basePlatformSOMAS User Manual

implementation and instantiation of *basePlatformSOMAS*: an extensible base platform for multi-agent systems in go

Written By

Matthew Scott
matthew.scott18@imperial.ac.uk

Mikayel Suvaryan
mika.suvaryan21@imperial.ac.uk

Ana Dimoska
ana.dimoska21@imperial.ac.uk

Jeremy Pitt
j.pitt@imperial.ac.uk

Imperial College London

October 2024

Contents

1 Introduction **1**

2 Implementation **2**
2.1 *agent* Module . . . 2
2.1.1 Composition Structure . . . 2
2.1.2 `IAgent` Interface . . . 3
2.1.3 `BaseAgent` Class . . . 5
2.2 *server* Module . . . 6
2.2.1 Composition Structure . . . 6
2.2.2 `IServer` Interface . . . 7
2.2.3 `BaseServer` class . . . 9
2.3 *message* Module . . . 10
2.3.1 Composition Structure . . . 11
2.3.2 `IMessagingFunctions` Interface . . . 11
2.3.3 `IMessage` Interface . . . 13
2.3.4 `BaseMessage` class . . . 14

3 Instantiation **15**
3.1 Agent Design . . . 15
3.2 Message Design . . . 17
3.3 Server Design . . . 18
3.4 Simulator Running and Output . . . 19

1 Introduction

This user manual serves as the full documentation for the `basePlatformSOMAS` package / library. The purpose of this package is to 'democratise' the ability to produce a self-organising, multi-agent system (SOMAS), by supplying a 'vanilla' platform that can be parametrised and extended. This removes the need for a prerequisite knowledge of system design, leaving the user to implement a set of agent functions, without having to produce the relevant server to handle it, or specify an agent common language for asynchronous communication.

This manual is split into two sections. The *implementation* gives the technical detail behind the package and its constituent components, and the *instantiation* describes how the package can be extended for use in a larger, more complex simulator. We finalise the *instantiation* section with a detailed example of how a 'hello world' simulator can be produced, where users can specify a multi-agent system with agents that can asynchronously say "hello" and respond with "world".

The `basePlatformSOMAS` codebase is fully open-source, and available both on GitHub: `github.com/MattSScott/basePlatformSOMAS`, or from Go's package repository: `pkg.go.dev/github.com/MattSScott/basePlatformSOMAS/v2`.

2 Implementation

The `basePlatformSOMAS` package comprises three modules:

1. The *agent* module supplies a parametrisable `IAgent` agent interface that abstracts the core functionalities of an agent, as well as a `BaseAgent` class that implements the interface, and can be composed to gain access to a base implementation of these core functionalities and allow method overriding.

2. Similarly to the *agent* module, the *server* module supplies a parametrisable `IServer` server interface that abstracts the core functionalities of a server, as well as a `BaseServer` class that implements the interface and can be composed to gain access to a base implementation of the core methods defined in the *IServer* interface.

3. The *message* module supplies an abstract `BaseMessage` component that can be extended to allow for the passage of various data structures between agents for communication. This component implements a generic `IMessage` interface, to create a common grammar for agents.

In this section, we describe each of the aforementioned components in detail. For reference, in this manual we name all interfaces with the pattern `I<Name>` and all classes with a capitalised first letter. In composition diagrams, any composed interfaces are put in bold to distinguish them from composed data classes.

2.1 *agent* Module

This section discusses the two components of the *agent* module: the `IAgent` interface, and `BaseAgent` class, as well as the intended use of these components through a composition diagram. This package is accessible from the main package by importing:
`github.com/MattSScott/basePlatformSOMAS/v2/pkg/agent`

2.1.1 Composition Structure

This module is written with a focus on *extensibility*. We supply a base interface, `IAgent`, along with a base implementation of this interface, `BaseAgent`.

The intended use of this module is to assist in creating a parametrised agent that is suitable for a given problem specification. For example, if the problem requires that all agents must sleep, the user should add a method to the agents' interface such as *Sleep*.

To achieve this, we recommend implementing an intermediary 'tier' of agent, in the form of an `IExtendedAgent` (the interface of an extended agent), which encapsulates all of the required functionality for a particular environment (the *action space*). This `IExtendedAgent` should compose (extend) the original `IAgent` interface to 1.) gain the core functionality from this interface and 2.)

