Distributed Systems: Concepts and Design

Chapter 6 Exercise Solutions

6.1 Construct an argument as to why indirect communication may be appropriate in volatile environments. To what extent can this be traced to time uncoupling, space uncoupling or indeed a combination of both?

6.1 Ans.

Volatile environments are environments where change is anticipated both in terms of the availability of computers and the underlying network. Mobile computing is one example of a highly volatile environment where users may connect and disconnect from the global network and may also experience periods of disconnection or weak connection because of location. Indirect communication is an appropriate strategy for such volatile environments for two key reasons:

Firstly, with the potential for failure, it is useful to have a communication mechanism that does not communicate directly with a given recipient, rather indirectly through an intermediary, providing a level of freedom for the system to replace, update, replicate or migrate the intended receiver (space uncoupling).

Secondly, with the potential for disconnection (of both sender and receiver), it is useful to have a communication mechanism that does not rely on both parties being available at the same time to communicate (time uncoupling).

6.2 Section 6.1 states that message passing is both time- and space-coupled – that is, messages are both directed towards a particular entity and require the receiver to be present at the time of the message send. Consider the case, though, where messages are directed towards a name rather than an address and this name is resolved using DNS. Does such a system exhibit the same level of indirection?

6.2 Ans.

As discussed in Section 13.2.3, in DNS, a name may map on to more than one IP address, for example to share load across a number of computers. Given this, a name server provides an additional level of indirection in that a sender may not be bound to a given receiver but rather refer to a logical name which is then bound to one of a number of possible receivers. This provides a similar effect to space uncoupling although the implementation details are different when compared to communicating through a more explicit intermediary such as a group, publish-subscribe system, message queue or shared memory abstraction.

6.3 Section 6.1 refers to systems that are space-coupled but time- uncoupled – that is, messages are directed towards a given receiver (or receivers), but that receiver can have a lifetime independent from the sender's. Can you construct a communication paradigm with these properties? For example, does email fall into this category?

6.3 Ans.

This is a subtle point and we invite the reader to dwell on this themselves. Is there a communication paradigm whereby a message is directed towards a given named receiver with known identity and hence bound in space, but where the sender and recipient have different lifetimes? This partially depends on the interpretation of different lifetimes, that is whether this implies the receiver may not exist yet or whether the receiver does exist

but is not available at a given time to receive a message. If we take the first interpretation, we can think of a non-computer example, that is a letter addressed to a yet unborn great-grandchild. We can also think about time capsules addressed to a given recipient. If we take the second interpretation, then email does fit this category, that is a mail message is directed towards a known receiver who may or may not be around at a given time but who will receive the message when next connected (see also Exercise 6.4 below). Note that this assumes that we are focusing on the delivery of the message to the intended person rather than their mailbox.

6.4 As a second example, consider the communication paradigm referred to as queued RPC, as introduced in Rover [Joseph *et al*. 1997]. Rover is a toolkit to support distributed systems programming in mobile environments where participants in communication may become disconnected for periods of time. The system offers the RPC paradigm and hence calls are directed towards a given server (clearly space-coupled). The calls, though, are routed through an intermediary, a queue at the *sending* side, and are maintained in the queue until the receiver is available. To what extent is this time-uncoupled? Hint: consider the almost philosophical question of whether a recipient that is temporarily unavailable exists at that point in time.

6.4 Ans.

This relates precisely to the point made above of how we interpret independent lifetimes. If we interpret this as the receiver (in this case the server) not yet existing then this is not time-uncoupled. If however we interpret it as not being available at a given time, that is the server exists but is disconnected, then this is time-uncoupled. Again we see this is rather a subtle issue and one that is not fully understood in the literature.

6.5 If a communication paradigm is asynchronous, is it also time-uncoupled? Explain your answer with examples as appropriate.

6.5 Ans.

Asynchronous communication is where a sender sends a message and then continues and does not have to block until the message is delivered to the receiver, that is the sender and receiver do not have to meet in time to exchange a message. Time uncoupling goes further by stating that senders and receives can have independent existences (see also the discussion in Exercises 6.3 and 6.4).

