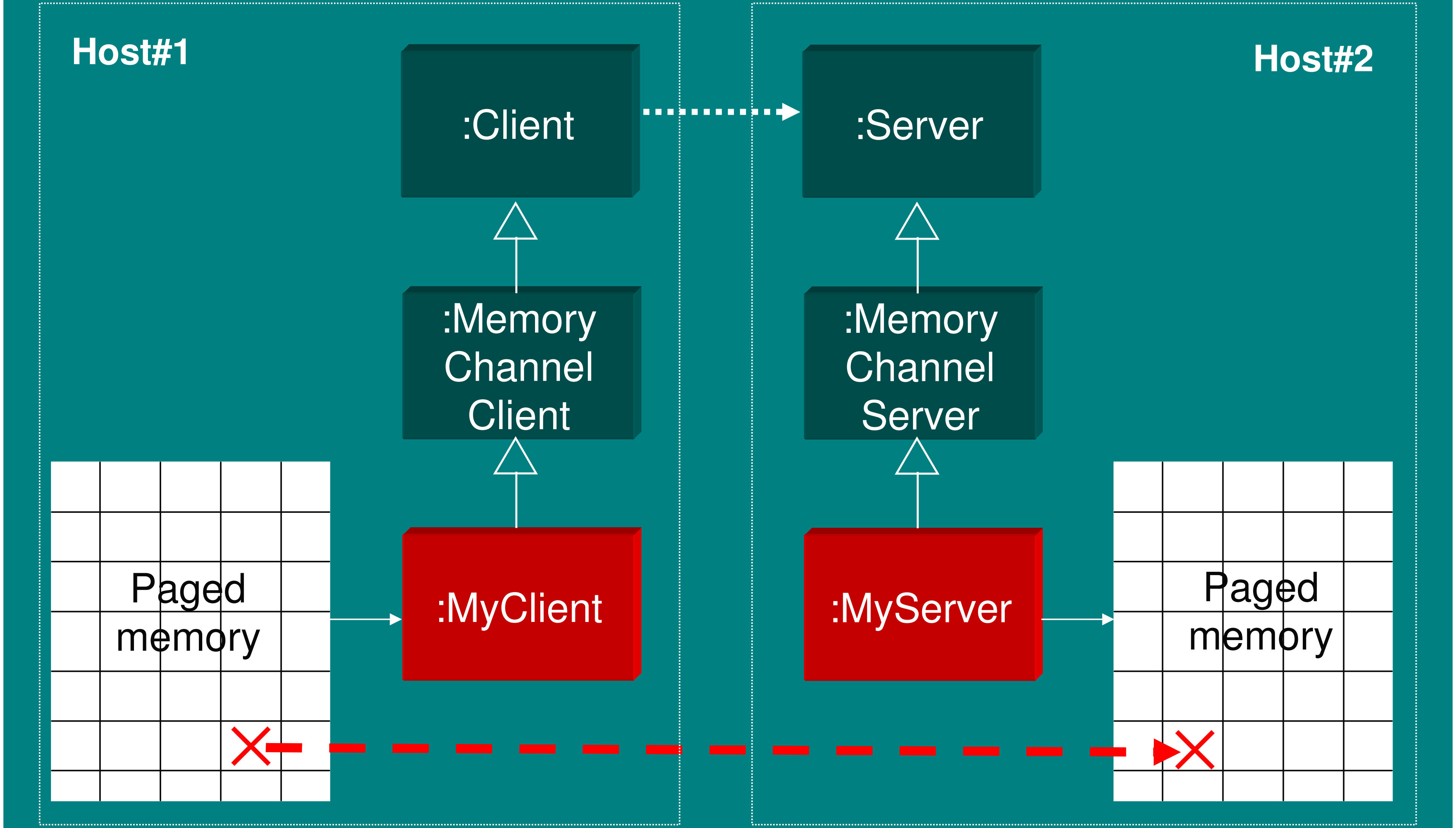


Riccardo Pompeo – http://www.sixtyfourbit.org – mailto:riccardo.pompeo@sixtyfourbit.org

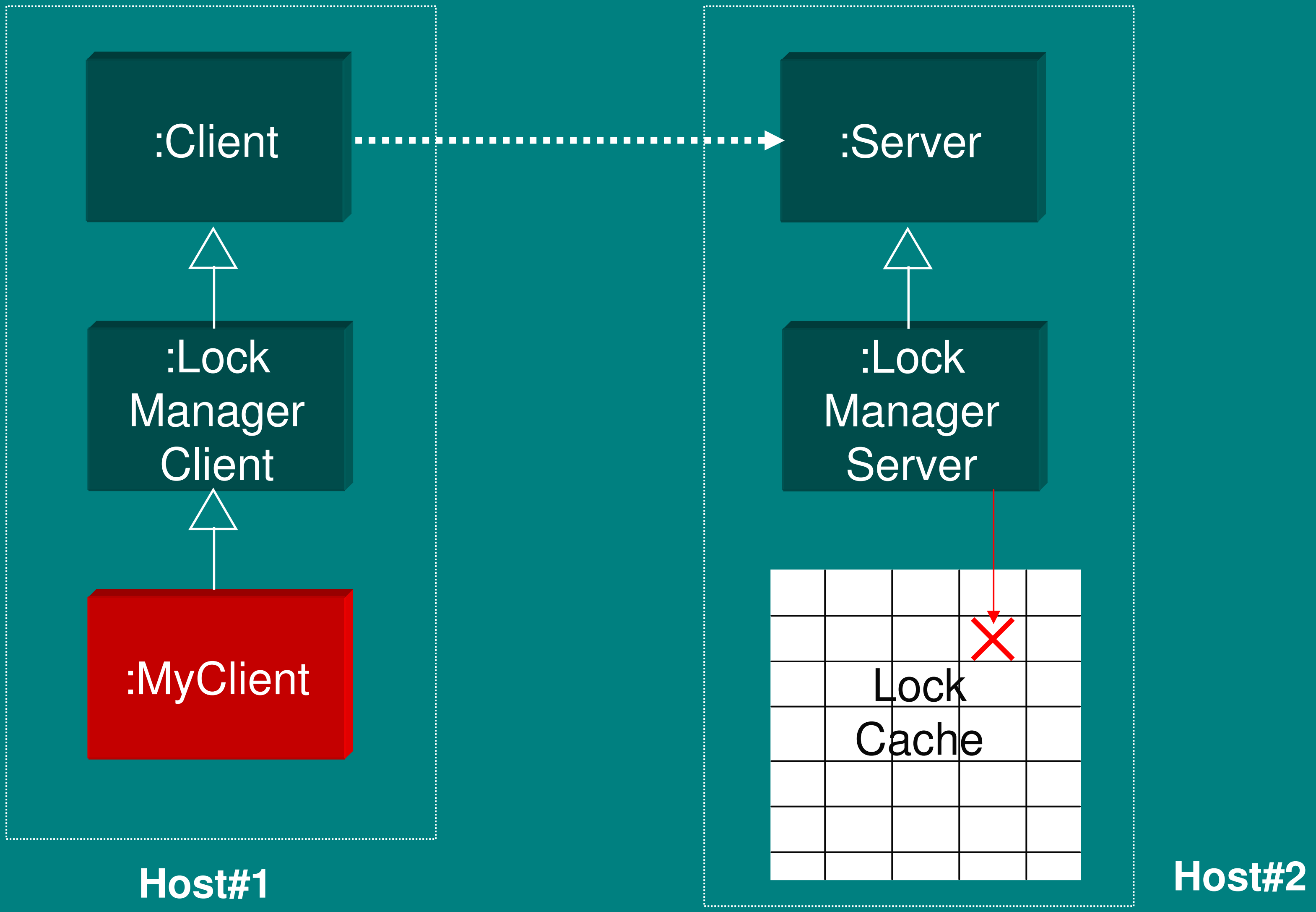
Properties (example11.cpp)



