**Example 7.4**
Using Java Threads for Asynchronous Requests

```
interface Callback {
  public void result(String s);
}

class PrintSquad implements Callback {
  public void Print(Team team, Date date) {
    AsyncReqPrintSquad a;
    a=new AsyncReqPrintSquad(team,date,this);
    a.start();
    // continue to do some work here.
  }
  public void result(String s) { // callback
    System.out.println(s);
  }
}

// thread that invokes remote method
class AsyncReqPrintSquad extends Thread {
  Team team;
  Date date;
  Callback call;
  AsyncReqPrintSquad(Team t, Date d, Callback c) {
    team=t; date=d; call=c;
  }
  public void run() {
    String s;
    s=team.asString(date); // call remote method
    call.result(s); // pass result to parent thread
  }
}
```

## Using Message Queues

The implementation of asynchronous requests with threads assumes that the server object is executing the requested operation while a thread in the client waits for the result. There may be situations, however, where requests need to be made that are more decoupled. Such situations often arise due to global distribution with much higher network latency or due to execution of objects in different time zones. As an example, consider a distributed application for the international stock market that is used to buy or sell shares on remote stock exchanges. The problem is that a stockbroker based in New York is working at a time when the Tokyo stock exchange is shut down and vice versa. When the New York broker uses a client object to make a request to sell shares to a server object in Tokyo, that request needs to be stored until the Tokyo stock exchange opens the next day. Also the result of the deal needs to be buffered until later in the day when business in New York reopens. Such decoupled asynchronous requests cannot be implemented using threads because the client object in New York and the server object in Tokyo are never operational at the same time and thus request information needs to be stored on some form of persistent storage.

Decoupled asynchronous requests are, rather, implemented using messages that may be temporarily stored. A request is represented as a pair of messages, a *request message* and a *reply message*. The main contents of the request message are the input parameters that are to be passed to the server object and an identifier for the requested operation. The main contents of the reply message are output parameters and the operation result, which are returned to the client.

Request and reply messages are exchanged between client and server objects by means of *message queues*. A message queue operates on the first-in first-out principle and guarantees that messages are removed in the same order in which they have been entered. Message queues often also implement persistent storage of messages in order to improve the reliability of message transport.

Message queues implement asynchronous requests that need to be stored persistently.

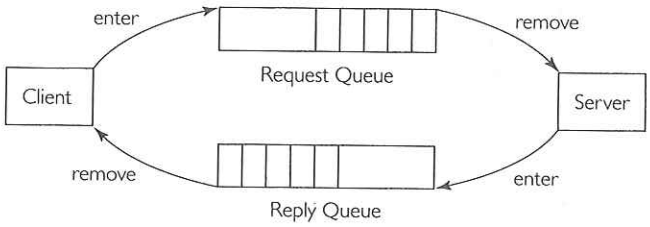



**Figure 7.4**
Using Message Queues for Asynchronous Requests

As shown in Figure 7.4, a pair of message queues is used to implement asynchronous requests between client and server objects. A *request queue* is used for transmission and buffering of requests and a *reply queue* transmits and buffers the replies. As queues operate on a first-in first-out basis, request messages are delivered in the order in which they arrive and the reply messages are delivered in the same order as the requests were sent to the server.

Message queues that can be used for the implementation of asynchronous requests are widely available. Implementations of message queues are generally referred to as *message-oriented middleware* (MOM). Their applicability is not restricted to object-oriented distributed systems. In fact, many MOMs that are being used today originated in mainframe operating systems. There are, however, several MOMs that integrate with object-oriented middleware. The CORBAservices include a messaging service that supports reliable queuing of messages. Implementations of that service provide an object-oriented interface to existing MOMs, such as IBM's MQSeries or DEC Message Queue. For COM, a message queuing system, currently named Falcon, is being specified while this book is being written.

## 7.2 Request Multiplicity

The requests that we have seen so far are *unicast* requests. This means that a client object requests an operation execution from exactly one server object. Other forms of requests can be obtained if we relax this assumption. We then have the situation of a *group request* where a client requests execution of the same operation from multiple server objects. A client may also wish to issue a *multiple request* to execute different operations from different objects.

A unicast request is made from one client object to one server object.

We discuss the principles underlying these two primitives in this section and show how they can be implemented.

## 7.2.1 Group Communication

A client that requests execution of the same operation from a number of servers makes a group request.

A client may wish to request execution of the same operation from a number of different objects. If these operations do not return results, there is no need for the client to know about the individual objects. The request can be made to a group of objects where the group members remain anonymous. We refer to these requests as *group requests*. They are used for notification purposes, for instance when multiple server objects need to be notified about an *event*, which occured in a client.

Group composition is managed by a channel and is unknown to the request producer.

Group composition is usually managed by a *channel*, through which group requests are posted. Objects interested in a certain type of event are referred to as *consumers*. They register with the channel and then start listening to the events and deregister if they want to stop being notified about events. The object wishing to make a group request in order to notify consumers about an event is referred to as a *producer*. It posts the request into the channel and the channel then forwards it to all registered consumers. Group requests are therefore *anonymous* forms of communication as the producer does not know its consumers.