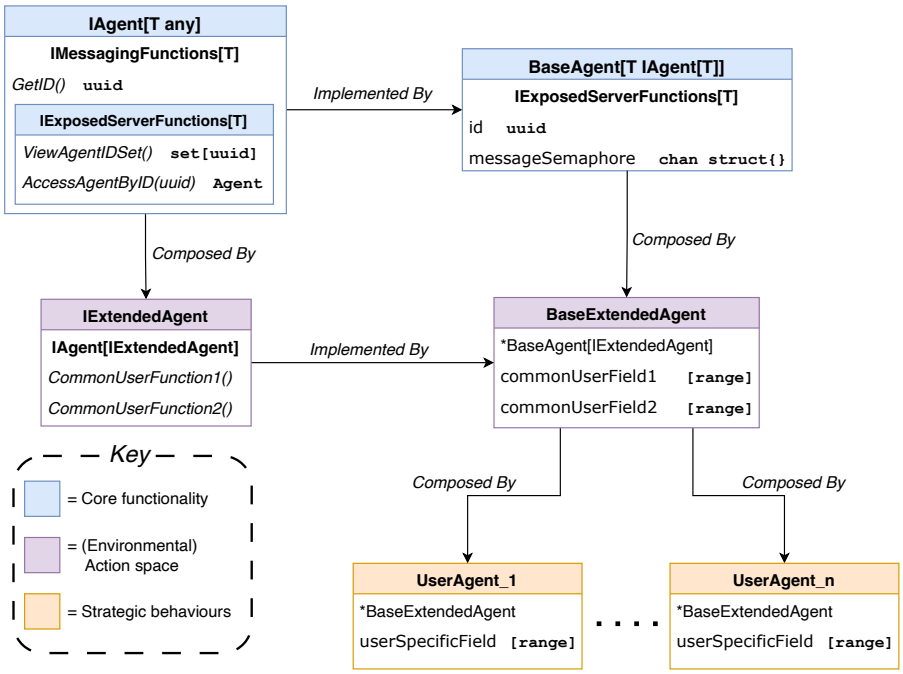

Figure 1: Composition structure diagram for producing a parametrised agent

allow the `IExtendedAgent` to be used in the *server* module. This `IExtendedAgent` can subsequently be implemented by a `BaseExtendedAgent` class, say, to provide a 'baseline' implementation of the environment-specific functions.

Finally, we envision a third 'tier' of agent, that has the 'agent specific' implementation of the environment-specific functions. This agent should have its own implementation of the `IExtendedAgent` interface, as well as any additional functions that are relevant to informing the agent's *strategic* behaviour. We illustrate this composition structure in Figure 1.

2.1.2 `IAgent` Interface

The 'vanilla' agent interface, `IAgent`, comprises three functions, all of which must be implemented by a user's custom agent. We specify this interface below:

```
package agent

type IExposedServerFunctions[T any] interface {
    // return hashset of all agent IDs
    ViewAgentIdSet() map[uuid.UUID]struct{}
    // access an arbirtrary agent by its ID
    AccessAgentByID(uuid.UUID) T
    // deliver message to agent via server invocation (hidden)
    DeliverMessage(message.IMessage[T], uuid.UUID)
    // notify that agent has completed talking phase (hidden)
    AgentStoppedTalking(uuid.UUID)
}

type IAgent[T any] interface {
    // composes message-passing capabiltites
    IMessagingFunctions[T]
    // injects server methods for accessing agents
    IExposedServerFunctions[T]
    // returns the unique ID of an agent
    GetID() uuid.UUID
}
```

Listing 1: `IAgent` interface

The first element of the `IAgent` interface is `IMessagingFunctions`. This is another interface, which `IAgent` composes. This composed interface holds the functionality for message passing, which is discussed in detail later in Section 2.3.

Secondly, we provide agents with the means of identifying, accessing and communicating with other agents in the system. Given that the server is responsible for restricting access to the underlying data structures, we leverage the *dependency injection* paradigm. This allows for a sub-interface of the server, `IExposedServerFunctions`, to be passed into the agent and allow the agents to invoke (restricted) methods on the server.

Within the `IExposedServerFunctions` interface, we supply four further functions, *ViewAgentIDSet* returns a hashset of all agent IDs that are currently alive in the system and, by calling *AccessAgentByID*, these IDs can be used to retrieve the agent instance associated with it. This can be useful for granting the ability to access methods on other agents to gain knowledge of the game state. Furthermore, *DeliverMessage* is used to send a message to an agent via the server and *AgentStoppedTalking* allows for agents to signal that they have completed messaging for a given turn. Both of these methods are formalised later as part of `IMessagingFunctions` in Section 2.3

The final element of this interface is the *GetID* method. Upon creating an agent, it is given a unique *ID* of type *UUID* (a secure alphanumeric string with low change of collision), for identification. The `BaseAgent` implements this method, and instantiates an agent with a randomised *UUID*.

Importantly, this interface is declared as a *generic* interface: i.e., `IAgent[T any]`. This allows for the interface to be parametrised with the `IExtendedAgent` interface, such that during message passing and agent access the appropriate type of agent can be retrieved.

2.1.3 `BaseAgent` Class

The `BaseAgent` class supplies a implementation of the `IAgent` interface, such that it can be composed for use in a `BaseExtendedAgent`, say. This class stores both the *ID* of the agent, which can be retrieved using its implementation of the *GetID* function, and a semaphore used to limit the number of asynchronous messages an agent can simultaneously handle. The latter limits thread spawnage, thereby protecting the simulator from memory oversaturation. We formalise this class below:

```
1  package agent
2
3  type BaseAgent[T IAgent[T]] struct {
4      // injects base agent with core methods defined on server
5      IExposedServerFunctions[T]
6      // stores ID of agent
7      id                          uuid.UUID
8      // limits maxmimum number of messages an agent can handle
9      messageLimiterSemaphore chan struct{}
10 }
11
12 // generates a new BaseAgent instance with a randomised ID
13 func CreateBaseAgent[T IAgent[T]] \\
14     (serv IExposedServerFunctions[T]) *BaseAgent[T] {
15     if serv == nil {
16         panic("Nil interface passed to CreateBaseAgent")
17     }
18     return &BaseAgent[T]{
19         IExposedServerFunctions: serv,
20         id:                      uuid.New(),
21         messageLimiterSemaphore: make(chan struct{}, \\
22                 serv.GetAgentMessagingBandwidth()),
23     }
24 }
```

Listing 2: `BaseAgent` class

When composing / extending the `BaseAgent` class, it is recommended to pass a pointer to this class to the composing class (see Figure 1). To achieve this, the *agent* package supplies the *CreateBaseAgent* function, which returns (a pointer to) an instantiated `BaseAgent` class, with a randomly generated *ID* and semaphore for limiting the number of messages that can be asynchronously handled by a single agent.