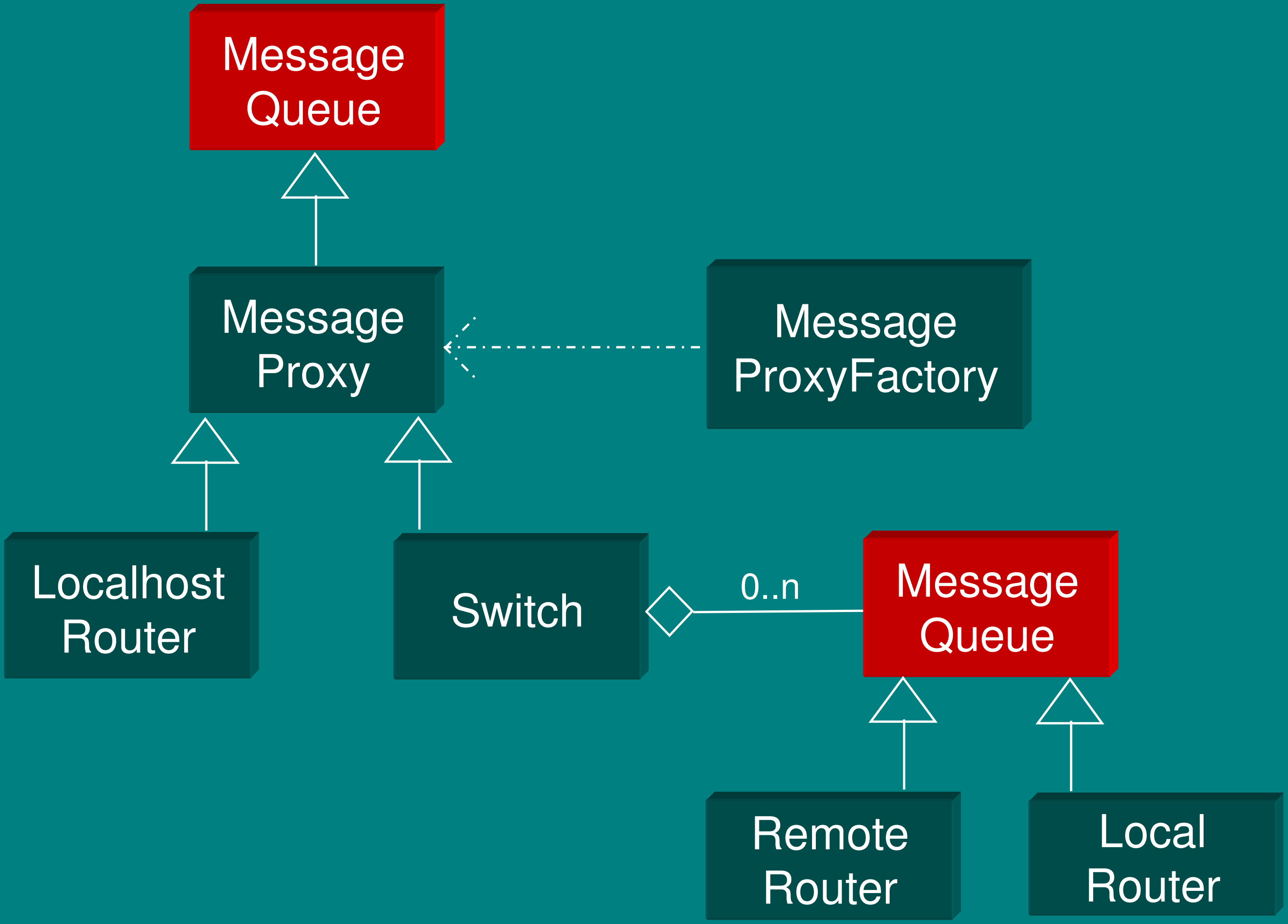
MemoryChannel (example12.cpp)



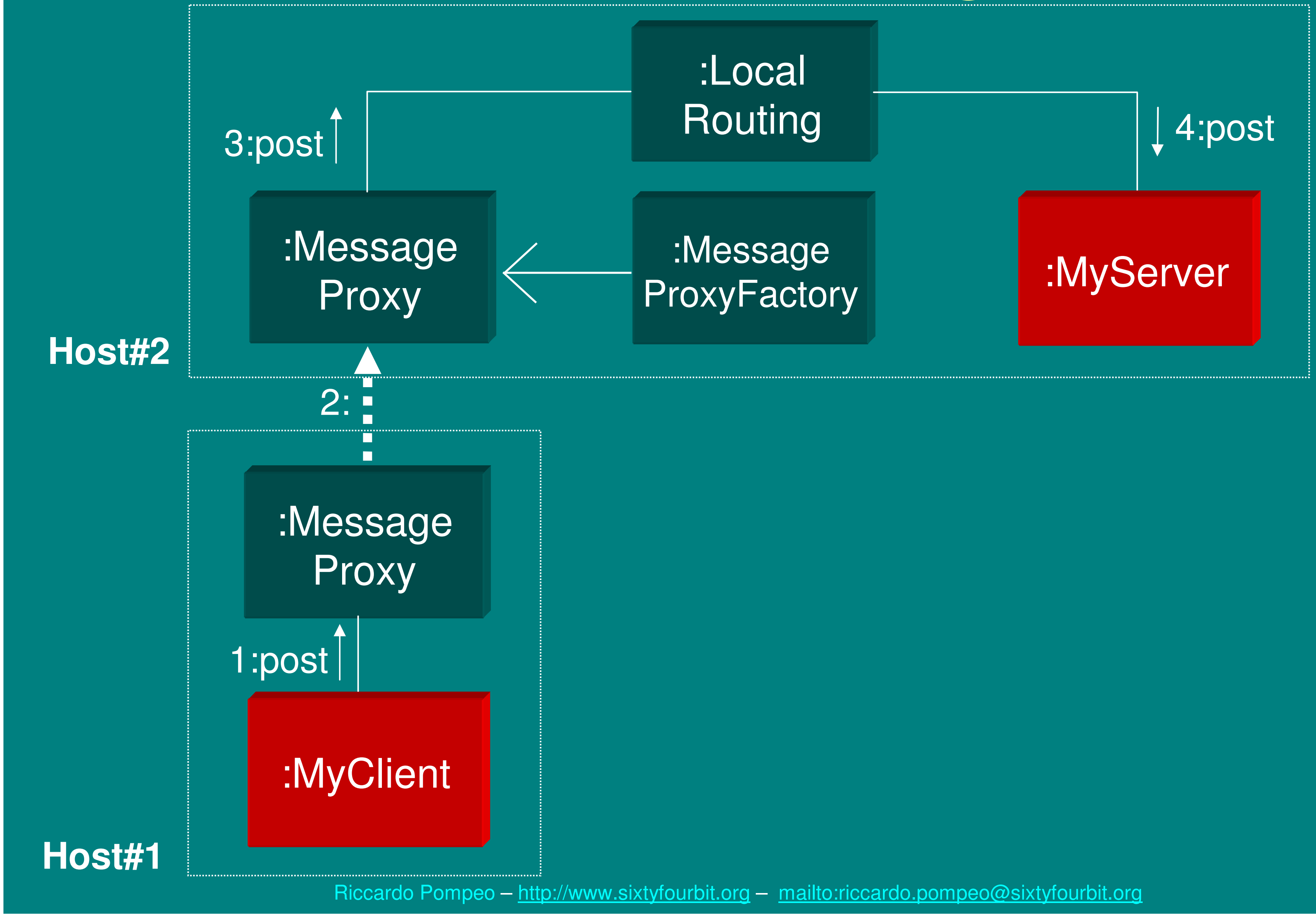
