DATA COORDINATOR PATTERN

Russell Greenspan, CS 527 – Fall 2004

CONTEXT

A data-driven peer-to-peer or client-server application.

PROBLEM

Often times it is necessary for multiple components of an application to send and receive data, either from one peer to another on a peer-to-peer network, or from a client to a server in a client-server network. The traditional way to approach this situation is to have each component manage its own channel to and from the data source, and to have the component interact directly with this data channel. However, this potentially poses many problems:

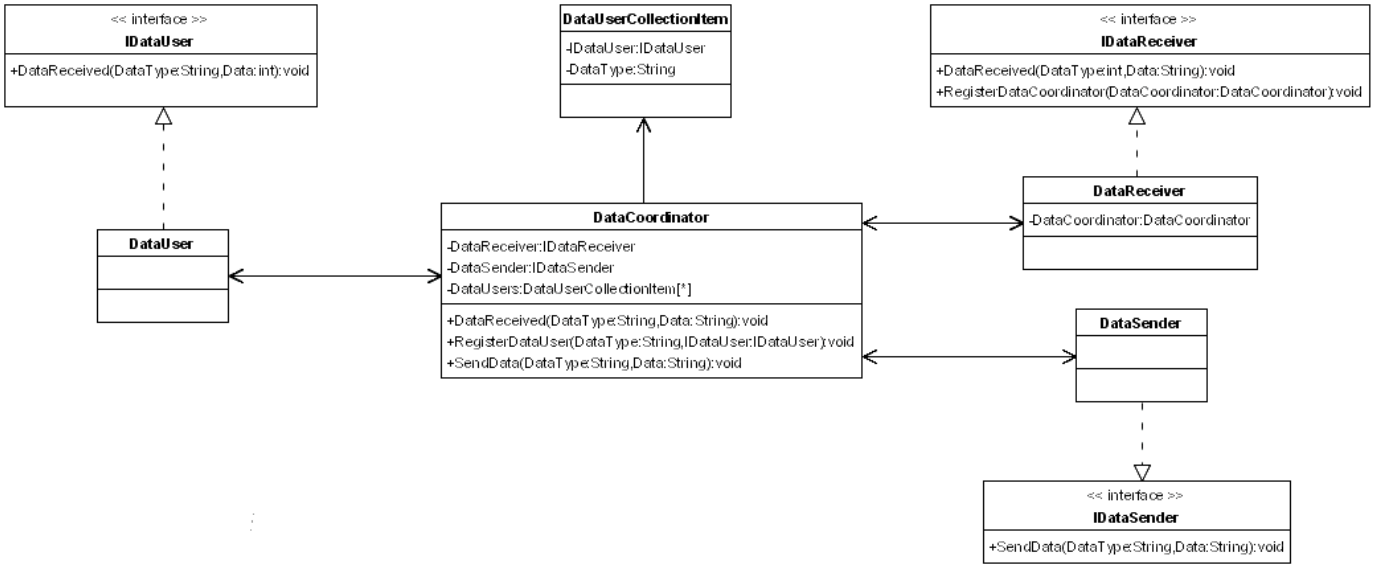
1. If the data source closes or is temporarily unavailable, all connections in each component must be reset.
2. There is no control on the number of open channels.
3. The component is responsible for cleaning up the data resources it uses.

SOLUTION

Abstract the data interaction from each component by having the components register themselves with a central DataCoordinator. The DataCoordinator is then responsible for maintaining a collection of components and the type of data the components are interested in receiving. The DataCoordinator is also responsible for sending data on the components' behalf. In this way, when two peer-to-peer clients inadvertently disconnect, the reconnection

