Assignment 3 — Tuple space

T-106.5600 Concurrent programming

Jan Lönnberg

Department of Computer Science and Engineering
Helsinki University of Technology

10th November 2009

Slides by Jan Lönnberg and Teemu Kiviniemi.

Introduction

The Story

- You have been assigned to write a prototype for a new platform-independent chat system.
- You must use tuple spaces to distribute messages.
- Your job is to implement a basic tuple space system and to get the message distribution part of the chat system running.

Introduction

You get:

- Tuple space API specification.
- Chat system API specification.
- Network tuple space server.
- Chat system testing UI that connects to the tuple space server.
- Test package.

Introduction

Your task is to:

- Implement tuple spaces.
- Implement a chat system backend using tuple spaces.
- These are two separate tasks; your tuple space implementation must work with any compliant chat system implementation and vice versa.

Tuple space

Tuple space libraries for Java

- JavaSpaces (in Jini/Apache River)
- SemiSpace
- lighTS/Polyester

Pronounciation

From splits most subtle, a marriage can topple:
First rabble, then rubble, theirs soon wasn't worth a ruble.
She stuck hard to Tupple, he pushed still for Toople;
Two tickets to Reno corrected the trouble.

Bertrand Meyer: On an open issue of programming language phonetics. Journal of Object Technology 2(2):109–110 (2003)

Tuple space API

You implement:

- `public class LocalTupleSpace implements TupleSpace`
  - `public LocalTupleSpace()`
    - Create an empty tuple space.
  - Methods `get` and `put` are defined in the interface `TupleSpace`.

Tuple space API

Methods of `public interface TupleSpace`

- `public void put(String[] tuple)`
  - Insert `tuple` in tuple space. The tuple in the tuple space must be unaffected by changes to `tuple` after `put` is complete.
  - `tuple` is a `String` array of any length greater than zero.
  - `tuple` may not be `null` or contain `null` values.

Tuple space API

Methods of `public interface TupleSpace`

- `public String[] get(String[] pattern)`
  - Remove and return a tuple matching `pattern` from tuple space. Block until one is available.
  - `pattern` may not be `null`, but may contain `null` values.
  - A tuple matches `pattern` if both have the same amount of entries and every entry matches.
  - An entry matches if either:
    - `pattern[position]` is `null`.
    - `pattern[position].equals(tuple[position])` is `true`.
  - If several matching tuples are found in the tuple space, any one of them may be returned.

Tuple space examples

Examples

- Tuples in the tuple space:
  1. `{ "tuple" }`
  2. `{ "space" }`
  3. `{ "pattern", "matching" }`
  4. `{ "pattern", "matching", "demo" }`
- Patterns:
  - `{ "tuple", null }` (matches none of the tuples)
  - `{ null }` (matches either 1 or 2)
  - `{ null, "matching" }` (matches 3)
  - `{ "pattern", "matching", null }` (matches 4)

Distributed use

Simple, safe and distributed

- Keeping all non-local data in the tuple space means that it does not matter whether other threads are in the same process.
- Local variables in methods can, of course, be used safely.
- Note that `synchronized` has no effect on other processes.

Distributed use

Example

```
public class MyClass {
    private final TupleSpace ts;
    public MyClass(TupleSpace ts) {
        this.ts = ts;
    }
    public void myMethod() {
        String[] myPattern = { "me", null };
        String[] myTuple = ts.get(myPattern);
        // ...
        ts.put(myTuple);
    }
}
```

Inconsistent order of `get()`s

Example

```
public void method1() {
    String[] t1 = ts.get(pattern1);
    String[] t2 = ts.get(pattern2);
    // ... put back when done
}
public void method2() {
    String[] t2 = ts.get(pattern2);
    String[] t1 = ts.get(pattern1);
    // ...
}
```

- To avoid deadlocks, the order of `get()` operations has to be consistent.

Inconsistent order of `get()`s

Example

```
public void method1() {
    String[] t1 = ts.get(pattern1);
    String[] t2 = ts.get(pattern2);
    // ...
}
public void method2() {
    String[] t1 = ts.get(pattern1);
    String[] t2 = ts.get(pattern2);
    // ...
}
```

- This avoids the deadlock.

Race conditions with too eager `put()`ting

Problem

- If tuples are `put()` back in tuple space too early, race conditions may happen.

```
String[] someTuple = ts.get(somePattern);
// critical section CS1
ts.put(someTuple);
// anything can happen here!
String[] otherTuple = ts.get(otherPattern);
// critical section CS2
ts.put(otherTuple);
```

- If something was true inside *CS1*, it can't be assumed inside *CS2*.

Race conditions with too eager `put()`ting

Solution

- It's usually a lot safer to do it like this (but remember to use a consistent order of `get()` operations):

```
String[] someTuple = ts.get(somePattern);
String[] otherTuple = ts.get(otherPattern);
// CS1 + CS2
ts.put(otherTuple);
ts.put(someTuple);
```

- The critical sections have been merged. More knowledge of the current state is known.

