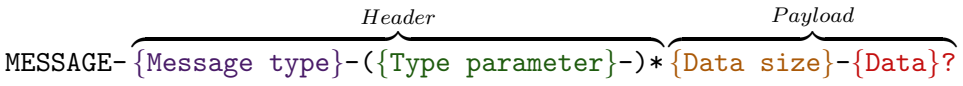
Chapter 1

Runtime Implementation

1.1 Messaging

1.1.1 Format

All messages start with a fixed prefix (`MESSAGE`) and end with a line break and their different parts are separated by hyphens. Messages consists of two logical parts: the header and the payload. The former specifies the type of the message and provides other information as type parameters which may be needed to process this kind of message, while the latter carries additional information in a form of a blob and its size in bytes.



Structure of the messages

Type parameters and payload The difference between the type parameters and the payload data is that type parameters can be understood and processed without 'unsafe' operations such as java deserialization. Their arity and types are determined by the message type and their values may consist of primitives, strings or enumerations. This – combined with a size limit on them – allows the runtime handler to always safely parse, process and optionally route the messages without requiring to understand the potentially unsafe data of the payload.

The data payload on the other hand deals with binary data produced by one of the actors, and as such should only be parsed by the actor clients, which are already equipped with the proper protective capabilities. The payloads size (in bytes) is also included in the message for safe message handling.

Both the type parameters and the payload are converted to base 64 strings as to make message formatting and parsing easier – due to no necessary separator escaping.

Message type	**Parameters**	**Payload**
Start notice	————	————
Shutdown notice	————	————
Log entry	Target actor Log message	————
Error report	Error message	————
Challenge result	Points Max points	————
Match result	Three way result	————
Actor binary	Type	Binary (nothing on requests)
Proxy call	Target bot	Target method Call parameters
Call result	Called method	Result data
Bot timeout	————	————

Type parameters and payloads of message types

1.1.2 Conversion and serialization

Conversion and deconversion Conversion is the process during which domain representation of a message in the form of a Message object is converted into an intermediate state as it gets ready to be sent. Its result is MessageDTO, an object which holds the same header the original message did, but has its payload – if any were present – transformed into a pro-

cessable, base 64 encoded string representation. Deconversion is its inverse operation. It is a distinct process from the general serialization of messages, as to ensure independence between the algorithms use in the two processes.

This loose coupling is beneficial, as conversion is an unsafe routine that only actor clients should conduct. Message (de)conversion is required when dealing with remote proxy calls, and this breakup of concerns allows the runtime handler to parse and route a call request or call result message without attempting to decode its payload containing jvm objects as transformed by standard java serialization. As object (de)serialization is dangerous – due to the construction of arbitrary large serialization 'bombs' –, it is best to pass these messages to the actor clients who are equipped with proper security measures to safely minister it.

Serialization and Deserialization Serialization takes payload-less messages or converted messages and transforms them to the aforementioned general message format. This includes encoding the message header and concatenating it to the prefix and optional (already converted) payload. This encoding of the header – and its pair decoding in deserialization – can be done safely and strictly, as the contents of the header is specified by the message type and its parameters are simple values that can be easily converted to strings.

As all parties require (de)serialization to communicate, the components providing these services are shared across all of them.

1.1.3 Communication

All messaging is done over point-to-point, full-duplex TCP channels between the actor clients and the runtime handler. These connections are initialized by the clients, who as a command argument receive the host – usually localhost – and the port they should connect to. These ports are free ports chosen randomly by the handler and reserved before the start of the clients.

Once these channels have been established, communication should begin as described by the communication model – starting with the clients requesting the game specific actor binaries.

If a game if finished – in a normal way or because of errors – or a communication channel should fail (e.g. due to unexpected crash of an actor), the runtime notifies all (still reachable) clients with shutdown notices.

After sending and receiving these, all participants are to assume the game has ended, the clients stop themselves and the handler reports the result of the challenge or match.

Just like serialization, low level communication is helped by components shared across all members of the runtime model.

