Universitat Oberta de Catalunya

PAC 3- DISTRIBUTED SYSTEMS

Index

1. Question 1

Explain what temporal and spatial coupling mean. Give examples for the four combinations of space and temporal coupling/uncoupling different from the ones that you can find in the book.

Space and time coupling or uncoupling are properties of distributed systems and define how dependent is the system in terms of space and time.

On one hand, in indirect communications where it is used an intermediary participant uncoupled from receivers and sender, it achieves:

- Space uncoupling defines the situation in which the sender and the receivers do not need to know the identity of them. It allows to do some changes in participants without difficulty.

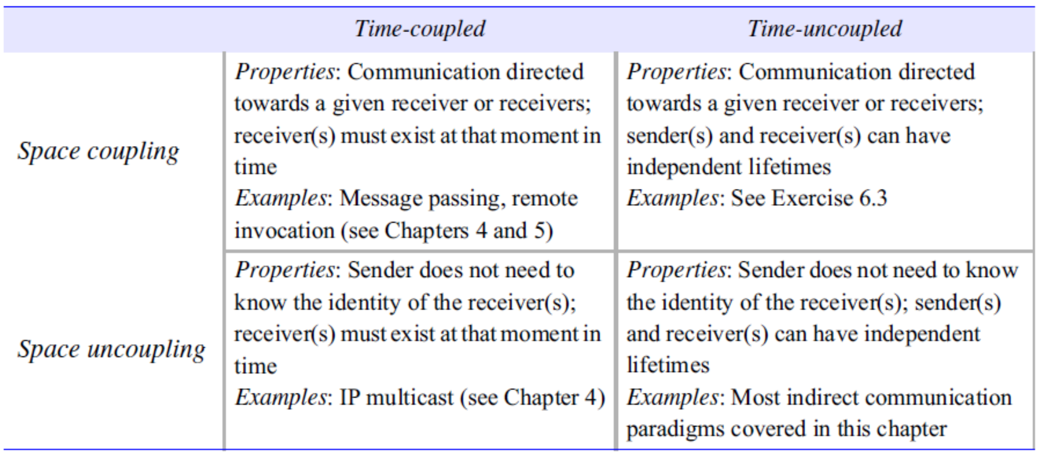
- Time uncoupling defines the independence of lifetime of both participants, receivers and sender. They do not need to exist at the same time to communicate.

The flexibility of uncoupling in indirect communication is counteracted from the difficulty of managing these systems.

On the other hand, explaining the opposite concepts, the space coupling refers to a situation where it is needed to know who the sender and the receivers are. Moreover, time coupling needs to exist both sender and receivers at the same time.

In the next table extracted from the book (1), it is summarized the four combinations of these properties:

Table 1. Properties and examples from (1).



	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 6.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Examples:

In my opinion, I relate the next examples with the different combinations of space and time coupling or uncoupling:

- **Time and space coupled:** in applications used to control and manage different machines in factories. The identity of both must be identified and it must be a real time application.

- **Time coupled and space uncoupled:** an application of radio online. The receivers are not known but sender and receiver must exist at the same time.

- **Time uncoupled and space coupled:** Dropbox. The identification of receiver/sender is known but receivers do not need to exist at the same time.

- **Time and space uncoupled:** Redis publish/subscribe messaging system. The receiver and the sender do not need to know the identity of the other and it is not necessary to be online at the same time.

2. Question 2

Briefly describe the components of the BitTorrent architecture. Explain and compare the following concepts used in BitTorrent: BitTorrent DHT and BitTorrent PEX protocol.

BitTorrent is a peer-to-peer file-sharing application with the aim to download large files. It is not for real-time streaming, but rather for download to be played later.

The main different components in BitTorrent, are:

- Chunks: splited fixed-sized pieces of a file. These are available at different sites across peer-to-peer network to decentralize the load charge and avoid bottlenecks in popular files.

- Tracker: is specified as a URL. It is the centralized server responsible for managing downloads of a particular file. This concept breaks the pure peer-to-peer principles, but it allows to centralize this information.

- Seeder: it is a peer with has a complete version of a file (with all chunks).

- Leecher: the peers that want to download a particular file (only have a partial of its chunks). When leecher completes the downloading, the next time will be considered as a seeder.

- Torrent (or swarm): include trackers, seeders and leechers.

- .torrent file: a file that contains metadata associated with that file with the next information: the name and length of the file, the location of a tracker and a checksum associated with each chunk (generated with SHA-1 hashing algorithm) to verify the download.

The process to be done when a peer wants to download a file is the next one: it contacts the tracker and get back a set of peers that can support the download. Once done it, the tracker will not be involved more in this downloading (this part of the protocol is decentralized because it implies different peers). After that, the chunks are requested in any order and the file will be downloaded.