Problems when removing tuples

Problem and solution

- It's easy to remove a tuple or forget to put it back.
- Another thread may assume that the tuple is still available.
  - The thread will wait forever when it tries to `get()` the tuple.
- Make sure it's not possible to delete a tuple permanently while another thread thinks it's available.

Tuple space performance

Latency and avoiding it

- If the tuple space is accessed over a network, latency may affect performance.
- If each `get()` and `put()` operation has more or less a constant latency, doing more operations decreases performance.
  - Using a single large tuple may be faster than using two smaller tuples.
  - Using fewer tuples increases performance also if the tuple space implementation is very simple (no hashing).
  - Using fewer tuples makes verifying behaviour easier.

Merging tuples

Example

- If the same two tuples are always used at the same time, the tuples should be merged together.
- Merging those tuples increases performance and helps avoiding `get()` ordering problems.

```
public void getSeparate() {
    String[] t1 = ts.get(pattern1);
    String[] t2 = ts.get(pattern2);
}
public void getMerged() {
    String[] tBoth = ts.get(patternBoth);
}
```

Tuple space implementation

Implementing the API

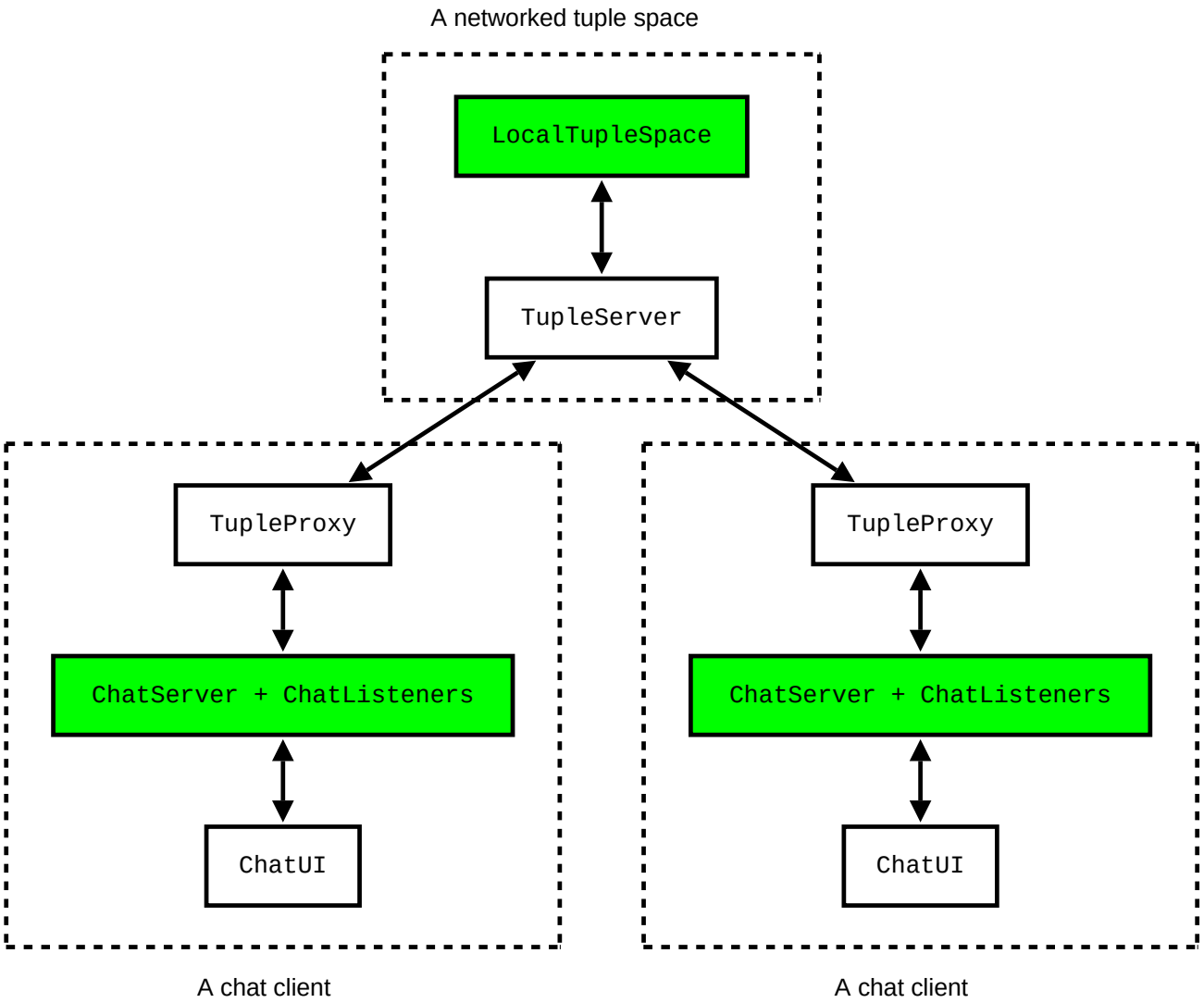
- Use the basic Java synchronisation primitives (those built into the language and `java.lang`, not `java.util.concurrent`).
- Inefficient solutions, such as polling and busy-waiting, will be rejected.
- Use of unsafe methods such as `java.lang.Thread.destroy()` will also lead to rejection.