1.2 Runtime handler

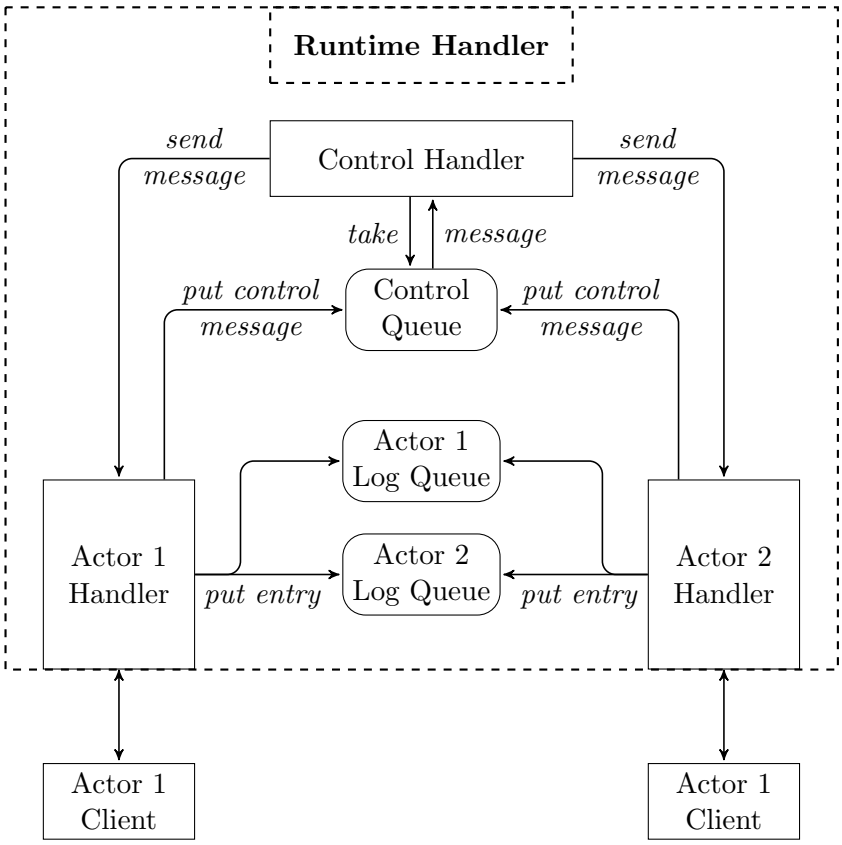
A runtime handler is made up of a couple of active components and passive data structures. Active components are primarily based on their own thread and communicate with each other either through direct calls or the passive structures. The are two kinds of these active components, a – per runtime handler – singleton control handler, and actor handlers for each actor client runtimes. Passive components come in the form of a control queue and log stores for each actors.

1.2.1 Control handler

The control handler is the heart of the runtime handler, as its name implies, it directs the flow of the game and manages the actor handlers. At its core it is a finite–state machine that processes control messages and optionally instructs the actor handlers to send messages based on the actions.

It is important to note that the control messages it consumes are slightly different from the general messages sent between the members of the runtime model, as they have been preprocessed by the actor handlers. The two main changes are that log messages have been removed (they are processed by the actor handlers), and each message contains its source actor. The control handler acquires these messages from the message queue, which acts as a thread–safe buffer between the components. The possible control state are:

- **Connection await** The initial state, it signals that not all actor clients are ready to start as not all have connected or received all the binaries they need.
- **Wait for Engine** The engine is working.
- **Wait for Bot** The engine had requested a proxy call, whose result has not been returned from the target bot's actor yet.

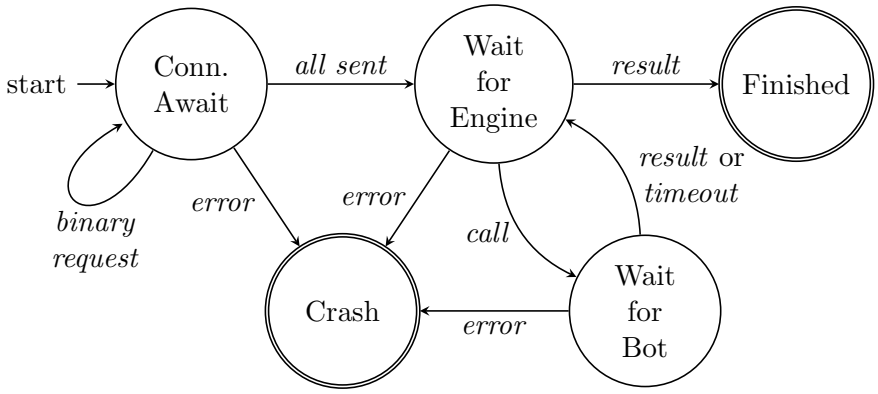


Runtime handler simplified component model
Challenge – two actors

- **Finished** The game had finished normally. *End*
- **Crash** The game finished abnormally due to some error. *End*