Similarly to the `IAgent` interface, this class is created with *generics* to allow for parametrisation. This class is able to be parametrised with any interface that composes / extends the `IAgent` interface, such that any composing classes can be used in the server with a common type, and also such that messaging can be performed with `ExtendedAgents`. As such, during instantiation the most recent 'parent' interface should be passed as a parameter (the `IExtendedAgent` in Figure 1, say).

The methods for accessing and interacting with other agents in the system are controlled, and given access by the *server* package. As such, the `BaseAgent`

is instantiated with an object implementing the `IExposedServerFunctions` interface (shown in Figure 1). Therefore, when the server object is created, it should be passed into the constructor, thereby granting the agent the respective methods from this interface.

2.2 *server* Module

This section discusses the two components of the *server* package: the `IServer` interface, and `BaseServer` class, as well as the intended use of these components through a composition diagram. This package is accessible from the main module by importing:
`github.com/MattSScott/basePlatformSOMAS/v2/pkg/server`

2.2.1 Composition Structure

Similarly to the implementation of the the *agent* package in Section 2.1, we design a set of 'core functions' that can be expanded upon for producing a parametrised server that is relevant to the environment-specific application of the module.

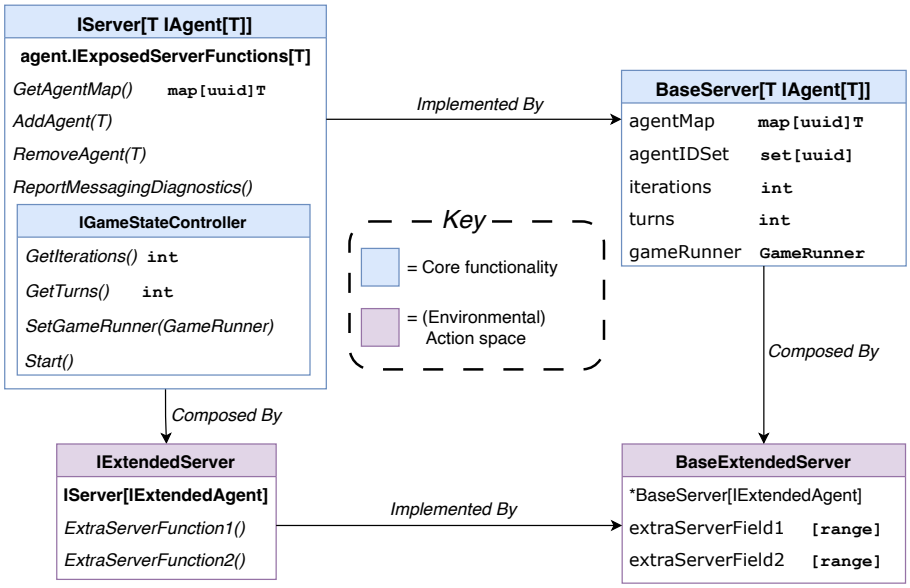

Figure 2: Composition structure diagram for producing a parametrised server

We supply an `IServer` interface, which encompasses the core functionality of a multi-agent system simulator: updates to the physical, shared *environmental state* and updates to the internal *agent state*. We produce a `BaseServer` class

that implements this interface, thereby providing a (basic) implementation of these functions which can be overriden through composition.

Unlike with the *agent* package, we don't envision a third 'tier' of composition. Whereas there was previously a need for agents to be given 'strategic' behaviours, requiring composition of an intermediate class, a single server is responsible for operation on all agents. As such, there is only a second 'tier' of composition, to produce an environmentally-specific server that adds additional functionality to the system, such as *RunSleepPhase*, continuing with the example of an *agent.Sleep* function, from before.

2.2.2 `IServer` Interface

The `IServer` interface defines all of the core functions that a server should carry out. We consider two kinds of functions: 1.) functions which affect the physical, shared *environmental state* and 2.) functions which affect the internal *agent state*.

We provide a supplementary interface for handling the flow of the physical game state which we call the `GameRunner`. This interface is formalised in Table 1

Method	**Range**
RunStartOfIteration(int)	**void**
RunTurn(int, int)	**void**
RunEndOfIteration(int)	**void**

Table 1: `GameRunner` interface

`GameRunner` is attached to the server instance in *SetGameRunner* for use in the *Start* method - the two of which define the structure for the simulation. Here, we allow users to provide the ordering of 'high frequency' behaviours that run every turn, with *RunTurn* (accepting as an argument the current turn and iteration), as well as the 'low frequency' behaviours that run at the start and end of each iteration, with *RunStartOfIteration* and *RunStartOfIteration*, respectively, both taking the current iteration as an argument. Continuing with the 'sleep' analogy, we can imagine *RunTurn* being defined as running *RunBrushTeethPhase* and then *RunSleepPhase*, say, whereas *RunStartOfIteration* may run *RunMakeBedPhase*, and *RunEndOfIteration* may run *RunStripBedPhase*.

