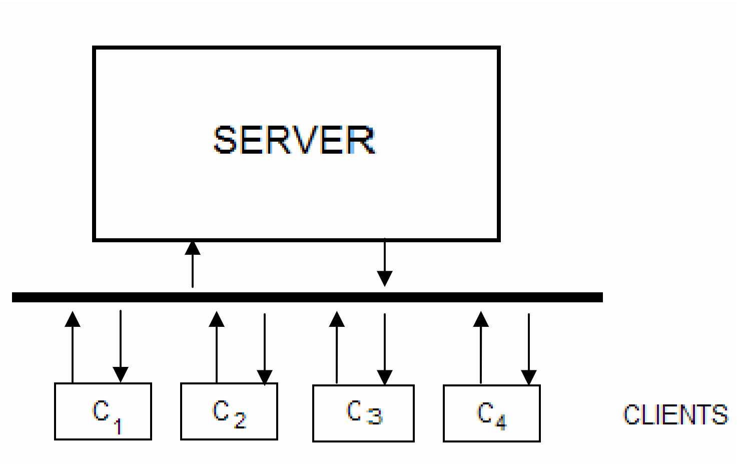
HomeWork 5

Done by:
Amritam Sarcar
Yong Wang

In this homework, an application based on Prism-MW that allows one to query one's scores in a networked environment is presented.

[1] Architecture Description

The architecture that has been implemented using Prism-MW has been documented here. The following few pages illustrate the description of the architecture.

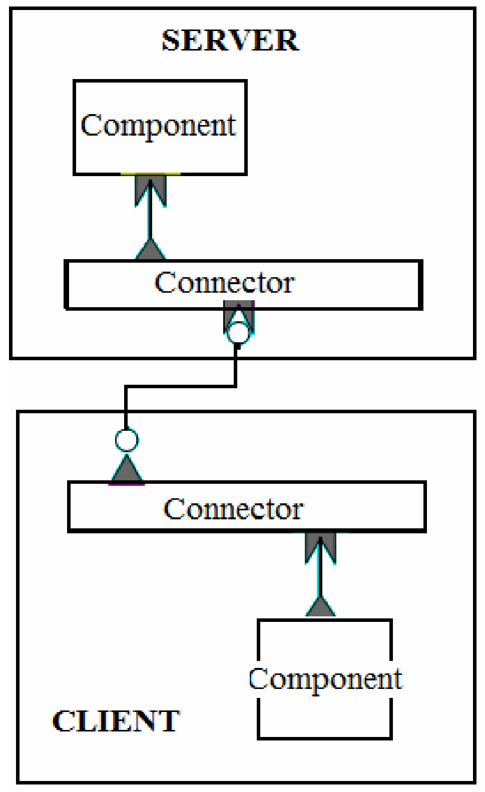


The diagram above shows two components Server and Client. C_1, C_2, C_3, …, C_n are several clients connected to the Server via the horizontal bar called the connector. Each of the clients interact with the server by sending a request and receiving the confirmation of the request through notification.

Prism-MW

The general architectural description of the client-server architecture in Prism-MW is depicted below.

It contains a distributed system which is implemented in Prism-MW that consists of a number of Architecture objects, each of which serves as a container for a single subsystem and delimits an address space. *Components* within and across the different *Architecture* objects interact by exchanging *Events.*



The Architecture Snapshot

Each component connects to a connector, and communicates with each other using Events. The client and server communicate through an extensible socket port. The responsible of *Query* is to get the request from work component, and send result to corresponding work component. Each of the *Worker Component* corresponds to each client; they are generated by *Net Monitor*. The monitor is a thread that checks regularly to determine whether client logs out or logs in. If a client logs in, it generates a *Worker Component* and assigns a unique key value to this component; the component would use this key value to identify its client. At the same time, the monitor sends an event with this key value to client to notify the client. So the client would use this key value as its identity to communicate later. For each event the client sends or server responds, the event includes this unique key. So we could allow multithread client work concurrently.