To simplify game flow management, the control handler is an extended state machine which allows states to store variables as internal state. This data allows for selective transitions to happen and help store results in the final (accepting) states. Each state may have its own set of parameters as follows:

Game flow The controller starts off in the `Connection await` state and remains there until all actors have received their required binaries. It keeps track of the sent libraries and after the last of these has been issued, a 'Start notice' message is sent to the engine client, the state is changed to `Wait for Engine` and the game starts.



Control state machine

Control state	**Parameters**
Connection await	sent binaries by receiver client
Wait for Engine	————
Wait for Bot	target bot, called method
Finished	challenge/match result
Crash	optional result, error description

Parameters of control states

During the game as the engine makes call for which the bots respond, the state is bouncing between `Wait for Engine` and `Wait for Bot`. As a security measure, the target bot and called method are also stored, so that it can be verified that the right bot answered a call. As per the game model, 'Bot timeout' messages are also possible in the place of 'Call results'.

After the game had come to a conclusion and the engine had sent the proper result message, the state reaches `Finished`, where it keeps the game result and stays.

Of course, while in any active state, an error might also happen, in which case the controller changes to `Crash` state. Although in this case a game result have not been reported by the engine, the controller might set a valid result anyway, based on the context in which the error occurred. If for example the game crashed due to a runtime exception while executing a code of a bot, the result of the challenge or match will be set to error from that bot – a result that is processed similarly to the opponent winning in

the case of a match, or a 0 point run in the case of a challenge.

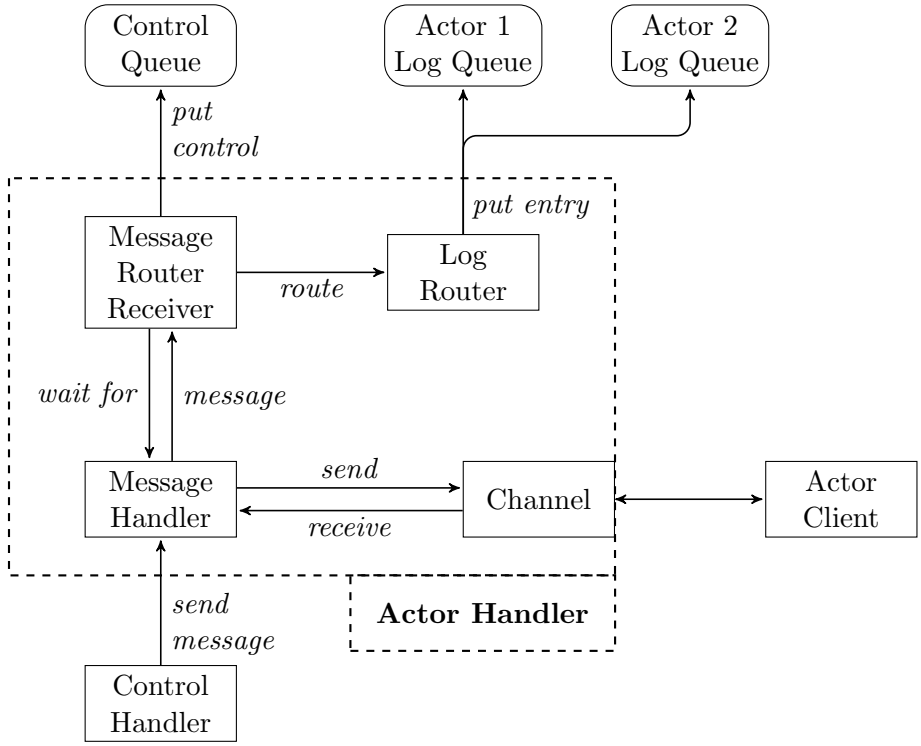
1.2.2 Actor handler

In contrast to the control handler, separate actor handlers are create for each actors. These composite components are responsible for communicating with the runtime clients, processing incoming messages, storing log entries, and forwarding control messages to the control handler in a digestible format.