Since the *Start* method is already implemented by the `BaseServer`, we need a means of accessing the user-defined `GameRunner` methods from within the base platform (i.e., passing information from child to parent). Therefore, users should call *SetGameRunner* after creation of the scenario-specific server. This then binds the `GameRunner` methods defined in the `ExtendedServer` onto the `BaseServer`. For completeness, the full *Start* method is shown in Listing 3.

```go
package server

func (serv *BaseServer[T]) Start() {
    serv.checkGameRunner() // panic if GameRunner not set
    for i := 0; i < serv.iterations; i++ {
        serv.gameRunner.RunStartOfIteration(i)
        for j := 0; j < serv.turns; j++ {
            serv.handleStartOfTurn()
            serv.gameRunner.RunTurn(i, j)
            serv.handleEndOfTurn()
        }
        serv.gameRunner.RunEndOfIteration(i)
    }
}
```

Listing 3: *Start* method in `BaseServer`

We also supply a second interface, `IAgentOperations`, that (for readability) is composed by the `IServer` interface to supply functionality for updating the *agent state*. In this server implementation, agents are stored in a *hashmap*, mapping an agent's (UU)ID to a (pointer to) the agent. This allows for constant-time insertion, deletion and retrieval of the agent based on its ID. As such, the method *GetAgentMap* returns this map. Supplementing this are the methods *AddAgent* and *RemoveAgent*, which add and remove an agent object from the map, respectively.

Due to the (slightly unpredictable) nature of the asynchronous messaging, we also provide the *ReportMessagingDiagnostics* method. This, when called, toggles the `DiagnosticsEngine` in the server (omitted from the user manual, for brevity) which gives details on the percentage of messages that are correctly delivered during each round of simulation.

Finally, given that the server controls access to, and communication between, the agents in the multi-agent system, we also provide the *IExposedServerFunctions* interface, which can be injected into the `BaseAgent` to allow it to invoke methods on the server (see *CreateBaseAgent* in Listing 2).

```
package server

type IAgentOperations[T baseagent.IAgent[T]] interface {
    // gives access to the agents in the simulator
    GetAgentMap() map[uuid.UUID]T
    // adds an agent to the server
    AddAgent(agentToAdd T)
    // removes an agent from the server
    RemoveAgent(agentToRemove T)
}

type IServer[T baseagent.IAgent[T]] interface {
    agent.IExposedServerFunctions[T]
    IAgentOperations[T]
    // access number of iterations in simulator
    GetIterations() int
    // access number of turns in simulator
    GetTurns() int
    // attach a game-runner to the server object
    SetGameRunner(GameRunner)
    // begins simulator
    Start()
}
```

Listing 4: `IServer` interface

As with the agent interface, this server is instantiated with the generic type `T`. This allows for the server to be parametrised with the agents' 'parent' interface (`IExtendedAgent`, in Section 2.1), to populate the server with the relevant type of agent (and therefore provide access to any additional functions gained from composing the `IAgent` interface).

2.2.3 `BaseServer` class

The `BaseServer` class supplies a implementation of the `IServer` interface, such that it can be composed for use in a `BaseExtendedServer`, say. This class stores the *AgentMap*, which can be retrieved using its implementation of the *GetAgentMap* function, and the number of iterations and turns the simulator should run for, which are accessible through the `GetIterations` and *GetTurns* methods, respectively.

```go
package server

type BaseServer[T IAgent[T]] struct {
    agentMap map[uuid.UUID]T
    agentIDSet map[uuid.UUID]struct{}
    iterations int
    turns int
    gameRunner GameRunner
    turnTimeout time.Duration
    agentMessagingBandwith int
    reportMessagingDiagnostics bool
}

// generate a base server instance
func CreateServer[T IAgent[T]] (iterations, turns int,
    turnMaxDuration time.Duration, messageBandwidth int) ->
                    *BaseServer[T] {
    return &BaseServer[T]{
        agentMap: make(map[uuid.UUID]T),
        agentIDSet: make(map[uuid.UUID]struct{}),
        iterations: iters,
        turns: turns,
        gameRunner: nil, // must be set before running Start()
        turnTimeout: turnMaxDuration,
        messageBandwidth: messageBandwidth,
        reportMessagingDiagnostics: false,
    }
}
```

Listing 5: `BaseServer` class

Creating a server instance requires calling the *CreateServer* function which accepts four arguments. The first two arguments, *iterations* and *turns* define the number of iterations and rounds, respectively, a simulator should be run for.

turnMaxDuration is used to facilitate the *asynchronous* message passing abilities of the server (later described in Section 2.3) by setting a maximal duration that that the server will wait, each round, for the agents to notify it that they are finished with the messaging session. This is a kind of 'failsafe' to both give agents enough time to handle the message processing (and therefore not advance the round too quickly), but not wait infinitely long, in the event that an agent might never say that they're ready to continue.

