Brilliant Students vs. Zombie Professors

Protocol Definition

Overview

This document defines the communication protocols for an interactive multi-player game, called the *Brilliant Students vs. Zombie Professors* (or *BSvZP)*. The system will consist of a variety of software components (processes) that will communicate with each other, namely: *Game (G), Playing Field (PF)*, *Clock Tower (CT)*, *Brilliant Students (BS)*, *Excuse Generators (EG)*, *Whining Spinners (WS)*, *Zombie Professors (ZP)*, a *Monitor (M)*, and *Referee* (*R*). The next section outlines the various types of conversations that may occur between these components and the general communication patterns that these conversations follow. It also defines messages that the protocols involves. The section after that defines how the software components must encode and decode them so they understand each other.

Conversations, Communication Patterns, and Messages

Table 1 lists the possible types of conversations involved in this system, along with which component initiates the conversation, other components involved, and general communication pattern. The communication pattern defines the possible message sequences in both normal and abnormal conditions.

The different communication patterns the protocols in Table 1 include *One-way Send*, *Request-Reply*, *Unreliable Multicast*, *Reliable Multicast*, *Ongoing Update Stream*, and *WebMethod Innovation*. Each communication patterns, each the last, use the messages that come from the list of specialization of the *Request* and *Reply* class in Figure 1. Figures 2 - 9 illustrate the possible message sequences for the *One-way Send*, *Request-Reply*, and *3-party XYZ* patterns.

Table 1 – Conversations and Protocols for the *BSvZP*

(Note: protocols in gray will be implemented later.)

Protocol / Conversation	**Initiator**	**Other Participants**	**Communication Pattern, Messages, and Semantics**
Game Registration *Registers a new game so others can discover it.*	*G*	*GR*	*WebMethod Invocation* of the *GR.GameRegister* method, which will take the game's label, end point, and status as parameters. This method will

			return a unique game id and a block of id's that can be assigned to agents that join the game.
Game Status Change	*G*	*GR*	*WebMethod Invocation* of the *GR.GameStatusChange* method, which will take the game's registry id and the new status, and optionally the name of the winner if the status if "Completed".
Get Games	*BS, EG, or WS*	*GR*	*WebMethod Invocation* of the *GR.GetGames* method, which will take a status as a parameter and return an array of all of the games (GameInfo objects) with that status.
Join Game	*BS*, *EG*, *WS*, *ZP*, or *R*	*G*	*Request-Reply*, with *JoinGame* and AckNak as request and reply messages, where • The *AgentInfo* attrAgentibute of the *JoinGame* request contains a *ComponentInfo* object and that object with only the AgentType specified • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* in the *AckNak* message will be a completed *ComponentInfo* object. • If the *Status* of the *AckNak* message is *Failure*, the agent could not join the game for some reason and the *Message* of the *AckNak* contains the specific reason or error message.
Remove Agent	*R*	*G*	*Request-Reply*, with *RemoveComponent* and *AckNak* messages • The *ComponentId* attribute is the *RemoverComponent* is the identifier of the component to remove from the playing field. • If the *Status* of the *AckNak* message is *Success*, then the *IntResult* of the *AckNak* is the component's Id. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Start Game	*G*	*BS, EG, WS, ZP, or R*	Reliable Multicast, with *StartGame* as the initial message, a *ReadyReply* as agent reply, and *AckNak* as the last message.
End Game	*R* *G*	*G* *BS*, *EG*, *WS*, or *ZP*	Unreliable Multicast, with *EndGame* as the message. The *EndGame* message will contain a list of winners.
Get Configuration	*BS*, *EG*, *WS*, *ZP*, *M*, or *R*	*G*	*Request-Reply*, with *GetResource* and *ConfigurationReply* messages, where • The *GetType* in the *GetResource* message is *Game Configuration*. • If the *Status* of the *ConfigurationReply* message is *Success*, then the *Config* object of the *ConfigurationReply* is a *Configuration* object.