Chat system

Structure

- Chat system is based on channels.
- Channels are identified by `String`s.
- Each message is sent to one channel.
- Zero or more listeners can also listen to a channel.
- When a listener connects to a channel, it is sent the `rows` last messages sent to the channel.
- After that, the listener receives all messages sent to the channel until it leaves the channel.

Chat system


A networked tuple space
LocalTupleSpace
TupleServer
TupleProxy
ChatServer + ChatListeners
ChatUI
TupleProxy
ChatServer + ChatListeners
ChatUI
A chat client
A chat client

Chat system API

Methods of `public class ChatServer`

- `public ChatServer(TupleSpace t, int rows, String[] channelNames)`
  - Create a new chat server to the empty `TupleSpace t` with a set of channels identified by the `String`s in `channelNames`.
  - For each channel a buffer of `rows` messages is created.
- `public ChatServer(TupleSpace t)`
  - Connect to a chat server using the previously initialized `TupleSpace t`.
  - Uses the channels and buffers already created in the tuple space.

Chat system API

Methods of `public class ChatServer`

- `public String[] getChannels()`
  - Return the list of channels.
- `public void writeMessage(String channel, String message)`
  - Write `message` to `channel`.
- `public ChatListener openConnection(String channel)`
  - Open a listening connection to `channel`.

Chat system API

The `ChatListener`

- `public class ChatListener`
  - `public String getNextMessage()`
    - Get the next message from the channel (wait if necessary).
  - `public void closeConnection()`
    - Close the `ChatListener` and leave the channel.

Chat system API

Communication and synchronisation requirements

- The chat system must be capable of working between machines connected only through the tuple space.
- A `ChatServer` object may be used by several threads concurrently.
- Each `ChatListener` object is used only by one thread.

Running the chat system

How to run the chat system

- To start a new network tuple space server:
  - `java tupleserver.TupleServer`
    - The command prints the TCP port on which the server listens for new connections to standard output.
- To connect to an empty tuple space, and create channels to it:
  - `java chatui.ChatUI <host>:<port> <buffer size> <channel name 1> [<channel name 2> [...]]`
- To connect to a previously initialized tuple space:
  - `java chatui.ChatUI <host>:<port>`

Assignment

Important notice

- Implementing the tuple space and the chat server is two separate tasks.
- Your chat server implementation must work with any compliant tuple space implementation and vice versa.

A tip

- Test your implementations with the tester available on the assignment web page.
- If the tester works correctly with your code, it is probably good. If not, you know there is something you can improve.

Assignment

Doing the assignment

- Group size: 1–2 students.
- Grading does not depend on group size.
- Grade: 0 (fail), 1 (pass) – 5 (pass with honours).

Submission

- Deadline is 2009-12-08 23:59.
- Submit a ZIP archive containing:
  - Your tuple space implementation.
  - Your chat server implementation.
  - Your report.
- There will be no extra round for re-submissions or late submissions.

Assignment

Coding style

- The program should be written in clear, reasonably object-oriented Java.
- Explanatory comments are required for code that is not self-explanatory to a programmer fluent in Java.
- English for variable names, method names, comments and reports is recommended.
- Cryptic method and variable names may not be used.

Assignment

Instructions and requirements

- Full instructions are on the course home page.
- Include a report explaining:
  - The reasoning behind your solution.
  - Measures you have taken to ensure your solution is correct
  - Any unexpected behaviour you may have encountered during testing and how you have investigated and resolved it.
- The instructions on the course web page are the authoritative description of the assignment.

Assignment

Questions and clarifications

- Technical questions and clarification requests should be sent to the course newsgroup:
  `opinnot.tik.rinnakkaisohjelmointi`
- Clarifications (if any) will be posted to the newsgroup.
- Hands-on sessions in Paniikki every Monday 16:15-18:00 until the deadline.

Assignment

Tips

- Read the instructions thoroughly.
- Test your code well. The supplied test package is useful.
- Follow the newsgroup.
- Start working now.

Conclusion

Conclusion

- The assignment is intended to help learn:
  - Monitors, condition variables and Java threads.
  - The use of tuple spaces for data storage, communication and synchronisation.