Finally, *messageBandwidth* defines the (maximum) number of messages that an agent can simultaneously process. Due to the asynchronous nature of the messaging system, this involves thread spawanage, which may saturate the simulator and lead to memory leaks if not appropriately maintained. As such, this bandwidth will allow messages to be dropped (i.e., not spawn a goroutine for the agent's handler) if this limit has been reached.

2.3 *message* Module

The final component we supply is the *message* module, which allows for the specification of complex message types and facilitates communication between

agents. This module is accessible from the main package by importing:
`github.com/MattSScott/basePlatformSOMAS/v2/pkg/message`

2.3.1 Composition Structure

As with the *agent* and *server* modules, the primary focus of this module is extensibility. We supply an architecture that allows for the specification of multiple, complex message types that are compatible with any agent, in any server defined under this package.

We continue with the general composition structure discussed in the previous two modules, where we supply an `IMessage` interface to encompass the core functionality of a message, and a `BaseMessage` class that gives a base implementation of this interface which can be composed in more complex, user defined messages.

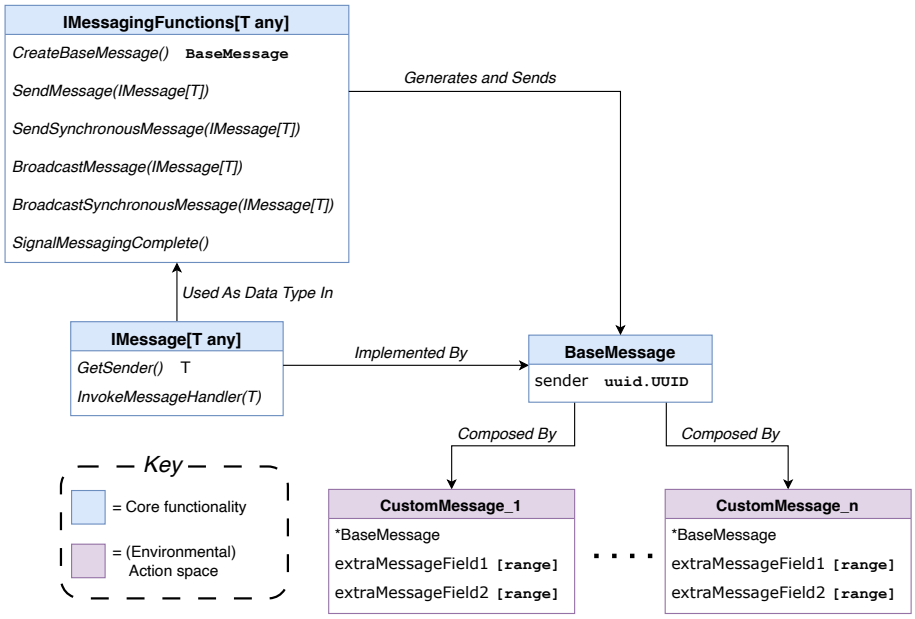

Figure 3: Composition structure diagram for producing various message types

In Figure 3, we formalise the `IMessagingFunctions` interface declared earlier in Figure 1.

2.3.2 `IMessagingFunctions` Interface

The `IMessagingFunctions` interface contains the set of methods needed for facilitating agent-to-agent communication, as well as the creation of a `BaseMessage`, for composition in more complex grammars. We formalise this interface

in Listing 6

```go
package agent

type IMessagingFunctions[T any] interface {
    // allows for creation of a base message
    CreateBaseMessage() message.BaseMessage
    // allows for sending a message to a single recipient
    SendMessage(message.IMessage[T], uuid.UUID)
    // allows for sending a message to a single recipient
        synchronously
    SendSynchronousMessage(message.IMessage[T], uuid.UUID)
    // allows for sending an async message across the entire
        system
    BroadcastMessage(message.IMessage[T])
    // allows for sending a sync message across the entire
        system
    BroadcastSynchronousMessage(message.IMessage[T])
    // signals that agent has completed messaging for the round
    SignalMessagingComplete()
}
```

Listing 6: `IAgentMessenger` interface

The first method supplied in this interface is *CreateBaseMessage*, which returns a `BaseMessage` object for use in composition for more complex message types. We can envision an `ExtendedAgent` implementing a *CreateExtendedMessage* method, which itself uses the composed *CreateBaseMessage* method to pass in the `BaseMessage` object to the constructor.

Secondly, we provide the *SendMessage* method for generic *asynchronous* agent-to-agent communication. This method takes, as arguments, the message to be sent (which must implement `IMessage`) and the ID of the recipient agent. Internally, this method invokes the *DeliverMessage* method of the `BaseServer`, which spawns a thread for the recipient agent's handler method.

As well as this, we have the *SendSynchronousMessage* method, which is (predictably) the *synchronous* version of *SendMessage*. Instead of spawning a thread for the recipient agent's handler, the method is invoked directly on the main thread (thereby slowing down the progression of the main loop). This method is useful for when either the handling of a message needs to be guaranteed, or the message passing should happen in a synchronous manner - before the start of a simulator phase. Conversely, if a message doesn't necessarily need to be handled, or can safely run in parallel to the main thread, the *SendMessage* method can be used. The latter gives a significant increase in simulation speed-up, at the cost of certainty of message handling.

