Tuple spaces: Tuple spaces offer a further indirect communication service by supporting a model whereby processes can place arbitrary items of structured data, called tuples, in a persistent tuple space and other processes can either read or remove such tuples from the tuple space by specifying patterns of interest. Since the tuple space is persistent, readers and writers do not need to exist at the same time. This style of programming, otherwise known as generative communication, was introduced by Gelernter [1985] as a paradigm for parallel programming. Several distributed implementations have also been developed, adopting either a client-server-style implementation or a more decentralized peer-to-peer approach.

Whereas distributed shared memory operates at the level of reading and writing bytes, tuple spaces offer a higher-level perspective in the form of semi structured data. In addition, whereas distributed shared memory is accessed by address, tuple spaces are associative, offering a form of content-addressable memory.

Tuple spaces were first introduced by David Gelernter from Yale University as a novel form of distributed computing based on what he refers to as generative communication [Gelernter 1985]. In this approach, processes communicate indirectly by placing tuples in a tuple space, from which other processes can read or remove them. Tuples do not have an address but are accessed by pattern matching on content (content-addressable memory, as discussed by Gelernter [1985]). The resultant Linda programming model has been highly influential and has led to significant developments in distributed programming including systems such as Agora [Bisiani and Forin 1988] and, more significantly, JavaSpaces from Sun (discussed below) and IBM’s TSpaces. Tuple space communication has also been influential in the field of ubiquitous computing.