LockManager (example13.cpp)



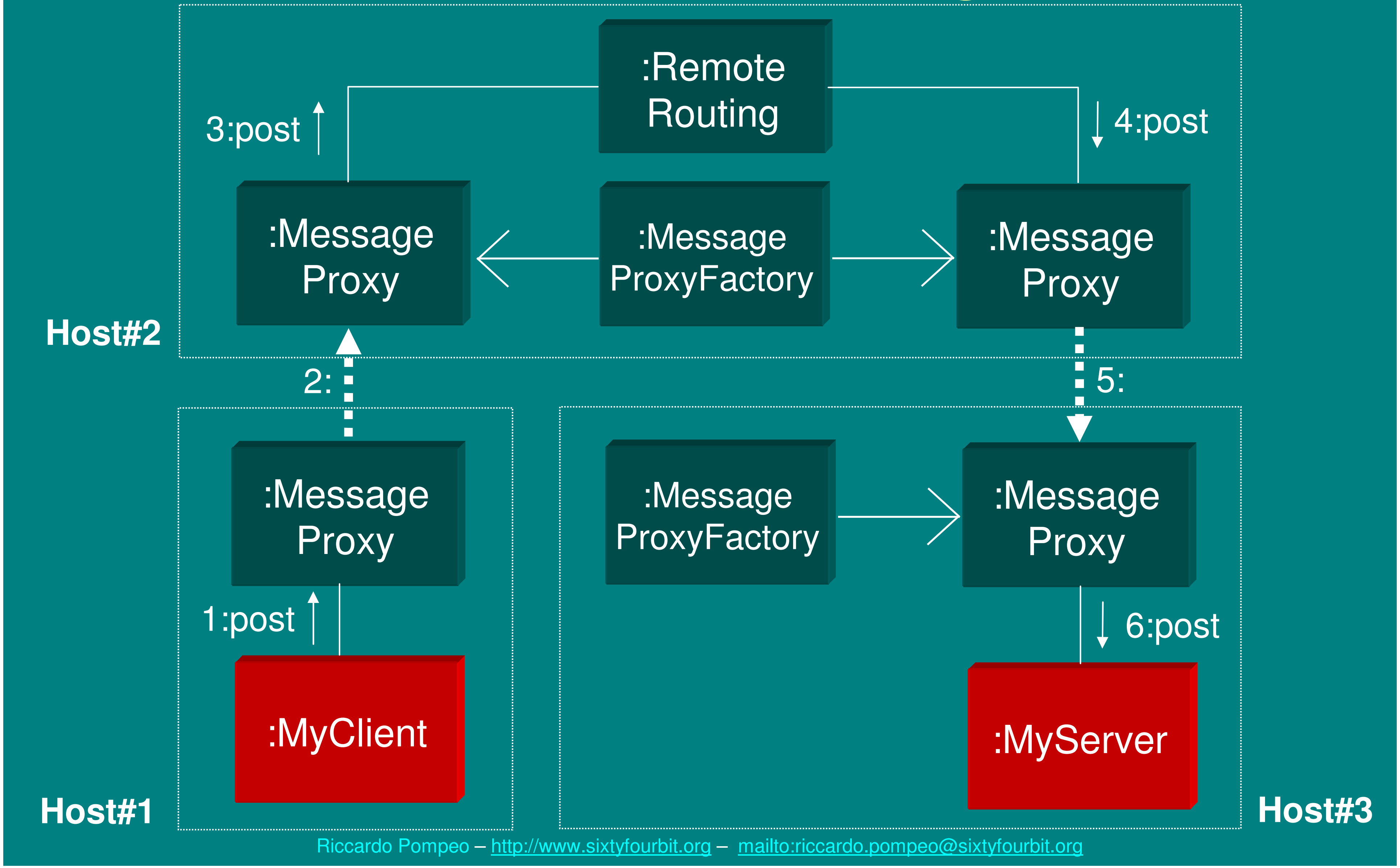
Message routing