Low level communication with the clients is done using a channel, an object which provides a safe, message–aware wrapper for the underlining network sockets. On top of this sits a message handler, which now provides operations based on the type of the clients (engine or bot). This handler is accessed by two components, the already mentioned control handler and the message router receiver. Both of these objects run on their own threads while using the message handler, but due to them only using the sending or receiving capabilities respectively, (and the fact that the underlying architecture is full–duplex capable) these actions can happen independently. The message router receiver's thread can be considered the actor handler's main thread, as it performs most of the tasks of the handler. It blockingly waits for incoming messages, then routes them according their purpose.

Control messages get transcribed into the format the control handler can understand and put onto the control queue, from where they will be taken by the control handler. Log messages on the other hand are passed to a log router, an object which puts them into their target's log queue. As game engines have the power to simultaneously send info entries to multiple actors, this might mean more than one overall targets.

Actor handlers are designed to be as reusable as possible when dealing with different types of actors. This means that only the message handler and the log router differ based on actor kind – as they either must show different sending methods to the outside, or must work fundamentally differently while routing messages –. Also their characteristics are hidden from the other components (except for the control handler) via abstraction achieved by inheritance of a base.

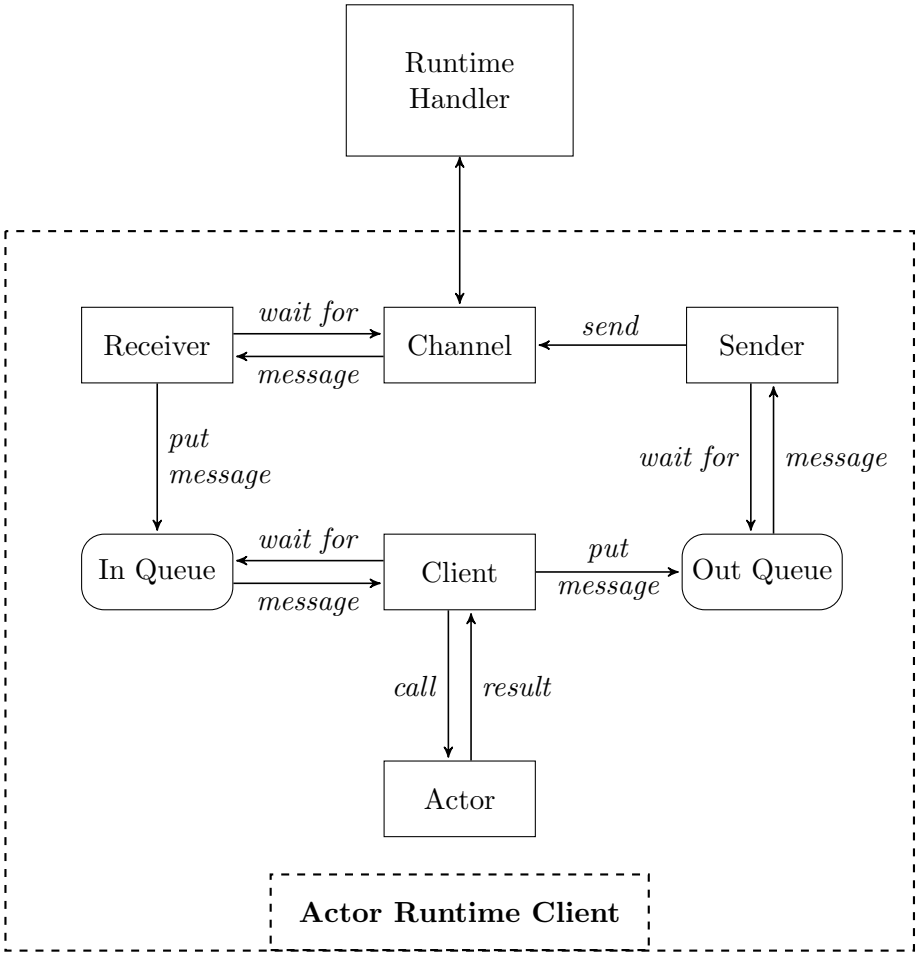


Actor handler model
Challenge – two actors

1.3 Actor runtime

The structure of actor runtimes is fairly simple and is largely identical between engine– and bot clients. Their core component is the 'client', which provides the logic necessary to manage the actor assigned to them, and therefore of course contains actor role (engine or bot) specific code. The rest of the components involved in the communication process are equivalent in each cases. The client is based on three active objects all running on separate threads, a message receiver, a message sender and the aforementioned client. These elements are communicating with each other – in a one–way manner – over blocking queues.