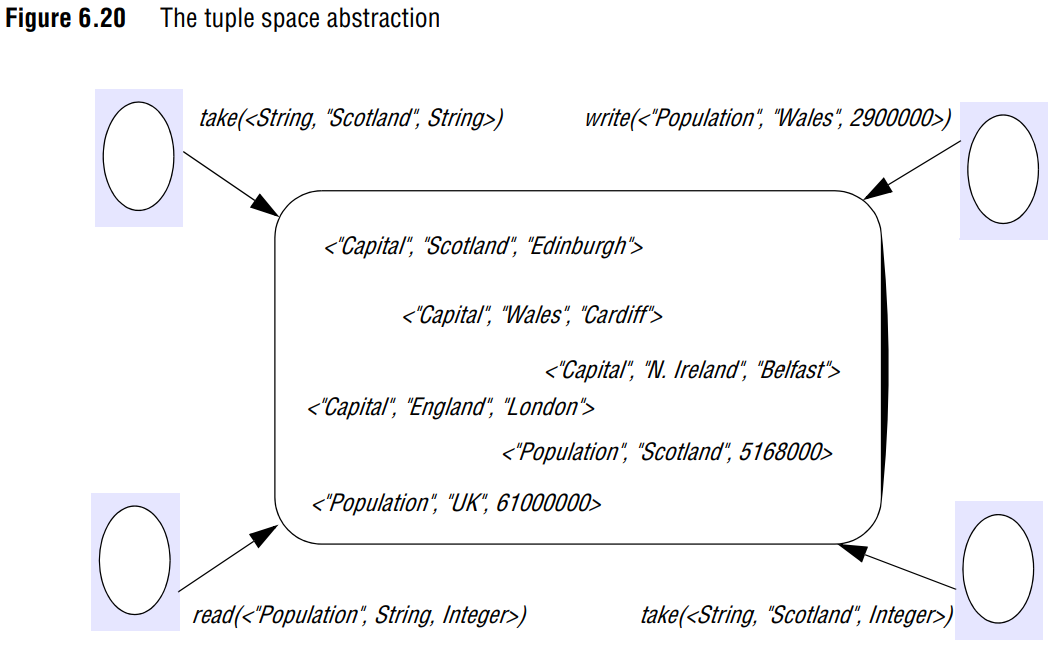
The programming model

In the tuple space programming model, **processes communicate through a tuple space – a shared collection of tuples**. **Tuples in turn consist of a sequence of one or more typed data fields such as <"fred", 1958>, <"sid", 1964> and** **<4, 9.8, “yes”>**. Any combination of types of tuples may exist in the same tuple space. **Processes share data by accessing the same tuple space: they place tuples in tuple space using the write operation and read or extract them from tuple space using the read or take operation**. The write operation adds a tuple without affecting existing tuples in the space. The read operation returns the value of one tuple without affecting the contents of the tuple space. The take operation also returns a tuple, but in this case it also removes the tuple from the tuple space. When reading or removing a tuple from tuple space, a process provides a tuple specification and the tuple space returns any tuple that matches that specification – as mentioned above, this is a type of associative addressing. **To enable processes to synchronize their activities, the read and take operations both block until there is a matching tuple in the tuple space**. A tuple specification includes the number of fields and the required values or types of the fields. For example, take(<String, Integer>) could extract either <"fred", 1958> or <"sid", 1964>; take(<String, 1958>) would extract only <"fred", 1958> of those two.

**In the tuple space paradigm, no direct access to tuples in tuple space is allowed and processes have to replace tuples in the tuple space instead of modifying them. Thus, tuples are immutable**. Suppose, for example, that a set of processes maintains a shared counter in tuple space. The current count (say, 64) is in the tuple <"counter", 64>. A process must execute code of the following form to increment the counter in a tuple space myTS:

myTS.take(<"counter", integer>);

myTS.write(<"counter", count+1>);



Note that ***write***, ***read*** and ***take*** are known as ***out***, ***rd*** and ***in*** in Linda; we use the more descriptive former names throughout this book. This terminology is also used in JavaSpaces, discussed in a case study below.

**Properties associated with tuple spaces:** Gelernter [1985] presents some interesting properties associated with tuple space communication, highlighting both space and time uncoupling as discussed in Section 6.1:

**Space uncoupling:** A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of several potential recipients. This property is also referred to as distributed naming in Linda.

**Time uncoupling:** A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely), and hence the sender and receiver do not need to overlap in time.

**Together, these features provide an approach that is fully distributed in space and time and also provide for a form of distributed sharing of shared variables via the tuple space**. Gelernter [1985] also explores a range of other properties associated with the rather flexible style of naming employed in Linda (referred to as free naming).

Variations on a theme

Since the introduction of Linda, refinements have been proposed to the original model:

• **The original Linda model proposed a single, global tuple space. This is not optimal in large systems, as it leads to the danger of unintended aliasing of tuples: as the number of tuples in a tuple space increases, there is an increasing chance of a read or take matching a tuple by accident**. This is particularly likely when matching on types, such as with take(<String, Integer>), as mentioned above. Given this, several systems have proposed multiple tuple spaces, including the ability to dynamically create tuple spaces, introducing a degree of scoping into the system (see, for example, the JavaSpaces case study below).

• Linda was anticipated to be implemented as a centralized entity but later systems have experimented with distributed implementations of tuple spaces (including strategies to provide more fault tolerance). Given the importance of this topic to this book, we focus on this in the implementation issues subsection below.

• Researchers have also experimented with modifying or extending the operations provided in tuple spaces and adapting the underlying semantics. One rather interesting proposal is to unify the concepts of tuples and tuple spaces by modelling everything as (unordered) sets – that is, tuple spaces are sets of tuples and tuples are sets of values, which may now also include tuples. This variant is known as Bauhaus Linda [Carriero et al. 1995].