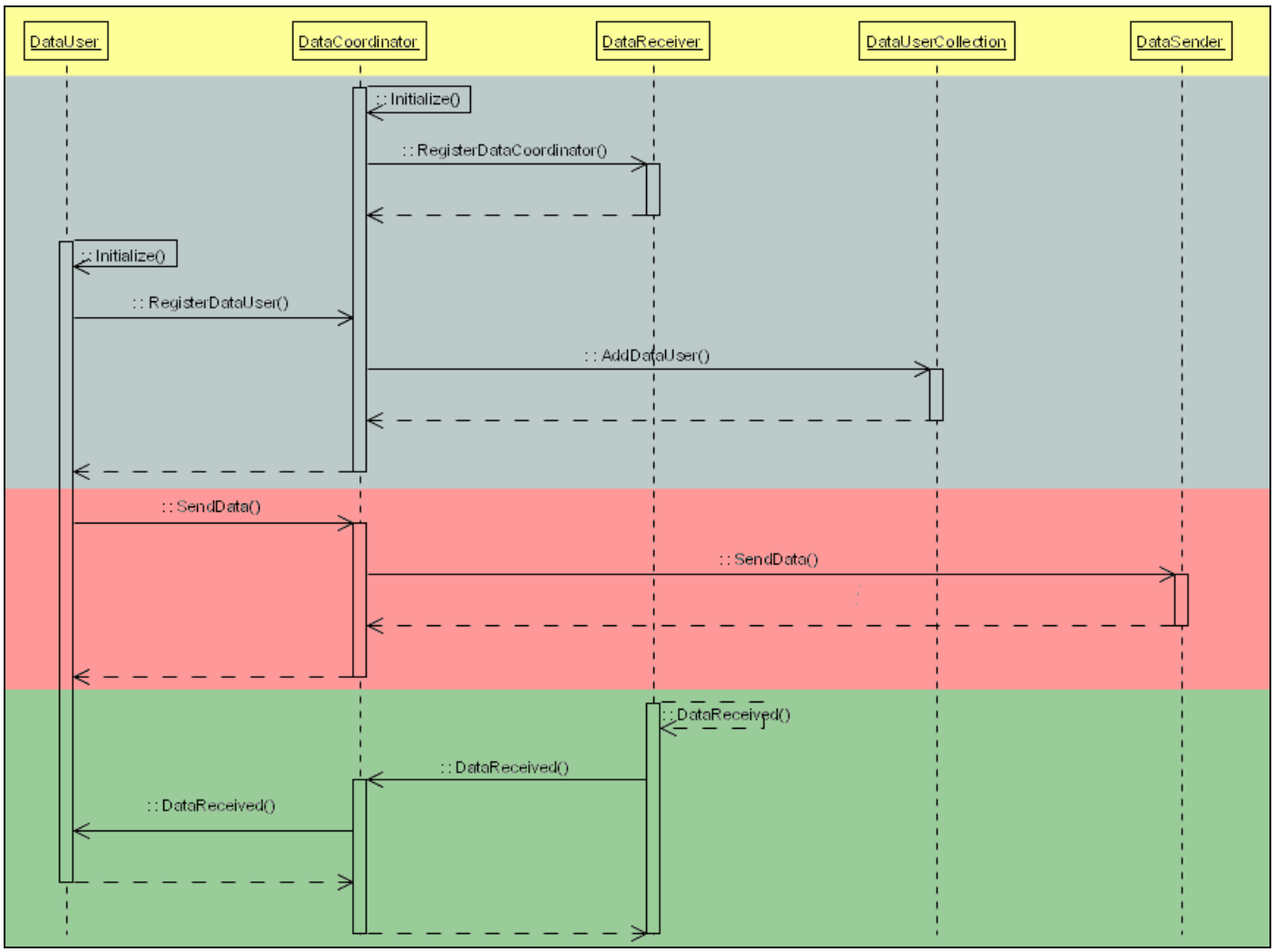
occurs in the DataManager, allowing the components to remain oblivious.

STRUCTURE



As pictured in the above class diagram, the DataCoordinator is the central component in the pattern. It sits between the DataUsers and the DataReceiver and DataSender, ensuring that it must be the mediator for all incoming and outgoing data.

DYNAMICS



The above class diagram shows the 3 scenarios:

1. Initialization
    a. The DataCoordinator initializes itself by registering with its DataReceiver. This allows the DataCoordinator to be alerted of all incoming data that the DataReceiver receives.
    b. Each DataUser component then calls `DataCoordinator::RegisterDataUser()` to tell the DataCoordinator that it is interested in receiving data of a certain type. The component calls `RegisterDataUser()` once for each type of data that it is interested in receiving.
    c. The DataCoordinator adds this component and the type of data it is interested in receiving to its internal collection.