The receiver waits for messages over the communication channel, processes them and places them in 'In queue', from where the client can get them. It may also initiate the exit of the client if a shutdown notice is received. Likewise, the sender sends messages placed in the 'Out queue' by the client.



Actor runtime client model

1.3.1 Base client

As the client's workflow is dependent on the actor's type it is managing, it is realized as two different classes – the `EngineClient` and the `BotClient` –, but many common functions are outsources from these to shared dependencies.

Class loading and security

The most important job of all actor clients is the provision of a safe runtime environment where the actor's and the game api's inherently untrusted code may be executed. This is supported by the standard solutions provided by the java virtual machine. The heart of guarding valuable resources – such

as files or I/O capabilities, is the strict separation of safe– and unsafe code with their respective permissions assigned.

Safe code is constituted by all the sources of the runtime client and its dependencies such as java libraries used – effectively the whole codebase. This partition is granted all possible permissions using a policy file, which is – with the enforcing security manager – put into effect via program arguments passed to the jvm.

client.policy

```
grant codeBase "file:${trusted.codebase}" {
    permission java.security.AllPermission "", "";
};
```

This policy grants all permissions to classes whose source URI matches the the file path also set as a vm argument through `-Dtrusted.codebase`. This property is in turn set by the starter script to the executable jar's path that not only contains the actor client but also all of its necessary dependencies as well.

As a security manager is installed, other code sources have no granted permissions by default, therefore all other (unsafe) code parts need only not to be marked as being from the same source.

The loading of the user–defined code blocks is implemented by a custom class loader (`BinaryClassLoader`) which works on in–memory byte array representations of java archives (JARs). These libraries are sent over to the actor client in actor binary messages, then the client passes them to the classloader for processing. The loader iterates through the compiled referencial type entries of the archive (`.class` files) and defines them using the standard native class loader methods. The binary class loader also sets these classes protection domain, which – due to different source definitions – provide no granted permissions to them.

Although this way no permissions are granted to the user defined programs, they would still be able to engage in unwanted behavior, namely multi–thread operations. The game model specifies strict 'ask–respond' style communication between the engine and the bots for clear gameplay and enforceable timeout limiting. This means that when – as a result of a proxy invocation – a method of a bot is called, it must finish all operations

when that method returns. As threads cannot be safely killed once started, this effectively forbids custom thread creation, even in non–straightforward ways, such as the use of an executor framework. Fortunately, while 'no granted permissions' still allow thread creation, there is a safe way of disabling thread creation for unsafe code. At the initialization of a thread object, the environment does check access permission, but the default behavior is to allow thread creation if the thread is created into the current thread group. The look up of the current group is done by the set up security manager using its `getThreadGroup` method. By overriding this method of the default security manager, we can return the root thread group, which will always be different from the caller's thread group, thus thread creation will fail when attempted from unsafe code.

Actor initialization

Initializing an actor instance to manage first starts with finding the game's bot interface. This is done by searching through the user defined types and finding the appropriate class object. This capability is shared between the different types of runtime clients.

Engine client After the bot interface has been found, the engine client creates one or two stub instances for this interface. This is done using the java runtime's built–in dynamic proxy creation capability through the `Proxy` class' static `newProxyInstance` method. This method takes an invocation handler – a callback that takes as parameters the context in which a method of the proxy was called, and meta data about that same method. My implementation of this callback is to create and send – put into the outgoing message queue to be more precise – a proxy call message. This message will contain the target bot, a unique identifier of the method (full signature), and the method arguments serialized.

After the stubs have been generated, the engine client searches through the user defined classes and searches for the engine and creates an instance of it with the previously created proxies as the constructor parameters.

Bot client The bot client finds the bot implementation that realizes the bot interface (and has a zero argument constructor) and instantizes it. Then the client uses the bot interface to create a skeleton for the remote invo-

cation. This object holds all the bot interface's methods (represented by their unique identifiers, same as on the engine client's side) mapped to the aforementioned bot instance's corresponding methods. When a proxy call message is received, this skeleton executes a lookup of the method identifier and calls the matching method of the bot and sends the result back as a call result message.