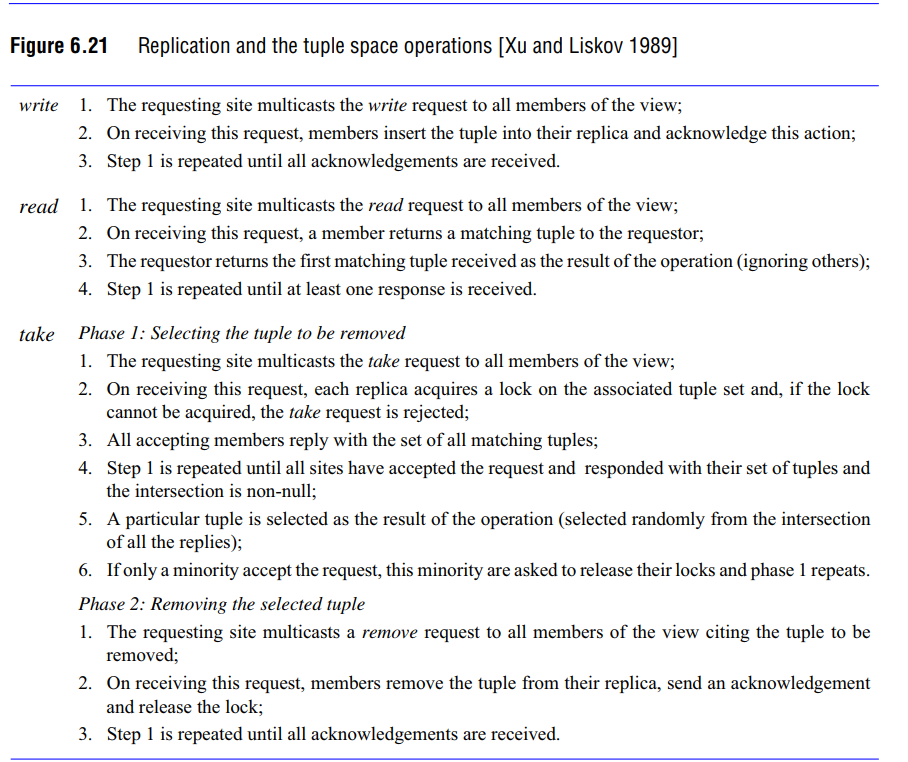
• Perhaps most interestingly, recent implementations of tuple spaces have moved from tuples of typed data items to data objects (with attributes), turning the tuple space into an object space. This proposal is adopted, for example, in the influential system JavaSpaces, discussed in more detail below.

Implementation issues

Many of the implementations of tuple spaces adopt a centralized solution where the tuple space resource is managed by a single server. This has advantages in terms of simplicity, but such solutions are clearly not fault tolerant and will not scale. Because of this, distributed solutions have been proposed.

**Replication:** Several systems have proposed the use of replication to overcome the problems identified above [Bakken and Schlichting 1995, Bessani et al. 2008, Xu and Liskov 1989]. The proposals from Bakken and Schlichting [1995] and Bessani et al. [2008] adopt a similar approach to replication, referred to as the state machine approach and discussed further in Chapter 18. This approach assumes that a tuple space behaves like a state machine, maintaining state and changing this state in response to events received from other replicas or from the environment.

**To ensure consistency the replicas (i) must start in the same state (an empty tuple space), (ii) must execute events in the same order and (iii) must react deterministically to each event**. **The key second property can be guaranteed by adopting a totally ordered multicast algorithm**, as discussed in Section 6.2.2. **Xu and Liskov [1989] adopt a different approach, which optimizes the replication strategy by using the semantics of the tuple space operations. In this proposal, updates are carried out in the context of the current view (the agreed set of replicas) and tuples are also partitioned into distinct tuple sets based on their associated logical names (designated as the first field in the tuple)**. **The system consists of a set of workers carrying out computations on the tuple space, and a set of tuple space replicas. A given physical node can contain any number of workers, replicas or indeed both; a given worker therefore may or may not have a local replica. Nodes are connected by a communications network that may lose, duplicate or delay messages and can deliver messages out of order. Network partitions can also occur.**



**A write operation is implemented by sending a multicast message over the unreliable communications channel to all members of the view. On receipt, members place this tuple into their replica and acknowledge receipt. The write request is repeated until all acknowledgements are received.** For the correct operation of the protocol, replicas must detect and acknowledge duplicate requests, but not carry out the associated write operations.

**The read operation consists of sending a multicast message to all replicas. Each replica seeks a match and returns this match to the requesting site. The first tuple returned is delivered as the result of the read.** This may come from a local node, but given that many workers will not have a local replica, this is not guaranteed.