BitTorrent and other many peer-to-peer protocols trust that all peers will contributes (this is known as the tit-for-tat mechanism). It is a mechanism to reward and gives preference to peers who has previously or who is currently uploading to that site. Besides, tit-for-tat support patterns of communication for downloading and uploading concurrently to optimize the bandwith. It is supported to unchoke peers.

An object’s GUID (identifier) is computed from all or part of state of the object using a hash function that makes the value of that considered as unique (with a high probability). These identifiers are used to determine the location of objects and retrieve them, for this reason, overlay routing systems are, sometimes, also called **distributed hash tables (DHT).**

It is considered that DHT takes responsibility for locating nodes and objects. Moreover, it ensures that any node can access any object by routing each request through a sequence of nodes, checking them to locate the destination object. BitTorrent (like other peer-to-peer systems) usually store multiple replicas of files to ensure availability. Nevertheless, DHT maintains knowledge of the location of all the available replicas and send requests to the nearest node (one without failure) that has a copy of the object. To identify nodes and objects, GUIDs are used.

The main operations of DHT are routing of requests to objects (as we have mentioned before), insertion of objects (to available one object to network), deletion of objects (having to indicate as unavailable) and node addition and removal (allowing joining and leave the service).

Referring to **Peer exchange (PEX),** it is a communication protocol to increase efficiency and speed of sharing files between peers. As I have described before, BitTorrent needs a tracker to locate the peers to download a file and it maintains the swarm. To reduce this reliance, PEX allows each peer to directly update others in the swarm.

DHTs reduce the load on the central tracker computer, however, PEX allows peers to exchange information about the swarm without polling a tracker or a DHT. For this reason, PEX decentralize the system distributing the knowledge of information of the swarm and it is more efficient.

PEX cannot be used to make the initial contact with a swarm, instead of this, each peer must connect to a tracker by .torrent file or use a router computer (bootstrap node) to find DHT with information of swarm’s list of peers.

3. Question 3

Explain the differences between publish/subscribe and message queue systems. Give examples of both to illustrate the differences.

Publish subscribe systems is the most used technique of indirect communication. It is based on a one-to-many system with two figures: publishers who publish structured events to an event service, and subscribers who received event notifications of their interest subscribed events. The goal of this systems is to ensure that events are delivered to all subscribers matched with the event filters.

Message queues is also a system of indirect communication but based on point-to-point service instead of publish-subscribe system. In this system exist the next figures: producer processes (it can be more than one) that can send messages to a specific queue, and other processes (consumer) that can receive messages from this queue (it can be more than one receiver who remove messages from a queue).

The main characteristics of **publish-subscribe systems** are:

- Heterogeneity: different components can work together when event notifications are used as the way to communicate.
- Asynchronicity: notifications are sent asynchronously to all subscribers with interest in event-generated publishers, so they are decoupled.
- Publish-subscribe allows to provide different delivery guarantees for notifications.

In case of **message queues**, the main properties are:

- Persistent: the queue will store the messages indefinitely (until they are consumed) and it will also commit the messages to disk to enable reliable delivery.
- Validity: any message sent is eventually received and the message received is identical as the one sent.
- Integrity: there are not sent duplicated messages.
- Guarantee that messages will be delivered but it cannot control the timing of delivery.
- Most commercially systems allow to contain a message within a transaction.
- Some systems support message transformation between formats to deal with heterogeneity.
- Some implementations provide support for security.

Both of systems are space and time uncoupled.

Subscriptions of **publish subscribe system** are filtered based on:
- Channel-based: publishers and subscribers refer to a named-channel (as a physical channel).
- Topic-based (or subject-based): subscribers subscribe to a topic (defined into a field of notification).
- Content-based: generalization of topic-based allowing the functionality to filter by queries.
- Type-based: linked with object concept. Subscriptions are defined by types of events and matched by a range of filters. It can be integrated with program languages.
- In some commercial systems are based on subscriptions to objects of interest. It is the same as type-based but with the particularity of that they focus on changes of state of objects (for example, a click of a button) rather than types of objects.

In **message queues**, the policy is usually first-in-first-out (FIFO) but also priority is allowed. Besides, the consumer can select the messages of a queue based on properties of the message.

In Figure 1 it is showed main operations of both systems.