Many message passing systems are asynchronous but directed towards a recipient that is assumed to exist as a given point in time (for example, the non-blocking variants of MPI as discussed in Section 4.6). In other words, an asynchronous system could decline the send if the recipient doesn't exist at that point in time, and hence is not time-uncoupled.

6.6 In the context of a group communication service, provide example message exchanges that illustrate the difference between causal and total ordering.

6.6 Ans.

As a reminder, causal ordering takes into account the causal relationships between messages, that is if a message happens before another message, this causal relationship must be preserved in the delivery of the associated message to all processes in a group. In total ordering, every process must see all messages in the same order. Clearly, total ordering is a stronger property than causal ordering in that causal ordering only imposes a delivery constraint on messages where a happens before relationship exists. Looking at this the other way round, causal ordering says nothing about the delivery order of messages that are deemed independent of one another.

Figure 15.11 provides one example of message exchanges that exhibit causal and total ordering. The bulletin board example that follows in Section 15.4.3 provides an excellent illustration of the two styles of ordering. If total ordering is employed, every user of the bulletin board would see every posting in exactly the same order. If causal ordering is employed, and we assume there is a causal ordering between messages related to the same thread, but there is no such relationship across threads, then users would see the ordering preserved within a thread, that is the conversation would be presented in the same order. Other messages would be interleaved though in different ways across all users.

As a second example, consider a multiplayer online game where there are two styles of message, one that updates the state of the game and others where players communicate with one another. To maintain the consistency of the game state, it is important to maintain a causal ordering across updates. It may also be important to maintain ordering of the other messages sent between players or players may see a reply to a message before the initial message. Clearly these causal relationships need to be preserved. A total ordering

may actually be required here as well if the game developers feel it is important to relate messages precisely to the current state of the game.

6.7 Consider the *FireAlarm* example as written using JGroups (Section 6.2.3). Suppose this was generalized to support a variety of alarm types, such as fire, flood, intrusion and so on. What are the requirements of this application in terms of reliability and ordering?

6.7 Ans.

As these are all critical messages, the key requirement is for reliable multicast as defined in Section 6.2.2 in terms of integrity, validity and agreement. Ordering is less of an issue in this example as there is no strong need to preserve ordering across alarm types. Similarly, this application is quite simple and there is no concept of a sequence of messages relating to a given alarm, for example alarm raised, alarm being dealt with, alarm dealt with. If this added sophistication was introduced into the example, then causal ordering would be required.

6.8 Suggest a design for a notification mailbox service that is intended to store notifications on behalf of multiple subscribers, allowing subscribers to specify when they require notifications to be delivered. Explain how subscribers that are not always active can make use of the service you describe. How will the service deal with subscribers that crash while they have delivery turned on?

6.8 Ans.

A notification mailbox service can readily be implemented using a system such as the Jini as mentioned in Section 6.3.1, with more details also contained in the companion web site for the book [www.cdk5.net/rmi]. If implemented using Jini, the mailbox service would provide an interface allowing a client to register interest in another object. The client needs to know the *RemoteEventListener* provided by the Mailbox service so that notifications may be passed from event generators to the *RemoteEventListener* and then on to the client. The client also needs a means of interacting with the Mailbox service so as to turn delivery on and off. Therefore define register as follows:

Registration register() ...

The result is a reference to a remote object whose methods enable the client to get a reference to a *RemoteEventListener* and to turn delivery on and off.

To use the mailbox service, the client registers with it and receives a Registration object, which it saves in a file. It registers the *RemoteEventListener* provided by the Mailbox service with all of the *EventGenerators* whose events it wants to have notification of. If the client crashes, it can restore the *Registration* object when it restarts. Whenever it wants to receive events it turns delivery on and when it does not want them it turns delivery off.

The design should make it possible to specify a lease for each subscriber.