**The take operation is more complex because of the need to agree on the tuple to be selected and to remove this agreed tuple from all copies.** The algorithm proceeds in two phases. **In phase 1, the tuple specification is sent to all replicas, and the replica attempts to acquired the lock on the associated tuple set to serialize take requests on the replicas** (write and read operations are unaffected by the lock); **if the lock cannot be acquired, the take request is refused. Each replica that succeeds in obtaining the lock responds with the set of matching tuples. This step is repeated until all replicas have accepted the request and responded. The initiating process can then select one tuple from the intersection of all the replies and return this as the result of the take request**. If it is not possible to obtain most locks, the replicas are asked to release their locks and phase 1 repeats.

**In phase 2, this tuple must be removed from all replicas. This is achieved by repeated multicasts to the replicas in the view until all have acknowledged deletion.** As with write requests, it is necessary for replicas to detect repeat requests in phase 2 and to simply send another acknowledgement without carrying out another deletion (otherwise multiple tuples could erroneously be deleted at this stage). The steps involved for each operation are summarized in Figure 6.21. **Note that a separate algorithm is required to manage view changes if node failures occur or the network partitions** (see Xu and Liskov [1989] for details). **This algorithm is designed to minimize delay given the semantics of the three tuple space operations**.

* read operations only block until the first replica responds to the request.
* take operations block until the end of phase 1, when the tuple to be deleted has been agreed.
* write operations can return immediately

**This**, though, **introduces unacceptable levels of concurrency**. For example, a read operation may access a tuple that should have been deleted in the second phase of a take operation. Therefore, additional levels of concurrency control are required. **Xu and Liskov [1989] introduce the following additional constraints:**

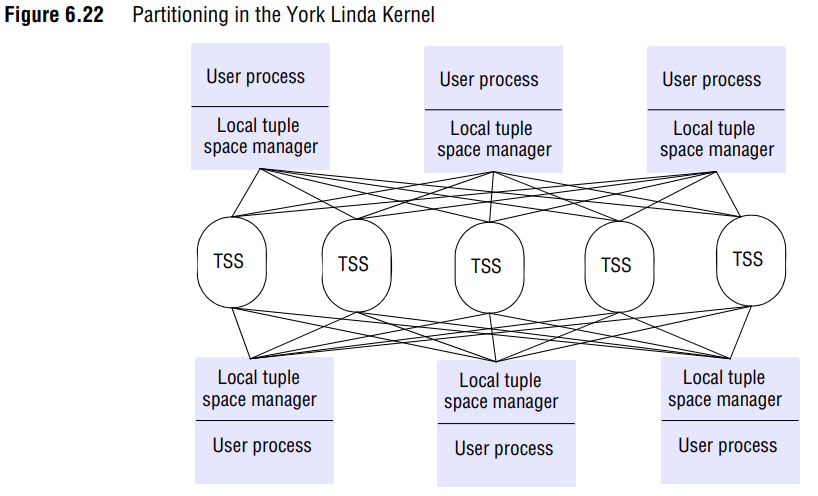
• The operations of each worker must be executed at each replica in the same order as they were issued by the worker;

• A write operation must not be executed at any replica until all previous take operations issued by the same worker have completed at all replicas in the worker's view. A further example of using replication is provided in Chapter 19, where we present the L2imbo approach, which uses replication to provide high availability in mobile environments [Davies et al. 1998].

Other approaches

A range of other approaches have been employed in the implementation of the tuple space abstraction, including partitioning of the tuple space over several nodes and mapping onto peer-to-peer overlays:

• **The Linda Kernel** developed at the University of York [Rowstron and Wood 1996] adopts an approach in which tuples are partitioned across a range of available tuple space servers (TSSs), as illustrated in Figure 6.22.



**There is no replication of tuples; that is, there is only one copy of each tuple. The motivation is to increase performance of the tuple space, especially for highly parallel computation**. **When a tuple is placed in tuple space, a hashing algorithm is used to select one of the tuple space servers to be used. The implementation of read or take is slightly more complex, as a tuple specification is provided that may specify types or values of the associated fields. The hashing algorithm uses this specification to generate a set of possible servers that may contain matching tuples, and a linear search must then be employed until a matching tuple is discovered. Note that because there is only a single copy of a given tuple, the implementation of take is greatly simplified.**

*Note: See more on Tuples Spaces on Chapter 19 for Mobile and Ubiquitous Computing* around pages 838-842