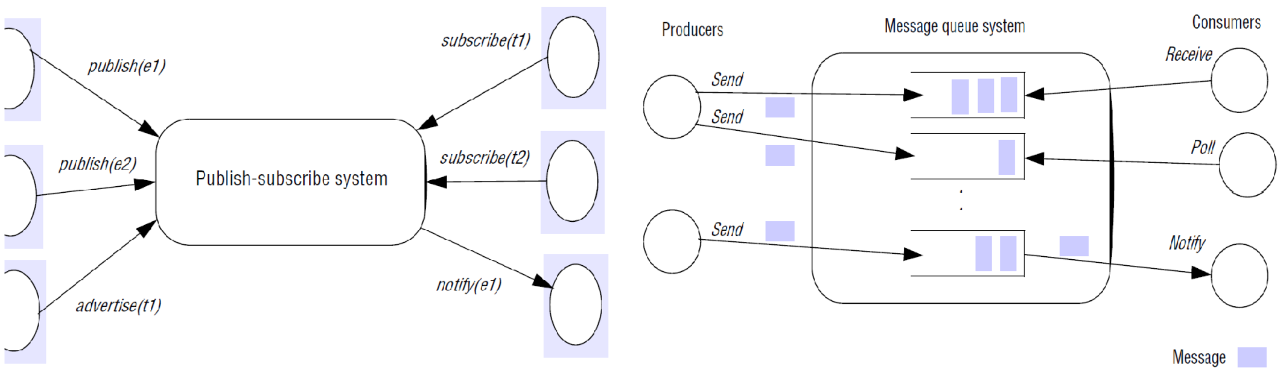


Figure 1. Comparison of operations in publish subscribe system vs message queues (1).

Referring to architecture, in **publish subscribe system** there are two possibilities: the centralized is the simplest one, based in a single node (broker) and which receives and sends all notifications. By this way, it is implemented by point-to-point messages. To avoid the bottleneck that the broker can be, a workaround is to use a network of brokers. Other kind of architecture is distributed, and it is more complex. There are a wide variety of implementation approaches.

In message queues there are also both possibilities: centralized and distributed. It can be centralized in one or more message queues managed by queue manager or distributed like in WebSphere MQ approach detailed in (1).

Finally, two examples of **publish subscribe system** are: RSS feeds and large-scale dissemination of events (like in advertisements of Google) are examples of publish-subscribe systems.

In case of **message queues**, the most common commercial product implementation is *IBM's WebSphere MQ* (as mentioned before, managed by queue managers).

Java Messaging Service (JMS) is an example of a middleware specification that supports both message queues and publish-subscribe.

4. Question 4

Read the following paper on MapReduce and answer the following questions. https://static.usenix.org/publications/library/proceedings/osdi04/tech/full_papers/dean/dean.pdf
a) Explain the main idea and need of MapReduce and how it is implemented.

MapReduce is a programming model which support the process and generation of large data sets. In order to use it, it is needed to define two functions: map and reduce.
The map function processes a key/value pair to generate a set of intermediate key/value pairs. The reduce function accepts an intermediate key and a set of values for that key and merges all values to form a possibly small set of values (typically just zero or one output value). The output of both functions are in different domain as the input.

The library provides support for reading input data and write output data in several different formats and it is opened by interfaces allowing users to define a new reader or writer. Besides, it has different functionalities like one to make easier to log and debug MapReduce operations, other like to show the information about the status of the operations, etc.

MapReduce is easy to use for programmers. Tasks such as the partition of input data, schedule the program's execution distributed in different machines and handle machine failures are carried out by the run-time system. This is one of the reasons that for instance, Google uses it in their clusters every day. Furthermore, mapReduce is highly scalable.

Focusing on the execution the process is:

1. MapReduce splits the input files into M pieces and start many copies of the program on a cluster of machines.

2. One copy is the master and the rest are workers managed by master. Master assigns each of M map tasks and R reduce tasks to different idle workers.

3. A worker with a map task assigned read the input split, parses key/value pairs and execute the map function. The intermediate key/value pairs produced are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions and the locations are sent to master who forwards these to the reduce workers.

5. Reduce worker receives this location and uses remote procedure calls to read the buffered data and sorts it by intermediate keys having grouped the occurrences of the same key. The sorting is needed because it is common to have different keys map sort to the same reduce task. If amount of intermediate data is too large to fit in memory is used an external sort.

6. Reduce worker iterates the sorted intermediate data of the previous step and for each unique intermediate key, passes this and the set of intermediate values to Reduce function. The output is appended to a final output.

7. When all the map and reduce tasks are finished, the master wakes up the user program and go back to the user code.

b) How does MapReduce handle failure of workers?

The possible failure of machines is considered by the library and is resilient to large-scale worker failures. Periodically, master pings every worker and waits certain amount of time. If worker do not response, master will mark the worker as failed, and this task is reset to idle and becomes eligible to reeschedule. When a worker has finished his work, it is reset back to idle state and it can be eligible to do other operation. Furthermore, when a map task is executed and this machine fails, apart from execute the same operation in other worker, reduce tasks are notified.

If a machine with completed map tasks failure, the map operation is re-executed because the output is inaccessible because is on the local disk of this machine. However, if this happen in completed reduce tasks it is not necessary to re-executed because the output is in a global file system.

