Tuple spaces and Object spaces

Piet van Oostrum

Oct 21, 2008

Tuple space

- Tuple
  - Ordered sequence of (typed) values
  - Usually it has a "name" (first element)
- Tuple space
  - Bag of tuples
  - Identical tuples may be present multiple times
  - Tuple space is associative memory
    - i.e. search on contents only, not on identity
- Linda program
  - Works on collection of ordered tuples
  - Tuple contains data (or possibly code)
  - Consists of a number of cooperating processes or threads
  - Only communication is by putting tuples in and getting tuples out of tuple space
- Linda is an abstract concept (ideas)

Mechanisms 1

- Putting

\[ \text{out}( N, P_2, \ldots, P_j ) ; \]

- out statement puts a tuple in tuple space
- Continues immediately
- First parameter is a name (string)
- Instead of values also 'formals' in the form \( i: \) integer may be given
- This is an unspecified but typed part of the tuple (wildcard)

Mechanisms 2

- Withdrawing

\[ \text{in}( N, P_2, \ldots, P_j ) ; \]

- in statement withdraws a tuple from tuple space
- Parameter is a template: contains values and typed variables
- Matching based on value parts
- Blocks until matching tuple is available
- Variables are filled with tuple values
- Tuple is removed from tuple space

- Reading

\[ \text{read}( N, P_2, \ldots, P_j ) ; \]

- read statement same as in
- But does not remove tuple from tuple space

Communication

No direct communication between A and B
Properties

- Pattern matching
  - $$\text{in}(P, i:\text{integer}, j:\text{integer})$$
    - Matches (+withdraws) tuple with name "P" and any integer value on the rest
  - $$\text{in}(P, i:\text{integer}, \text{FALSE})$$
    - Matches (+withdraws) tuple with name "P", any integer value in second place and FALSE in third place
    - Tuples with wildcards match any actual value in in statement
    - Compare with Prolog: binding of free variables

System characteristics

- Nondeterminism
  - Which process succeeds $$\text{in}()$$'s or $$\text{read}()$$'s
  - Which tuple you get when there are more matches
- Atomicity
  - Of insertion and withdrawing
- Sender and receiver do not know each other
- Distribution in space
  - Shared memory alike
- Distribution in time
  - Mailbox alike: once put, stays forever until withdrawn
  - No explicit synchronisation
  - Communication possible after sender dies
  - Semantics is independent from timing (modulo nondeterminism)

Process notation 1

- Simple statement
  - $$\text{in}()$$
  - $$\text{out}()$$
  - $$\text{read}()$$
- Compound statement
  - $$[\ldots]$$

Process notation 2

- Combinations
  - Sequence
    - $$s_1; s_2; \ldots; s_j$$
  - Concurrency (and parallelism):
    - All $$s_i$$ run parallel
  - Mutual exclusive concurrency (or parallelism):
    - Only one of the $$s_i$$ will run (one that is ready) non-deterministically
    - Blocking $$\text{in}$$ or $$\text{read}$$ will not be chosen
    - If all $$s_i$$ block then the whole statement blocks
  - $$s_1 | s_2 | \ldots | s_j$$

Process notation 3

- Repetition
  - $$*s$$
  - Input and action syntactic sugar
    - $$*[\text{in( -t )} ; s]$$
    - $$\text{in( -t )} \Rightarrow s$$
  - Macro
    - def SYNCH_SEND(s:tuple) $$[\ldots]$$

Small examples

- Remote procedure equivalent
  - Client side
    - $$\text{out}(P, me, \ldots)$$
    - $$\text{in}(me, \ldots)$$
  - Server side
    - $$\text{in}(P, who:name, \ldots) \Rightarrow$$
      $$[\text{body} ;$$
      $$\text{out}(who, \ldots)$$
    $$]$$
    - $$me$$ is replaced by unique process identification
    - How to do Remote Method Invocation (RMI)?
Semaphore equivalent

- P(sem)
  - `in( sem )`
- V(sem)
  - `out( sem )`

Explicit synchronisation

- Rendez-vous style of communication
  - Communication style in Ada programming language
  - 'client' does `proccname(p1, p2, ...)`
  - 'server' does
    - accept `proccname(p1, p2, ...)` do statements end
    - parameters `p1, p2, ...` may be in or out
    - client and server wait until they are both ready before data transfer
    - both client and server continue only after exchange of data completed and accept body executed