BroadcastMessage is a 'quality of life' method used to send a message to all other agents in the system. Internally, this iteratively calls *SendMessage* for every other agent ID. Similarly, *BroadcastSynchronousMessage* is used as the synchronous equivalent, where *SendSynchronousMessage* is called internally, instead.

Finally, *SignalMessagingComplete* is used to signify that an agent has com-

pleted all of their messaging for a round. With asynchronous messaging, it is important to leave a sufficient amount of time, each round, for agents to complete their sending and handling of messages. It is possible that the main thread will advance before the agents' coroutines have had time to complete, so consideration must be taken for the overheads of spawning threads and invoking the message handlers. By calling *SignalMessagingComplete*, agents can inform the main thread that it can advance to the next turn safely.

2.3.3 `IMessage` Interface

The `IMessage` interface comprises the core functionality for producing a customised message class. This interface has three methods, which all extended messages must implement: a method for accessing the *sender* of a message, a method for accessing the *recipients* of a message and a method for telling the agent how to handle the specific type of message:

```
1  package message
2
3  // base interface structure used for messages
4  // can be composed for more complex message structures
5  type IMessage[T any] interface {
6      // returns the sender of a message
7      GetSender() T
8      // calls the appropriate messsage handler method on the
           receiving agent
9      InvokeMessageHandler(T)
10 }
```

Listing 7: `IMessage` interface

The *GetSender* method returns the generic type used to parametrise the `IMessage` interface. As such, by parametrising it with `IExtendedAgent`, say, any agents that access the sender of this message object will have access to the (exported) functions declared in the `IExtendedAgent` interface. This can help with informing how the message should be handled, since more information about the sender may be required for strategic behaviours.

The *InvokeMessageHandler* method is used to implement the visitor design pattern, such that specific agent methods can be called on a particular message object. We envision that the `IExtendedAgent` interface adds all of the appropriate message handler methods, as in Figure 4.

Following this, the custom message object can then implement the *InvokeMessageHandler* function to call the appropriate handler from the agent class, passing the message itself as a parameter. Note that `BaseMessage` does not implement *InvokeMessageHandler*, therefore this is necessary to ensure that any class that composes `BaseMessage` implements the `IMessage` interface.

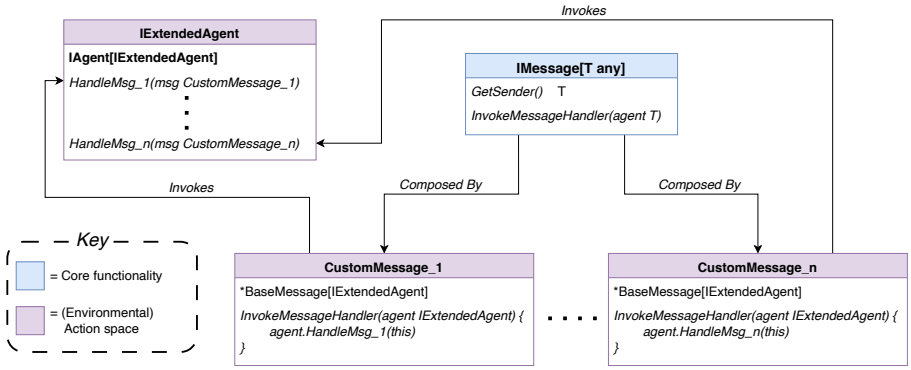

Figure 4: Visitor design pattern for handling various message types

2.3.4 `BaseMessage` class

The final component in the *messaging* package is the `BaseMessage` class. This class provides a base implementation of the `IMessage` interface, which can be used in composition for more complex message types.

This class contains a single field, `sender`, which stores the ID of sender of the message. We envision that this agent object is then accessible through the *AccessAgentByID* method defined on the agent class in Section 2.1.

```
1  package message
2
3  // base interface structure used for message - can be composed
       for more complex message structures
4  type IMessage[T any] interface {
5      // returns the sender of a message
6      GetSender() uuid.UUID
7      // calls the appropriate message handler method on the
           receiving agent
8      InvokeMessageHandler(T)
9  }
10
11 // new message types can extend this
12 type BaseMessage struct {
13     Sender uuid.UUID
14 }
15
16 func (bm *BaseMessage) GetSender() uuid.UUID {
17     return bm.Sender
18 }
```

Listing 8: `BaseMessage` class

3 Instantiation

In this section, we demonstrate how this package can be used to produce a 'Hello World' simulator, using all three of the modules described in Section 2. This simulator comprises firstly a single type of agent: the `HelloWorldAgent`, used for the sending and handling of messages. This agent design is described in Section 3.1. Secondly, we provide two message types: `HelloMessage` and `WorldMessage`, used for sending 'Hello' and 'World' respectively as part of the agent common language (ACL). These messages types are described in Section 3.2. Finally, we provide a server: the `HelloWorldServer`, used to run the simulator and control the gamestate of the `HelloWorldAgents`. This server design is described in Section 3.3. Having formalised this simulator, we then provide an example main file for running the program, as well as an expected console output in Section 3.4.