Referring to master failure, it can be considered to save checkpoints, and if master fails, a new copy can start from the last checkpoint. The implementation described in the article (2) does not do this and responsible clients to retry if they desire in this case.

In order to have the same output although possible failures, it is considered to do atomic commits of map and reduce task outputs by the generation of temporary files. If master receives the message of worker with the names of temporary's files of a task already done, ignores it, otherwise, records these to master data structure. In case of reduce tasks, when the task is completed, the temporary file is renamed to final file.

In non-deterministic functions, it can be possible that failures produce different outputs than if the processes are all correct for example.

c) How does MapReduce handle large datasets?

The map phase is divided into M pieces and reduce phase into R pieces. Ideally, M and R should be much larger than the number of worker machines to improve dynamic load balancing and it is better to recovery when a worker fails.

In practice, M tends to have an individual task of 16Mb to 64Mb approximately and R is calculated by a small multiple of the number of worker machines we expect to use.

In the article (2) one case of processing large data is described, the indexing system of Google with more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations and there are described some benefits of using MapReduce library in this case. For instance: the code is simpler and cleaner than using other libraries, the performance is good enough and the indexing process has been converted into one much easier to operate because library deals with machine failures, slow down machines and other problems.

5. Question 5

Apache Spark is an engine used for the analysis of Big Data. Read the following article and answer the following questions.
http://homepages.cs.ncl.ac.uk/paolo.missier/doc/p56-zaharia.pdf
a) Discuss its main design characteristics and how they help Spark to achieve its purposes.

Apache Spark is a project to design an unified engine for distributed data processing. The key programming abstraction is "*Resilient Distributed Datasets*" (RDD), collections of objects partitioned across a cluster that can be manipulated in parallel. By using these RDD, Spark does not need separate engines (SQL, streaming, machine learning and graph processing) to capture a wide range of processing workloads. Spark achieves similar performance and optimizations than specialized processing but running as libraries over a common engine.

The applications that use Apache Spark are many different: from finance applications to scientific data processing passing from combine libraries for SQL, machine learning and graphs.

To create RDDs, users called "transformations" (*map, filter and groupBy*) to their data.
RDD are exposed as functional programming API in Scala, Java, Python and R, and it is only needed to pass local functions to run on the cluster.

Spark evaluates RDDs lazily, it does not execute immediately the computation of a transformation, even if it returns a new RDD object. In every transformation called, Spark creates an execution plan based on the whole graph of transformations to find the most efficient way to execute it. As an example, Spark can join into one execution the multiple operations in a row of map or filter.

RDDs are defined by default as "ephemeral" because they get recomputed each time used in an action (such as count). Nevertheless, users can also persist selected RDD in memory for rapid reuse, considering that if it not fits in memory, it will be saved at the disk).

RDDs are fault tolerant by using an approach called "lineage" that is more efficient than replication in data-intensive workloads because it does not need to write into network. Each RDD tracks the graph of transformations used to build it and if there are any lost partitions, these operations will be reexecuted.

There are different libraries included with Apache Spark, the main four are: SQL and DataFrames, Spark Streaming, Graph and MLlib. Users get used to combining multiple of Spark's libraries to do some powerful applications.

b) How does Apache Spark complement Apache Hadoop (Apache's implementation of MapReduce)?

Spark has a programming model such as MapReduce but completing the functionality with RDD. The main implementation keys are considered in the question above.

The most important benefits of using of Spark are:

- Applications are easier to develop by using a unified API.

- It is more efficient to combine processing tasks than in MapReduce. In MapReduce for example, it requires writing the data to storage to pass it to another engine, instead of Spark, that it can run diverse functions over the same data, often in memory.

- Enables new applications (interactive queries on a graph and streaming machine learning).

Data sharing is the main difference between Spark and MapReduce, however, the individual operations (like map or groupBy) are similar. Data sharing provides an increase of speedup. In the article an example of logistic regression comparing the speed of Spark in front of using MapReduce is described. It is mentioned a new record of the Daytona GraySort challenge, comparing between the faster speed of Spark than MapReduce (and with less machines).

In the article (3), MapReduce is showed as inefficient in sharing data across timestamps because relies on replicated external storage. It is also considered the latency of minutes to hours of MapReduce (for this reason, it is used in batch environments). These two limitations are improved in Spark by using RDDs.

To sum up, RDDs builds on MapReduce's ability to emulate any distributed computation but with an increase of being enough more efficient. The main limitation of Spark is the increase of latency due to synchronization in each communication step, but usually it is not enough to be considered as a problem.

References

1. **George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair.** *Distributed Systems: Concepts and Design.* s.l. : Pearson, 2012.

2. **Ghemawat, Jeffrey Dean and Sanjay.** MapReduce: Simplified Data Processing on Large Clusters. s.l. : Google, Inc.

3. **Several authors.** Apache Spark: A unified engine for Big Data Processing. 2016.