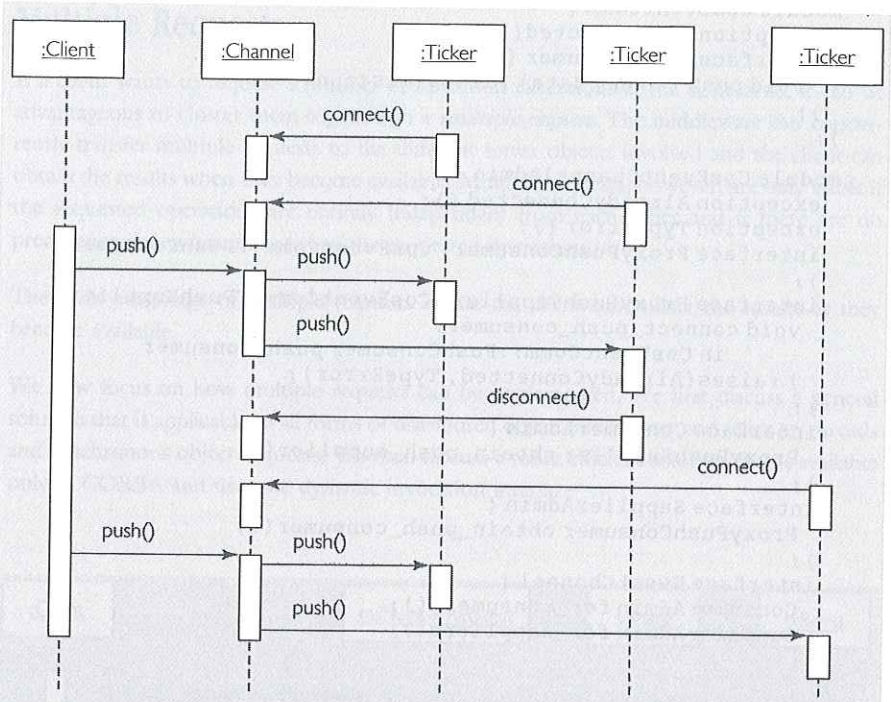
Group requests are anonymous.

### Implementing Group Requests

Group requests can be implemented using synchronous requests.

One of the CORBAservices is the *Event service*. It supports the implementation of group requests based on standard CORBA object requests. Such a service is currently not available for Java/RMI or COM. However, implementations of the CORBA Event service use synchronous, unicast object requests. These are also available in Java/RMI and COM and therefore an event service for Java/RMI or COM can easily be implemented following the design for the CORBA Event service. Thus, the discussion of the CORBA Event service should be seen as an example of how a group request can be implemented on top of synchronous unicast object requests. CORBA events are broadcast using the *event channels* mechanism to identify the subgroup of event consumer objects that are to be notified about a certain event. Event consumers therefore connect to those event channels that correspond to events in which they are interested. For an event producer, an event channel represents the community of those consumer objects that are interested in the event. The producer, however, only knows that the community exists; it does not know how large it is or which objects belong to it. The event communication between producers and consumers is entirely anonymous.

The operations that event channels provide are separated into a number of interfaces. The rationale for this split is that the designers of the CORBA Event service wanted to support the concatenation of multiple event channels. Such a concatenation might be useful if different implementations have different qualities of service. One event channel might, for example, be capable of storing event data persistently. Thus they had to devise a strategy so that event channels can also act as event consumers and producers. This is achieved by splitting the event channel interfaces into `ProxyPushConsumer` and `ProxyPushSupplier` interfaces, which represent the event consumer to a supplier and




## Example 7.5
Group Communication Example

Traders who deal in shares would launch a stock exchange ticker in order to see details of deals that have come through. The successful completion of a share deal would be an event about which they would want to be notified. Hence, the trading application of the stock exchange, which determines the price for a deal, broadcasts the details of the deal to every active ticker. The trading application uses group requests in order to notify all the tickers in one go and it is not and should not need to be aware of the tickers that are currently active.

the supplier to a consumer. An excerpt of the IDL interfaces to the event service is shown in Figure 7.5. The `EventChannel` interface has two operations that return administration objects, which have operations to return the proxy objects. Objects that wish to receive group requests would then implement the `PushConsumer` interface and connect with a `ProxyPushSupplier` using the `connect_push_consumer` operation.

The model of event communication discussed above is the *push* model, in which the producer takes the initiative and pushes events through the channel to the consumer. The event may contain data but the event channel does not have any knowledge about the type of this data, as it is of type `any`. It is worthwhile to note that the CORBA Event service supports several other models of event-based communication. It includes a *pull* model in which the consumer requests delivery of an event from the producer which is then pulled through the event channel. It also supports typed communication where the push and pull operations have a particular type attached to them so that it can be proven when the object is compiled that the right type of data is communicated through the channel.