3.1 Agent Design

Firstly, in line with the inheritance structure described in Sections 2.1 and 2.2, we specify a 'parent' agent that holds the common methods needed for the "hello world" functionality and is used to parametrise the messaging and server interfaces. We call this the `HelloWorldAgent`.

```go
package main

type IHelloWorldAgent interface {
    agent.IAgent[IHelloWorldAgent]
    CreateHelloMessage() *HelloMessage
    HandleHelloMessage(HelloMessage)
    CreateWorldMessage() *WorldMessage
    HandleWorldMessage(WorldMessage)
}

type HelloWorldAgent struct {
    *agent.BaseAgent[IHelloWorldAgent]
}

func (hwa *HelloWorldAgent) HandleHelloMessage(msg HelloMessage) {
    fmt.Printf("%s said: 'Hello'\n", hwa.GetID())
    response := WorldMessage{hwa.CreateBaseMessage()}
    hwa.SendMessage(&response, msg.Sender)
}

func (hwa *HelloWorldAgent) HandleWorldMessage(msg WorldMessage) {
    fmt.Printf("%s responded: 'World'\n", hwa.GetID())
    hwa.SignalMessagingComplete()
}

// inject methods from server into agent
func CreateHelloWorldAgent(serv agent.IExposedServerFunctions[IHelloWorldAgent]) IHelloWorldAgent {
    return &HelloWorldAgent{
        BaseAgent: agent.CreateBaseAgent(serv),
    }
}
```

Listing 9: `IHelloWorldAgent` class

The `HelloWorldAgent` must encapsulate all the functionality for sending and handling `HelloMessage` and `WorldMessage` instances. We therefore propose the `IHelloWorldAgent` interface, which composes the `IAgent` interface. We also define two additional methods: *HandleHelloMessage* and *HandleWorldMessage*.

HandleHelloMessage is used for the handling of the `HelloMessage`, which logs 'Hello' (and the agent ID) to console and responds to the sender of the message with a `WorldMessage`. *HandleWorldMessage* behaves similarly, handling the `WorldMessage` by logging 'World' (and the agent ID) to the console, and then signifying that the agent has finished their messaging for that turn. This code is formalised in Listing 9.

We also attach the constructors for these message types to the agent, in the form of *CreateHelloMessage* and *CreateWorldMessage*. These are formalised later in Listings 10 and 11.

3.2 Message Design

We provide two message types for distribution across the system: a `HelloMessage`, used for the sending of 'Hello' and a `WorldMessage`, used for the sending of 'World'.

In line with the inheritance structure described in Section 2.3, all messaging types should implement the `IMessage` interface by both composing the `BaseMessage`, and implementing the *InvokeMessageHandler* method. The `HelloMessage` should therefore invoke the *HandleHelloMessage* method from the `HelloWorldAgent`

```
1  package main
2
3  type HelloMessage struct {
4      message.BaseMessage
5  }
6
7  func (hwa *HelloWorldAgent) CreateHelloMessage() *HelloMessage {
8      return &HelloMessage{
9          // embed base message
10         BaseMessage: hwa.CreateBaseMessage(),
11     }
12 }
13
14 func (d HelloMessage) InvokeMessageHandler(ag IHelloWorldAgent)
       {
15     ag.HandleHelloMessage(d)
16 }
```

Listing 10: `HelloMessage` class

and the `WorldMessage` should invoke the *HandleWorldMessage* method.

```
1  package main
2
3  type WorldMessage struct {
4      message.BaseMessage
5  }
6
7  func (hwa *HelloWorldAgent) CreateWorldMessage() *WorldMessage {
8      return &WorldMessage{
9          // embed base message
10         BaseMessage: hwa.CreateBaseMessage(),
11     }
12 }
13
14 func (d WorldMessage) InvokeMessageHandler(ag IHelloWorldAgent)
       {
15     ag.HandleWorldMessage(d)
16 }
```

Listing 11: `WorldMessage` class

3.3 Server Design

The final component for producing the `HelloWorldSimulator` is the server. The `BaseServer` provided in the package already supplies methods for running asynchronous messaging and synchronising the start and ends of rounds. Therefore, for the `HelloWorldSimulator`, all that is required is to compose the `BaseServer` struct, in a `HelloWorldServer`.

This user-created server must implement `GameRunner`, however, in line with Section 2.2.2. As such, we also attach the *RunTurn* method, used for 'seeding' the conversation: each agent broadcasts a `HelloMessage` to kick off the 'Hello World' back-and-forth. Furthermore, to properly implement the `GameRunner` interface, we must also define the *RunStartOfIteration* and *RunEndOfIteration* methods. For simplicity, we just log the current iteration to the console.

```
1  package main
2
3  type IHelloWorldServer interface {
4      server.IServer[IHelloWorldAgent]
5  }
6
7  // access methods from the server package
8  type HelloWorldServer struct {
9      *server.BaseServer[IHelloWorldAgent]
10 }
11
12 // have all agents 'kick-off' the conversation
13 func (serv *HelloWorldServer) RunTurn(i, j int) {
14     fmt.Printf("Running iteration %v, turn %v\n", i+1, j+1)
15     for _, ag := range serv.GetAgentMap() {
16         // compose base message
17         msg := HelloMessage{ag.CreateBaseMessage()}
18         ag.BroadcastMessage(&msg)
19     }
20 }
21
22 // implement GameRunner
23 func (serv *HelloWorldServer) RunStartOfIteration(iteration int)
       {
24     fmt.Printf("Starting iteration %v\n", iteration+1)
25     fmt.Println()
26 }
27
28 // implement GameRunner
29 func (serv *HelloWorldServer) RunEndOfIteration(iteration int) {
30     fmt.Println()
31     fmt.Printf("Ending iteration %v\n", iteration+1)
32 }
```