6.9 In publish-subscribe systems, explain how channel-based approaches can trivially be implemented using a group communication service? Why is this a less optimal strategy for implementing a content-based approach?

6.9 Ans.

In a channel-based approach, events are published to a named channel and subscribers subscribe to a given channel and receive all messages sent on that channel. This can be implemented directly using group communication, that is when a channel is created, you create an associated group; subscriptions are implemented through joining the group and then subsequent events from publishers are sent to the group and hence delivered to all subscribers who are then members of the group.

This trivial mapping is not possible with content-based approaches where communication is more associative and based on content (more specifically, based on arbitrary queries expressed over the content). This cannot easily be implemented using group communication. One option would be to send all events over one group and then do the required filtering in each subscriber, but this would be really inefficient. In practice, as we have seen in Section 6.3.2, efficient implementation of content-based approaches requires the creation of an suitable overlay implementing a content-based routing algorithm, that is routing is influenced by content.

6.10 Using the filtering-based routing algorithm in Figure 6.11 as a starting point, develop an alternative algorithm that illustrates how the use of advertisements can result in significant optimization in terms of message traffic generated.

6.10 Ans.

The filtering-based routing algorithm described in Figure 6.11 significantly reduces the amount of traffic generated in comparison to flooding approaches by maintaining state information on each node in the overlay. In particular, each node must maintain a list of all directly connected subscriptions and also routing information that determines if there is a valid (matching) subscription down a connected path. This means that publish events are only forwarded down paths that will lead to a valid subscription, thus reducing the amount of traffic generated specifically by publish events. The downside is that subscription events need to be propagated fully through the overlay for this to be optimal and this flooding leads to a significant amount of traffic, an overhead which can be reduced by advertisements.

An advertisement-based approach applies the same principle to subscriptions as the above scheme applies to publish events. Advertisements are issued by publishers to indicate an intent to publish particular kinds of events in the future. Each node must now hold an additional advertisement-based routing table. In the same way that the initial (subscription-based) routing table optimizes the forwarding of publish events (by ensuring there is a valid path to a matching subscription), the second advertisement-based routing table is used to optimize the routing of subscriptions. In particular, subscriptions only need to be forwarded only into parts of the network where publishers have issued overlapping advertisements. This approach is described in detail in [Muhl *et al*. 2006].

6.11 Construct a step-by-step guide explaining the operation of the alternative rendezvous-based routing algorithm shown in Figure 6.12.

6.11 Ans.

This algorithm relies on an implementation of two functions, *EN(e)* and *SN(s)*. As described in Section 6.3.2, *SN(s)* takes a given subscription, *s*, and returns one or more rendezvous nodes that take responsibility for that subscription. Each such rendezvous node maintains a subscription list, and forwards all matching events to the set of subscribing nodes. Second, when an event *e* is published, the function *EN(e)* also returns one or more rendezvous nodes, this time responsible for matching *e* against subscriptions in the system. In the description below, we do not look at the implementation of the two functions; we only assume that a correct implementation is provided whereby the intersection of *EN(e)* and *SN(e)* is non-empty. Appropriate implementations of the two functions are considered in Exercise 6.12.

The algorithm is quite straightforward given appropriate implementations of the two functions.

On the occurrence of a subscribe event, the associated rendezvous nodes are calculated, using *SN(s)*. If the local node is one of the rendezvous nodes, the subscription is added to the local subscription list for future reference, otherwise it is propagated to other rendezvous nodes.

On the occurrence of a publish event, the associated rendezvous nodes are determined by calling *EN(e)* on the associated event, *e*. If the local node is in this set, the subscription list is consulted to determine subscriptions that match. The event is then sent directly to each of these subscribers through notify messages ensuring all subscribers will see this event. If the local node is not a rendezvous node, the publish event is propagated to the nodes contained in *EN(e)*.

Note: there is an error in the first impression of the book in that the last line of the receive implementation should be:

```
else
    send publish(e) to rvlist;
```

rather than

```
send publish(e) to rvlist - i;
```

This will be fixed in subsequent impressions.