2. Data Sending
    a. When a component wishes to send data to a remote component, it calls `DataCoordinator::SendData()`, which then forwards the call to the appropriate DataSender. The DataSender is responsible for actually implementing the functionality necessary to forward the data. For example, the DataSender might use HTTP or some custom protocol to send the data.

3. Data Receiving
    a. When data is received from a remote component, the DataReceiver looks up the DataCoordinator that has registered with it and forwards the data to the DataCoordinator. The DataCoordinator then cycles through its internal collection of registered DataUsers and calls `IDataUser::DataReceived()`, forwarding the data to any components that wanted to received data of this type.

IMPLEMENTATION

The DataManager uses a hash-table or collection to allow components to register themselves and the type of data they are interested in receiving with the DataCoordinator. The DataCoordinator then sends all outgoing data with this type-tag specifying which component requested that the data be sent. All replies include this tag, and so when data is received for a specific component, the DataCoordinator looks up the component in its hash-table and forwards the data to the proper component via the interface call IDataUser_DataReceived().

The data senders and receivers have also been abstracted from the DataCoordinator, so that the DataCoordinator can use any sender or receiver implementing the required interfaces.

```
Class DataCoordinator
{
   map<DataUserCollectionItem> pDataUsers;
   DataReceiver* pDataReceiver;
   DataSender* pDataSender;

   public void Initialize()
   {
      pDataReceiver->RegisterDataCoordinator(this);
   }

   public void DataReceived(String DataType, String Data)
   {
      foreach (pDataUser in pDataUsers)
      {
         if (pDataUser.DataType == DataType)
            pDataUser.IDataUser.DataReceived(DataType, Data);
      }
   }

   public void SendData(String DataType, String Data)
   {
      pDataSender->SendData(DataType, Data);
   }
}

Interface IDataUser
{
   public void DataReceived();
```

```
}

Class DataUser : IDataUser
{
   DataCoordinator* pDataCoordinator;


   public void IDataUser_DataReceived(String DataType, String Data)
   {
      //handle incoming data
   }

   private void doInitialization()
   {
      pDataCoordinator->RegisterDataUser("Type1", (IDataUser)this);
   }

   private void sendSomeData()
   {
      pDataCoordinator->SendData("Type 1", "Test Data");
   }
}
```

EXAMPLE

Consider a peer-to-peer application. The application maintains a dynamic set of shared tools that the application's users use to communicate. Imagine that one of these tools is a shared whiteboard, where each user can draw on the whiteboard and have the details of his mouse movements sent to the peer so that the same drawing can be rendered remotely. The application features a variety of different tools of this nature, where each tool performs features some type of collaborative functionality as the whiteboard.

At load time, each tool registers itself with its application's DataCoordinator. Then, when tools need to send data, they make the calls through the DataCoordinator. One application's DataSender then sends this data out, where the other peer application's DataReceiver is listening for it. The other peer's DataReceiver then forwards the data to its DataCoordinator, which forwards the data to that application's whiteboard, where the rendering is done. Thus the applications have shared data and the collaboration is complete.

CONSEQUENCES

The main benefit of this pattern is that components are decoupled from interacting with each other directly, thus eliminating a web of connections that is nearly impossible to maintain. Since all data flows through the DataCoordinator, if one DataCoordinator's outgoing channel is lost, the DataCoordinator can reset it without any of the components using the DataCoordinator needing to know about it.

Similarly, if necessary the DataCoordinator can throttle outgoing messages by using some type of timer to send components' data. This ensures that all components do not try to send large streams of data at the same instant, potentially flooding the remote peer application's DataReceiver.

The main liability of this pattern is that there is a single point of failure in the application: the DataCoordinator. This means that its implementation must be fail-proof, or else each

component using it will fail. Similarly, there could be thread-safe issues when multiple streams of data are received at the same time, so the DataCoordinator might need to use a mutex lock to ensure that `DataCoordinator::DataReceived()` is only called one thread at a time.

RELATED GOF PATTERNS
Observer, Mediator