Simplified emulation (if only one pair of communicating processes):

```plaintext
def SYNCH_SEND(s:tuple)
  [ in(get); out(s); in(got) ]
def SYNCH_RECV(s:tuple)
  [ out(get); in(s); out(got) ]
```

Disk head scheduler

- Active monitor, filtering/sorting requests
  - Note: A −→ B means messages go from A to B through tuple space, not direct communication.

```plaintext
out( ... , me , ... )
in( me , ... )
```

Client side

```plaintext
def DISK_SVC( t:track; d:data )
  [ out( disk_req, me, t, d ) ;
in( me )
]
```

Disk head scheduler – Continued

Server side

```plaintext
DISK_FILTER & DISK
```

```plaintext
def DISK
  * [ out( get_diskreq ) ;
in( disk_driver, who:name, i:track, d:data ) ;
perform physical i/o ;
out( who )
]
```

Disk head scheduler – Continued

```plaintext
def DISK_FILTER
  DECLARATIONS & * [ DISK_ENQUEUE | DISK_DEQUEUE ]
def DECLARATIONS
  disk_idle:boolean = false
  & other necessary data structures
def DISK_ENQUEUE
  [ in( disk_req, who:name, i:track, d:data ) ;
if ( disk_idle )
  [ out( disk_driver, who, i, d ) ;
disk_idle = false
] else insert who, i, d in i/o queue
```

Disk head scheduler – Continued

```plaintext
def DISK_DEQUEUE
  [ in( get_diskrequest ) ;
who:name, i:track, d:data &
  [ if queue is empty
disk_idle = true
else
  [ who, i, d = get and remove from queue ;
out( disk_driver, who, i, d ) ;
  ]
]```
Linda implementation

- All the tricky stuff is hidden in an implementation
- Absence of shared memory
  - How to find tuples?
  - Where to keep tuples?
- Naming

```c
in( xx, ... )
```

- Implies naming system able to look for data named `xx`
- Naming system: distributed directory/database

Linda implementation

- Tuple location
  - Tuple availability (via `out()`) or tuple need (via `in()`) must be known everywhere
  - Either broadcast when information is available
  - Or ask (everywhere) for it
  - Or in between
- Other issues
  - Buffering (of many `out()`’s)
  - Failure, transaction problems

Linda extensions – 1

- Non-blocking operations
  - Original Linda had a **non-blocking** `in`, called `inp`
  - In case there is no matching tuple this returns a 'not-found' indication.
  - Was thought to be nice to prevent eternally blocking processes
  - This makes Linda programs time-dependent, however
  - Similar for `read`
  - Definitions of `inp` and `rdp` where ambiguous
- Other interpretation
  - `inp` blocks until a tuple is available or it can be proven that there will never be one (a deadlock occurs)
  - In case of a deadlock one participant will receive a failure
  - this can only be done if all Linda processes are known
  - This interpretation tries to remove the time-dependency

Linda extensions – 2

- Multiple Tuple Spaces
- Good for modularity
- Tuples spaces in tuples (recursion)

![Nested tuplespaces](image)

![Flat tuplespaces](image)

PyLinda

- A Linda(-like) system written in Python
- Multiple tuple spaces
- Tuple spaces live in servers
- Servers connect to a master server
- `inp` and `rdp` extensions with deadlock detection
- bulk operations

```bash
# start the master server
> linda_server
# start server connected to server at <ip-address>
> linda_server -c <ip-address>
```

A Linda master server has a main tuple space called “universe”.

Linda extensions – 3

- Bulk Tuple Operations
  - The “multiple read problem” is the problem that you can’t reliably do a `read` for a collection of tuples.
  - You might get the same tuple multiple times
  - So you must use `in` multiple times
  - And then put back the tuples
  - You can’t do it atomically
- Solution: bulk operations:
  - `collect (T1, T2, template)`: moves all tuples that match `template` from tuple space `T1` to tuple space `T1`
  - `copy-collect (T1, T2, template)` makes a copy
PyLinda
Client:

```python
import linda
linda.connect()
linda.universe._out((1, 2, 3))
linda.universe._in((1, 2, int))
```

- Template "wild cards" are given by a type instead of a value.
- The _in returns a tuple.

JavaSpaces
- Tuple Space + Java
- On top of
  - Serialization + RMI
  - Events & Transactions
- Differences w.r.t. Linda
  - Multiple spaces
  - Tuple: class Entry, typed, fields may be objects
  - Matching via templates (with wildcards)
  - Notification mechanism for availability of tuple (instead of blocking read)
  - Transaction properties
  - Identity mechanism for security

JavaSpaces – Usage
- Writing (out)