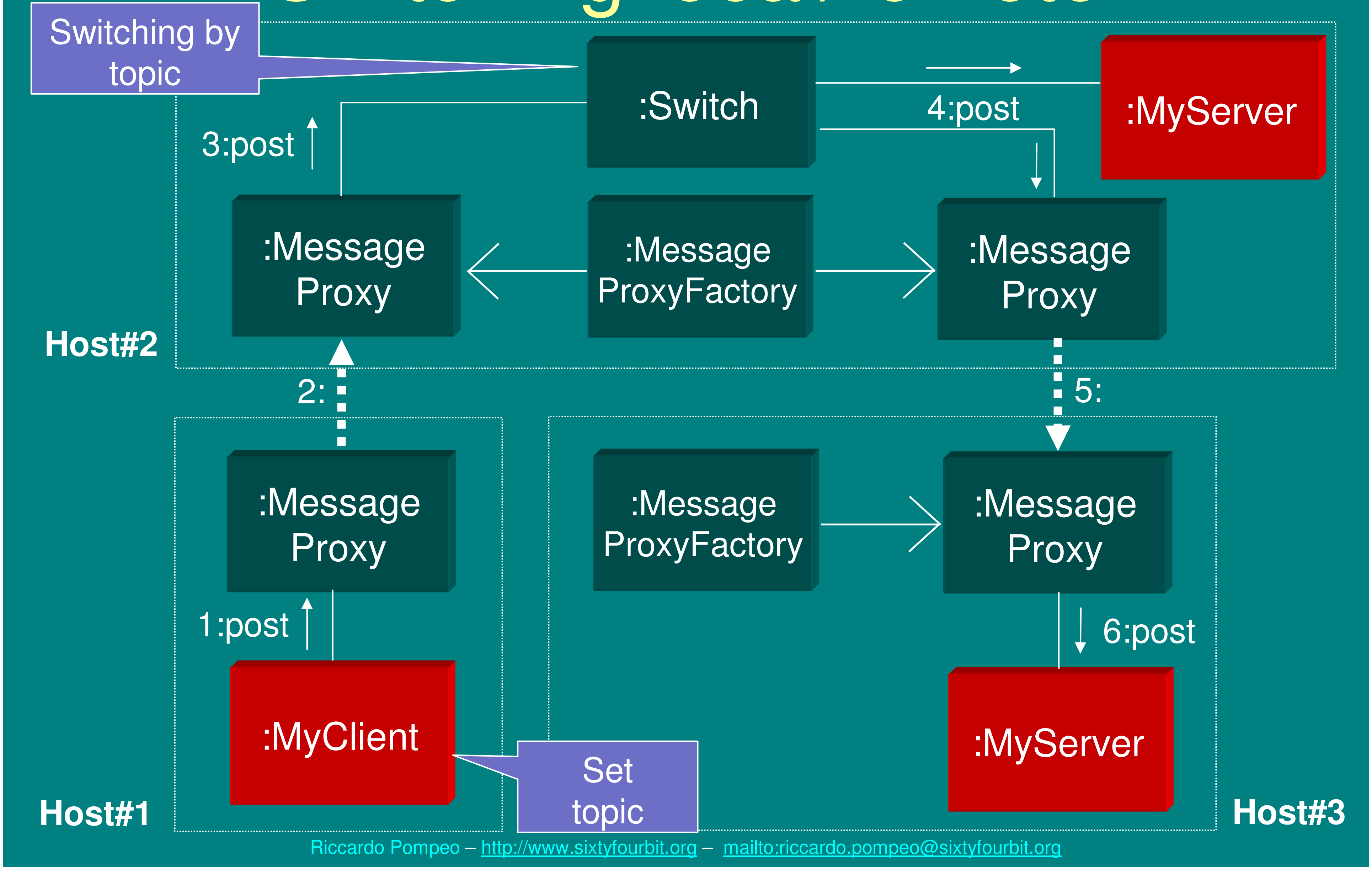
Local routing



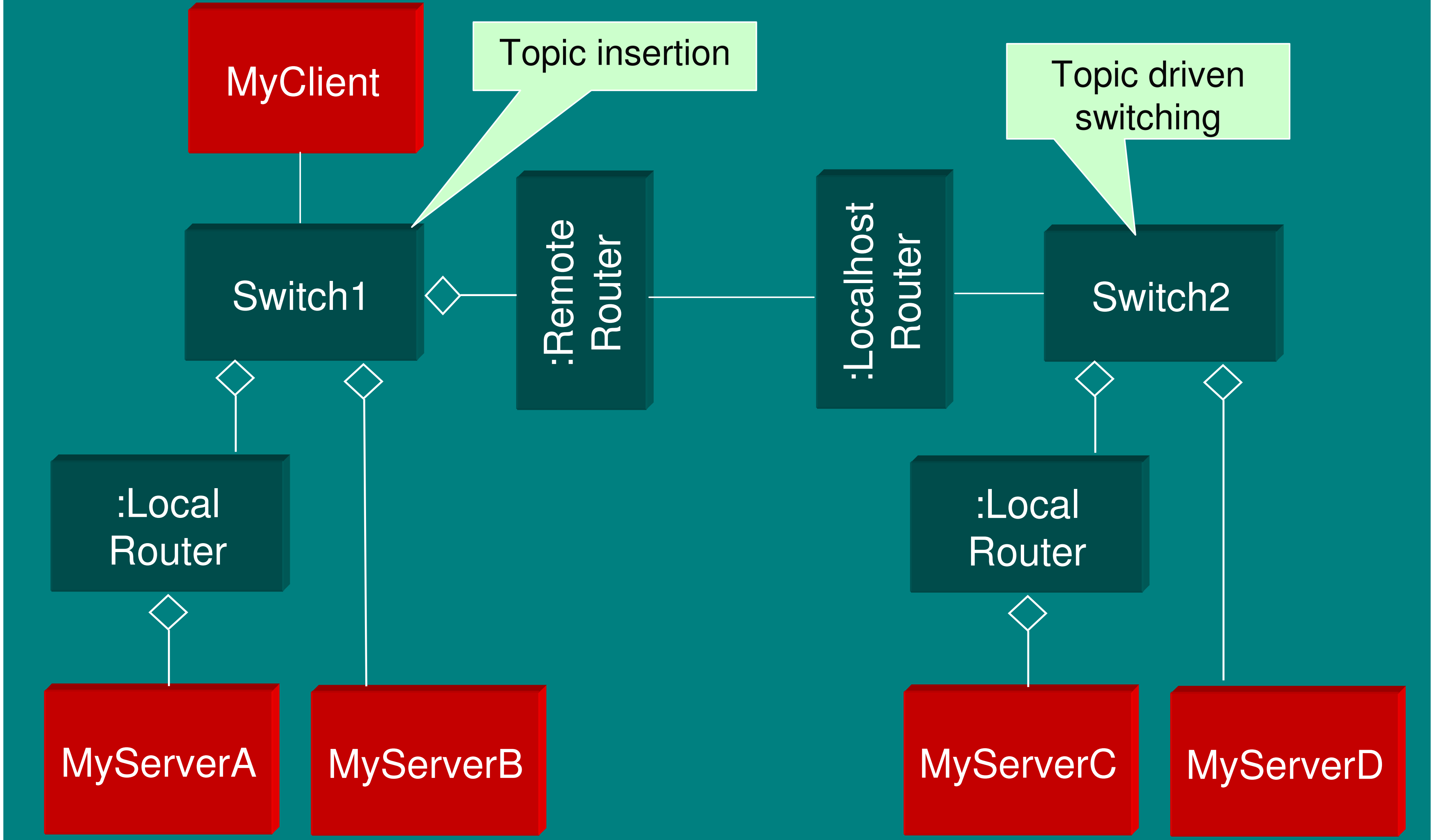
Remote routing