			• Otherwise, the request failed and the *Config* is null.
Get Playing Field Layout	*BS*, *EG*, *WS*, *ZP*, or *R*	*G*	*Request-Reply*, with *GetResource* and *PlayingFieldReply* messages, where • If the *Status* of the *PlayingFieldReply* message is *Success*, then the *Layout* of the *PlayingFieldReply* is a *PlayingFieldLayout* object. • Otherwise, the request failed and the *PlayingFieldLayout* is null.
Get Brilliant Student List	*BS*, *EG*, *WS*, *ZP*, or *R*	*G*	*Request-Reply*, with *GetResource* and AgentListReply messages, where • The *GetType* in the *GetResource* message is *Brilliant Student List*. • If the *Status* of the AgentListReply message is *Success*, then the *AgentList* of the AgentListReply is a *ComponentList* object containing *ComponentInfo* objects about all *BrilliantStudent* objects currently on the playing field. • Otherwise, the request failed and the *AgentList* is null.
Get Excuse Generator List	*BS*, *EG*, *WS*, *ZP*, or *R*	*G*	*Request-Reply*, with *GetResource* and AgentListReply messages, where • The *GetType* in the *GetResource* message is *Excuse Generator List*. • If the *Status* of the AgentListReply message is *Success*, then the *AgentList* of the AgentListReply is a *ComponentList* object containing *ComponentInfo* objects about all *ExcuseGenerator* objects currently on the playing field. • Otherwise, the request failed and the *AgentList* is null.
Get Whining Spinner List	*BS*, *EG*, *WS*, *ZP*, or *R*	*G*	*Request-Reply*, with *GetResource* and AgentListReply messages, where • The *GetType* in the *GetResource* message is *Excuse Generator List*. • If the *Status* of the AgentListReply message is *Success*, then the *AgentList* of the AgentListReply is a *ComponentList* object containing *ComponentInfo* objects about all *WhiningSpinner* objects currently on the playing field. • Otherwise, the request failed and the *AgentList* is null.
Get Zombie Professor List	*BS*, *EG*, *WS*, *ZP*, or *R*	*G*	*Request-Reply*, with *GetResource* and AgentListReply messages, where • The *GetType* in the *GetResource* message is *Excuse Generator List*. ***If the Status of the AgentListReply message is Success, then the AgentList of the AgentListReply is a ComponentList object containing ComponentInfo objects about all ZombieProfessor objects currently on the playing field.***

			• Otherwise, the request failed and the *AgentList* is null.
Get Excuse	*BS*	*EG*	*Request-Reply*, with *GetResource* and *ResourceReply* messages, where • The *GetType* in the *GetResource* message is *Excuse*. • If the *Status* of the *ResourceReply* message is *Success*, then the *Resource* of the *ResourceReply* is an *Excuse* object. • Otherwise, the request failed and the *Resource* is null.
Get Whining Twine	*BS*	*EG*	*Request-Reply*, with *GetResource* and *ResourceReply* messages, where • The *GetType* in the *GetResource* message is *Whining Twine*. • If the *Status* of the *ResourceReply* message is *Success*, then the *Resource* of the *ResourceReply* is a *Whining Twine* object. • Otherwise, the request failed and the *Resource* is null.
Send Out Time Tick	*CT*	*BS*, *EG*, *WS*, or *ZP*	*Unreliable Multicast*, with *TickMessage* as the messages.
Validate Tick	*PF*	*CT*	*Request-Reply*, with *ValidateTick* and AckNak messages, where • The *ComponentId* attribute in *ValidateTick* message is the identify of the component that wants to use the *Tick* • If the *Status* of the *AckNak* message is *Success*, then the tick is valid. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Move	*BS* or *ZP*	*G, CT*	*Request-Reply*, with *Move* and AckNak messages, where • The *ComponentId* attribute in the *Move* message is the identify of the component that wants to use the *Tick* • The *ToSquare* attribute in the *Move* message is where the agent (BS or ZP) wants to move • The *EnablingTick* attribute in the *Move* message is a valid Tick that agent hasn't used for any other purpose. • If the *Status* of the *AckNak* message is *Success*, then the move took place. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Throw Bomb	*BS*	*G*	*Request-Reply*, with *Throw Bomb* and AckNak messages, where • The *ComponentId* attribute in the *Throw Bomb* message is the identify of the component that wants to throw the bomb. • The *Bomb* attribute in the *Throw Bomb* message has to be bomb containing at least one *Excuse* and one *Whining Twine*