6.12 Building on your answer to Exercise 6.11, discuss two possible implementations of *EN(e)* and *SN(s)*. Why must the intersection of *EN(e)* and *SN(s)* be non-null for a given *e* that matches *s* (the intersection rule)? Does this apply in your possible implementations?

6.12 Ans.

As mentioned in Chapter 6, a common mechanism for implementing the two functions is to map on to an underlying Distributed Hash Table (DHT). For example, consider Scribe, a publish-subscribe system implemented over Pastry. Scribe is a topic-based publish-subscribe system. In Scribe, *EN(e)* and *SN(s)* are identical operations and are implemented by the underlying hash function applied to the topic name. This ensures that rendezvous-nodes are chosen at random from the underlying nodes in the associated overlay.

Meghdood [Cao and Singh 2005] offers a second example of how to implement rendezvous-based routing. Meghdood is a content-based publish-subscribe system and, as such, the designers were interested in determining rendezvous nodes not just from the topic or one field but from the entire content. This is a much harder problem. The solution is to map on to a Content Addressable Network (or CAN). A CAN is an overlay network representing an n-dimensional space where each node is responsible for a partition of this space. Meghdood provides a particular subscription language which effectively defines this multi-dimensional space (the space of all possible subscriptions) with the underlying CAN then providing the associated partitioning of this space. The node responsible for each partition is exactly equivalent to the rendezvous node. Please refer to the Meghdood paper for details.

It should be fairly obvious why the intersection rule is required. In all rendezvous-based routing algorithms, subscriptions are sent to one or more rendezvous nodes with the subscriptions stored at these nodes. For a publish event to be routed to all subscriptions, for any event, the calculated set of nodes offered by *EN(e)* must overlap with the corresponding calculation of *SN(s)* to find a node that will then match to the subscription and forward it accordingly.

In the first implementation, the intersection rule is trivially met as *EN=SN*. For the second, the argument the rule is also met. In Megdhood, an event maps to a CAN region which cover all possible subscriptions that can map to that event (known as the event region).

6.13 Explain how the loose coupling inherent in message queues can aid with Enterprise Application Integration. As in Exercise 6.1, consider to what extent this can be traced to time uncoupling, space uncoupling or a combination of both.

6.13 Ans.

Enterprise Application Integration is concerned with providing an integration framework for linking separately deployed services within an organisation, for example student record systems, financial systems and virtual learning environments within a university. These are independently developed and deployed systems that are not necessarily designed to talk to each other. In this context, message queues provide a level of indirection where services do not need to communicate directly with each other but can communicate through message queues. This is important as such services do not know about each other. Different topologies of interconnection can also be implemented using the message queue framework with this framework then providing the required coordination logic across the services.

The key property here is space uncoupling, that is not needing to know the identity of the recipient. Time uncoupling is not an important requirement as we can assume that such services are generally available all the time.

6.14 Consider the version of the *FireAlarm* program written in JMS (Section 6.4.3). How would you extend the consumer to receive alarms only from a given location?

6.14 Ans.

This particular example is implemented using a *TopicConnection* in JMS, that is using the publish-subscribe style of connection. The desired effect can be achieved by associating a user-defined property with each message, capturing the originating location of this message with the consumer specifying a message selector based on topic and this property.

6.15 Explain in which respects DSM is suitable or unsuitable for client-server systems.

6.15 Ans.

DSM is unsuitable for client-server systems in that it is not conducive to heterogeneous working. Furthermore, for security we would need a shared region per client, which would be expensive.

DSM may be suitable for client-server systems in some application domains, e.g. where a set of clients share server responses.

6.16 Discuss whether message passing or DSM is preferable for fault-tolerant applications.

6.16 Ans.

Consider two processes executing at failure-independent computers. In a message passing system, if one process has a bug that leads it to send spurious messages, the other may protect itself to a certain extent by validating the messages it receives. If a process fails part-way through a multi-message operation, then transactional techniques can be used to ensure that data are left in a consistent state.