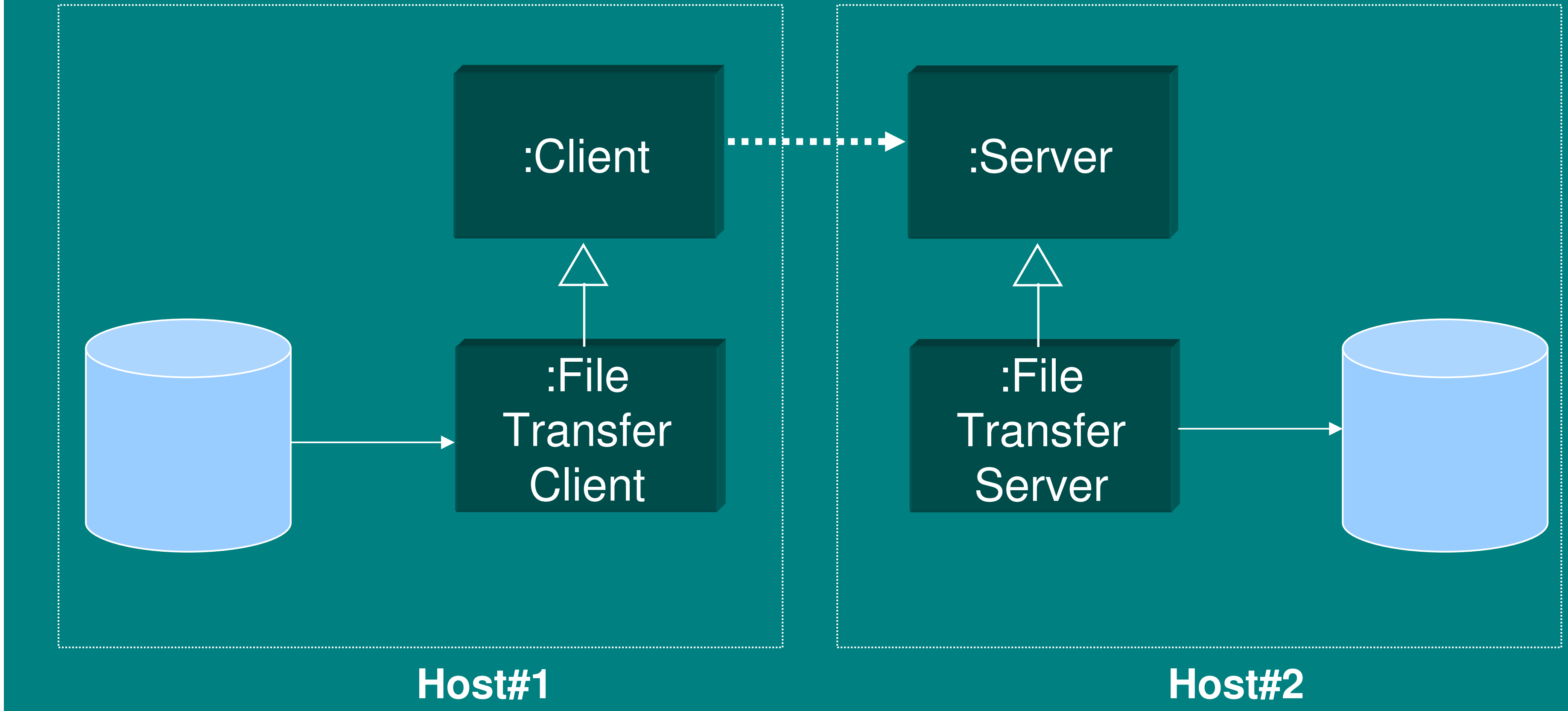
Switching local/remote



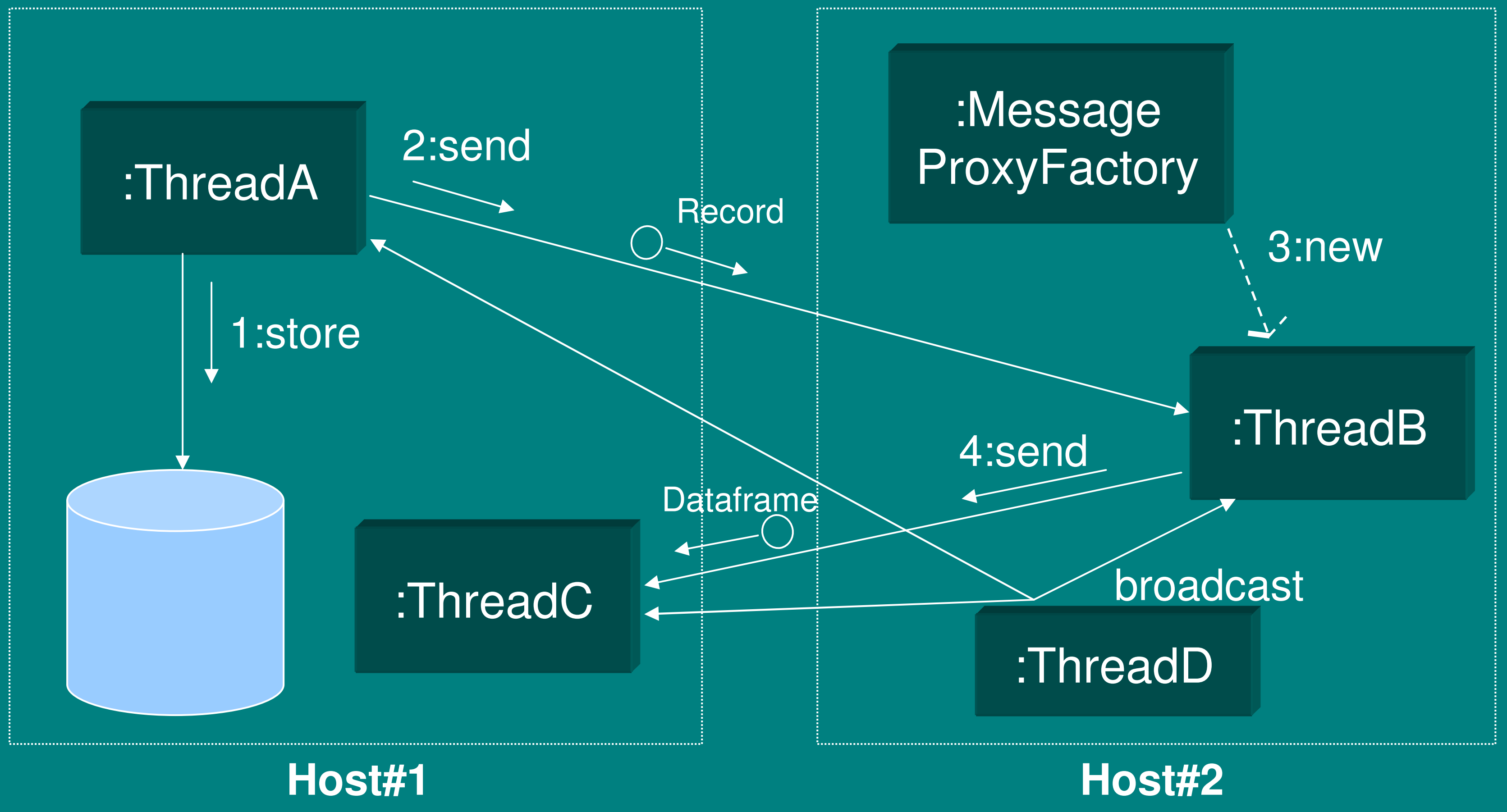
Routing (example14.cpp)



FileTransfer (mqftp.cpp)



Peer to peer (peer.cpp)


:ThreadA
2:send
Record
:Message
ProxyFactory
3:new
1:store
:ThreadB
4:send
Dataframe
:ThreadC
broadcast
:ThreadD
Host#1
Host#2