Listing 12: `HelloWorldServer` class

Finally we must instantiate the `HelloWorldServer` and inject the agents. This can all be achieved within a custom constructor. Here, we compose the `BaseServer`, passing in the number of turns, iterations, the maximum round duration and the maximum agent bandwidth. We then manually inject the `Hel-`

loWorldAgents. Finally, we set the GameRunner interface to the HelloWorldServer object. This enables the BaseServer to have knowledge of the composing class, and call the appropriate 'runner' methods.

```
1  package main
2
3  func CreateHelloWorldServer(numAgents, iterations, turns int,
       maxDuration time.Duration, agentBandwidth int) *
       HelloWorldServer {
4      serv := &HelloWorldServer{
5          BaseServer: server.CreateBaseServer[IHelloWorldAgent](
               iterations, turns, maxDuration, agentBandwidth),
6      }
7      for i := 0; i < numAgents; i++ {
8          serv.AddAgent(CreateHelloWorldAgent(serv))
9      }
10     serv.SetGameRunner(serv)
11     return serv
12 }
```

Listing 13: HelloWorldServer constructor

3.4 Simulator Running and Output

Having specified all of these components, we can now produce a working simulator through creation of a main file. Here, we create the simulator with four agents, running for one iteration, with one turn (for simplicity in logging the output). We give agents a maximum duration of 1ms for signalling the end of their messaging phase, and the ability to simultaneously handle 100 messages.

```
1  package main
2
3  func main() {
4      serv := CreateHelloWorldServer(4, 1, 1, time.Millisecond,
           100)
5      // toggle message diagnostics
6      serv.ReportMessagingDiagnostics()
7      // begin simulation
8      serv.Start()
9  }
```

Listing 14: Main file for simulator

By running this main file with `go run .` the output in Figure 5 is produced (with variation in the order of message passing and agent ID). This demonstrates that the simulator is running as expected, with four agents interacting and sharing messages between one another.

```
mss2518@IC-TLF4XP94VJ SOMAS_Base_Platform % go run .
Starting iteration 1

Running iteration 1, turn 1
6ddfd8ef-c331-4e6c-9950-51b5d39e16b0 said: 'Hello'
6ddfd8ef-c331-4e6c-9950-51b5d39e16b0 said: 'Hello'
6ddfd8ef-c331-4e6c-9950-51b5d39e16b0 said: 'Hello'
d378e786-b65d-41b2-a133-437804da358a responded: 'World'
d378e786-b65d-41b2-a133-437804da358a said: 'Hello'
d378e786-b65d-41b2-a133-437804da358a said: 'Hello'
6ddfd8ef-c331-4e6c-9950-51b5d39e16b0 responded: 'World'
c5e26a5c-8845-4f2d-9d98-5a1b9b0f0746 responded: 'World'
c5e26a5c-8845-4f2d-9d98-5a1b9b0f0746 said: 'Hello'
c41f587e-d1b1-4ce8-89d0-a5cc12239f73 said: 'Hello'
d378e786-b65d-41b2-a133-437804da358a said: 'Hello'
c5e26a5c-8845-4f2d-9d98-5a1b9b0f0746 responded: 'World'
6ddfd8ef-c331-4e6c-9950-51b5d39e16b0 responded: 'World'
c41f587e-d1b1-4ce8-89d0-a5cc12239f73 responded: 'World'
c5e26a5c-8845-4f2d-9d98-5a1b9b0f0746 said: 'Hello'
c41f587e-d1b1-4ce8-89d0-a5cc12239f73 said: 'Hello'
d378e786-b65d-41b2-a133-437804da358a responded: 'World'
c41f587e-d1b1-4ce8-89d0-a5cc12239f73 said: 'Hello'
d378e786-b65d-41b2-a133-437804da358a responded: 'World'
6ddfd8ef-c331-4e6c-9950-51b5d39e16b0 responded: 'World'
c5e26a5c-8845-4f2d-9d98-5a1b9b0f0746 responded: 'World'
c5e26a5c-8845-4f2d-9d98-5a1b9b0f0746 said: 'Hello'
c41f587e-d1b1-4ce8-89d0-a5cc12239f73 responded: 'World'
c41f587e-d1b1-4ce8-89d0-a5cc12239f73 responded: 'World'
100.000000% of messages successfully sent (21 delivered, 0 dropped)
100.000000% of agents successfully ended messaging (4 ended, 4 total)

Ending iteration 1
mss2518@IC-TLF4XP94VJ SOMAS_Base_Platform %
```

Figure 5: Output from `HelloWorldSimulator`