Now consider that the processes share memory, whether it is physically shared memory or page-based DSM. Then one of them may adversely affect the other if it fails, because now one process may update a shared variable without the knowledge of the other. For example, it could incorrectly update shared variables due to a bug. It could fail after starting but not completing an update to several variables.

6.17 Assuming a DSM system is implemented in middleware without any hardware support and in a platform-neutral manner, how would you deal with the problem of differing data representations on heterogeneous computers? Does your solution extend to pointers?

6.17 Ans.

The middleware calls can include marshalling and unmarshalling procedures. In a page-based implementation, pages would have to be marshalled and unmarshalled by the kernels that send and receive them. This implies maintaining a description of the layout and types of the data, in the DSM segment, which can be converted to and from the local representation.

A machine that takes a page fault needs to describe which page it needs in a way that is independent of the machine architecture. Different page sizes will create problems here, as will data items that straddle page boundaries, or items that straddle page boundaries when unmarshalled.

A solution would be to use a 'virtual page' as the unit of transfer, whose size is the maximum of the page sizes of all the architectures supported. Data items would be laid out so that the same set of items occurs in each virtual page for all architectures. Pointers can also be marshalled, as long as the kernels know the layout of data, and can express pointers as pointing to an object with a description of the form "Offset o in data item i", where o and i are expressed symbolically, rather than physically.

This activity implies huge overheads.

6.18 How would you implement the equivalent of a remote procedure call using a tuple space? What are the advantages and disadvantages of implementing a remote procedure call–style interaction in this way?

6.18 Ans.

To implement a remote procedure call, it is necessary for the 'client' to *write* a tuple to the tuple space that is effectively directed towards a given server and operation name within this server. This tuple would include the following fields: an indicator that this is a request, a nonce uniquely representing this request, the server name, the operation name and then the list of parameters. The client would then immediately perform a *take* operation, thus blocking the client until the remote procedure call completes (it is also important this is a *take* operation to remove the tuple from the tuple space). The server will sit in a loop performing *take* operations on tuples that match the server name, picking up the other values as required to carry out the operation. The server will then *write* a tuple with the following fields: an indication that this is a reply message, the nonce as picked up from the request, and the list of results. The corresponding *take* operation at the client is blocking awaiting a tuple that is a reply with the same nonce as the request. This operation can then collect the intended results and return them to the application.

One advantage of this implementation is that a server need not be present at the time the request is made (whether through failure or temporary disconnection) and then the server will match associated tuples when it reconnects with the tuple space. In addition, if we allow multiple servers to co-exist matching on the same server name, this would automatically provide a form of load balancing and would also introduce a level of fault-tolerance enabling the system to cope with failure of servers and continue to provide a level of service.

The disadvantage of implementing a remote procedure call in this way is the extra indirection involved in the implementation, hence inevitably adding to the latency of the interaction.

6.19 How would you implement a semaphore using a tuple space?

6.19 Ans.

Semaphores can trivially be implemented in tuple spaces using the *write* and *take* operations. As a reminder, semaphores have two operations *P* (or *wait*), and *V* (or *signal*). The *P* operation is implemented by doing *write("Sem")* with the V operation implemented by a matching *take("Sem")*.

6.20 Implement a replicated tuple space using the algorithm of Xu and Liskov [1989]. Explain how this algorithm uses the semantics of tuple space operations to optimize the replication strategy.

6.20 Ans.

This is a programming exercise left to the reader with further details of the algorithm found in the Xu and Liskov paper. Figure 6.21 can also be used to inform this implementation.

In terms of optimization, this algorithm is based around a deep understanding of the semantics of tuples spaces and the associated operations, recognizing that tuples are immutable, that *reads* do not interfere with the tuple space and it is only necessary to find one tuple (which may be local or may be found nearby), that *take* operations only need to lock tuples spaces until the appropriate tuple is selected for deletion with this supplemented by the additional rules listed at the top of page 270, again extracted from the required semantics associated with the concurrent execution of different operations.