			• The *TowardsSquare* attribute in the *Throw Bomb* message represent the target of the bomb. If the bomb doesn't have enough *Whining Twine* to go that distance, it will fail short, in some other square. • The *EnablingTick* attribute in the *Move* message is a valid Tick that agent hasn't used for any other purpose. • If the *Status* of the *AckNak* message is *Success*, then the bomb was thrown (but possibly not all the way to the target. The *ObjResult* attribute contains a *Square* object that describes where the bomb landed. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Eat	*ZP*	*G*	*Request-Reply*, with *Eat* and *AckNak* messages, where • The *ZombieId* attribute in the Eat message is the identity of the zombie that wants to eat something else. • The *TargetId* attribute is the identity of the target agent that the zombie wants to eat. • If the *Status* of the *AckNak* message is *Success*, then the *Eating* took place. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Change Strength	*G*	*BS*, *EG*, *WS*, or *ZP*	*Request-Reply*, with *ChangeStrength* and *AckNak* messages, where • The *DeltaValue* attribute is the delta value that needs to be apply to the receiving agent's current strength. • If the *Status* of the *AckNak* message is *Success*, then the operation was successful. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Collaborate	*BS*	*BS*	*Request-Reply*, with *Collaborate* and *AckNak* messages, where • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* attribute contains *ComponentInfo* object that describes the current target of the receiving agent. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
GetStatus	*M* or *R*	*BS, EG, WP,* or	*Request-Reply*, with *GetStatus* and *StatusReply* messages, where

		ZB	• If the *Status* of the *StatusReply* message is *Success*, then the *StatusInfo* attribute contains *ComponentInfo* object that describes the current status of the receiving agent. • Otherwise, the *ComponentInfo Message* of the *StatusReply* is null.
Agent Update Stream			*Ongoing Update Stream*. With *StartUpdateStream*, *AckNak*, *AgentListReply*, and *EndUpdateStream* as message. • An agent will send the game a *StartUpdateStream* message to start the update stream. • The game will send back an *AckNak* message with the *Status* if the stream is ready to start. • After that, the game will send AgentListReply messages periodically, until the agent sends a *StopUpdateStream* message.
Exit Game	*BS, EG, WP, or ZP*	*G*	*Request-Reply*, with *ExitGame* and AckNak as request and reply messages, where • If the *Status* of the *AckNak* message is *Success*, then the agent was successes removed from game. • If the *Status* of the *AckNak* message is *Failure*, the agent could not be exited from the game for some reason and the *Message* of the *AckNak* contains the specific reason or error message.

Message Encoding / Decoding

A message will be encoded recursively using the following rules:

1. The encoding of a *Message* object involves writing its Class Id, the length of its encoded properties, and its properties into a *ByteList*.

1.1. The encoding properties process is a pre-defined order of the class

1.2. Each property is encoded as follows:

1.2.1. A primitive numeric value (e.g. an integer) is written out in network byte order

1.2.1.1. Byte – 1 byte

1.2.1.2. Int16 – 2 bytes

1.2.1.3. Int32 – 4 bytes

1.2.1.4. Int64 – 8 bytes

1.2.1.5. Single Precision Real – 4 bytes

1.2.1.6. Double Precision Real – 4 bytes

1.2.2. A char is encoded by writing a two-byte Unique representation of the char value.

1.2.3. A string is encoded by writing out its length as an Int16 (in network byte order) and a sequence of bytes, where the bytes are a Unicode representation of the string.

1.2.4. A Boolean value is written out as a byte with a value of 0 (false) or 1 (true)

1.2.5. An array or list of primitive values is encoded by first writing out the count of elements in the array or list as an Int16 (in network byte order), followed by an encoding of each value following rules 1.2.1 – 1.2.4

1.2.6. A property whose value is object is first represented from a byte containing a “1” for True or a “0” for False. A true means that the object is present and its encoding follows. A false means the object is not present. The encoding of the objects follows Rule 1 recursively.

1.2.7. An array or list of objects is encoded by first writing out the count of elements in the array or list as an Int16 (in network byte order), followed by an encoding of each object following Rule 1