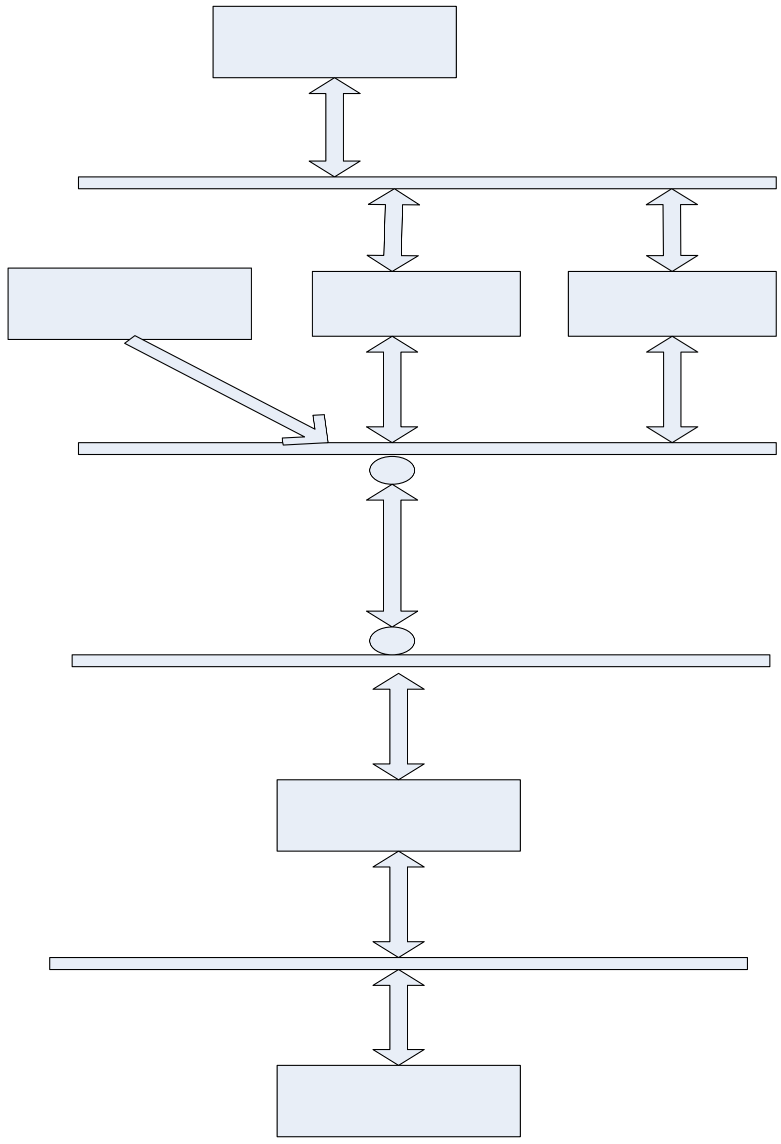


Figure.1 C/S message flow base on C2 architecture

Why we need a monitor and a unique key?

The reason is that the Prism-MW framework controls all the detail information about socket connection and disconnection. The only API it exposes to application program is the getConnections(). Prism has a vector that stores all connected socket, however, it doesn't return this socket connection to application program. When a component sends an event, Prism broadcasts this event to all connected clients. If we do not introduce unique key, for each event, server receives, server would not be able to distinguish this message has come from which client? So when client sends a request event, it needs to add its own key as identity.

How does application know that some clients are connected?

Monitor checks all connected sockets from socket Distribution() and compare those connections with previous one to determine which client logs in and which client logs out. For logout clients, corresponding components are deleted; for login clients, a *worker component* is dynamically generated that is responsible to receive and respond events to this login client and add this new component to the existing architecture. Client receives the event from extensible socket port; it only cares for those events which has the same key as its own. Because the extensible socket port in server side broadcast all events to all connected clients, we cannot do anything about it (concept of C2 connector) unless we are allowed to change the low level source code of Prism-MW. Of course, based on the same situation, if the client wants to know whether the server is still alive or not, it has its own monitor thread to check its own extensible socket port.

Flow of the System

The flow of system:

1. Server is up, monitor thread checks connection regularly.
2. A client connected, monitor knows that by checking connections vector. It generates a component to communicate this client later. The monitor also assigns a unique key to this component and an event including this key to client.
3. Client receives a key event and stores this key. Later, each time when client sends a request to server, it would carry this key as its identity. This key is stored in checker component.
4. GUI sends a request event to checker. Checker checks whether the pin number is correct or not. If the input is invalid, the checker sends a wrong pin event as response to gui. Otherwise, the checker adds key to event and forwards the query event request to server side. The work component with the same key (serverchecker in program) gets this event, and forwards it to query component to search the score.
5. The query component gets query event, it gets pin number from event, and search the score, send result event with score to reply request. Server Checker gets the result event by comparing its key with result event key. And write to client through its own connection.
6. Clients get the result event. Checker forward to GUI. Finally, GUI component gets result event and display it on its screen.

For each component they handle all those events as below:
GUI Component:
Request event: query event (input string)
Response event: wrong pin(result string); result event(score result); connect event(status of net notification)

Checker component:

Request event: query event (input string) from gui; connect event (request to connection from client)

Response event: key event (from server); result event (score result from server);

Server Checker:

Request event: query event (from client)

Response event: result event(score result) from query component;

Query component:

Request event: query event (from client)

Test case:

1. When server is up, client start, input a invalid pin number such as abc, get response Pin number invalid.
2. When server is up, client start, input a valid pin number such as 111, get response rose,70.
3. When server is up, client start, input a valid pin number such as 666, get response Pin number doesn't exist.
4. When server is up, client start, after a while, server down, all clients get the notification And query button convert to connect button.
5. When client start, if server is down, display connect button.
6. All test cases above support more than one client concurrently.