```java
JavaSpace space = getSpace();
AttrEntry e = new AttrEntry();
e.name = "DOS";
e.value = new GIFImage("DOSexample.gif");
EntryRep erep = e.rep(); // or new EntryRep(e)
space.write(erep, null, null);
```

- Tuple = Entry, out = write
- AttrEntry is subclass of jive.javaSpace.Entry
- Match implies class ‘equality’
- Entry is packaged into EntryRep before handing it over to a space

JavaSpaces – Usage
- Taking (in)

```java
JavaSpace space = getSpace();
AttrEntry e = new AttrEntry();
e.name = "DOS";
e.value = null; // wildcard
EntryRep erep = e.rep();
space.take(erep, null, null);
```

- in = take
- null value means wildcard
- Parameters for transaction & identification
Purpose: allow groups of services and users to federate into a single, dynamic distributed system (Jini community)

Goals
- Simplicity of access
- Ease of administration
- Support for easy sharing – “spontaneous” interactions
- Self-healing of Jini communities

Main operations
- Discovery: find a lookup service
- Join: register your service with a lookup service
- Lookup: find a service in the lookup service

Done by type: Java interface type
- Local object (like CORBA proxy/stub) returned to client
- Invoke: use the local object to call the service

Jini – scenario
- Device is plugged into the network
- Device registers service(s)
- Class to use the service is copied to the Lookup Service

Jini – scenario
- Two devices have been registered
- There UI classes have been copied to the Lookup Service

Jini Example
- Start: one service – lookup – running on network
- Printer starts up
  - Finds lookup service
  - Registers self with lookup service (no user intervention)
- Laptop with word processor enters room
  - Word processor finds lookup service
  - Word processor looks up printer
  - Word processor can also optionally
    - Register to get callback if printer goes away
    - Register to get callback if a new printer registers itself
  - Word processor invokes printer (sends it a printer job)
- Printer (not word processor) controls dialog box – only it knows what it should look like, perhaps in ways not known when word processor made
Jini code example (Client)

```java
import net.jini.core.entry.*;
import net.jini.core.lookup.*; // etc.
try {
    System.setSecurityManager(new RMISecurityManager());
    lookup = new LookupLocator("jini://localhost");
    registrar = lookup.getRegistrar(); // Lookup Service
    id = registrar.getServiceID();
    aeAttr = new Entry[1];
    aeAttr[0] = new Name("MyServer");
    // First 2 params: ServiceID (UUID) and class (servicetypes).
    template = new ServiceTemplate(null, null, aeAttr);
    myServerInterface =
        (MyServerInterface) registrar.lookup(template);
    if (myServerInterface instanceof MyServerInterface)
        { ... myServerInterface.sayHello() ... }
}
```

Jini code example (Server)

```java
public class MyServer extends UnicastRemoteObject
    implements MyServerInterface, ServiceIDListener {
    public String sayHello() throws RemoteException {
        return ("Hello World from MyServer!");
    }
    public static void main(String[] args) {
        int i;
        MyServer myServer;
        Entry[] aeAttr;
        JoinManager joinmanager;
    }
```

Jini code example (Server) contd.

```java
try {
    System.setSecurityManager(new RMISecurityManager());
    aeAttr = new Entry[1];
    aeAttr[0] = new Name("MyServer");
    myServer = new MyServer();
    joinmanager = new JoinManager(myServer, aeAttr,
        myServer, new LeaseRenewalManager());
    // Third parameter is a ServiceID or ServiceIDListener
    ...
} catch (Exception e)
```

Jini – facilities

- Leasing: automatic garbage collection
  - Service granted for a limited period of time: a lease
  - If lease not renewed (it expires), resources freed
- Transactions
  - Two-phase commit
  - Note: Jini, and JavaSpaces are not databases
  - Jini (JavaSpaces) supports full transactions (two-phase commit)
- Events
  - Can register for callbacks for events of interest

References

- David Gelernter, Generative Communication in Linda. ACM TOPLAS 7(1), Jan 1985, pg 80-112
- Javaspaces http://java.sun.com/docs/books/jini/javaspaces/