

Assignment 3 — Tuple space

T-106.420 Concurrent programming

Jan Lönnberg

Department of Computer Science
Helsinki University of Technology

29th November 2005



Introduction

The Story

- Prototype for new platform-independent chat system.
- Use tuple spaces to distribute messages.
- Your job is to implement a basic tuple space system.
- Get the message distribution part of the chat system running.

Goal

- Implement tuple spaces.
- Implement a simple chat system using a tuple space.



Task

You get:

- Tuple space API specification.
- Chat system API specification.
- Chat system testing UI.

Your task is to:

- Implement tuple spaces.
- Implement a chat system backend using tuple spaces.



Task

Doing the assignment

- Group size: 1–2.
- Submission before 2006-01-16 03:00.
- Submit a tar.gz archive containing:
 - Tuple space implementation.
 - Chat system backend implementation.
 - Report in PDF form.
- Full instructions will be on the course home page.
- Grading: fail/pass/pass with honours.



Tuple space API

Java interface

- `public class TupleSpace`
 - `public TupleSpace()`
 - Create an empty tuple space.
 - `public void put(Object[] tuple)`
 - Insert tuple in tuple space.
 - `public Object[] get(Object[] pattern)`
 - Return a tuple (an array of entries) matching `pattern` from tuple space. Block until one is available.
 - A tuple matches a pattern if both have the same amount of entries and every entry matches.
 - A `null` entry in the pattern matches any object in that entry in the tuple.
 - Any other object `p` in the pattern matches any object `t` in the corresponding entry in the tuple for which `p.equals(t)`.



Implementation

Synchronisation

- 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.

Memory space

- Implementation need not be distributed.



Chat system

Structure

- Chat system is based on **channels**.
 - Identified by `Strings`.
 - A message is sent to one channel.
 - A process can also listen to a channel.
 - When a listener connects to a channel, it is sent a log of 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 API

Server interface

- `public class ChatServer`
 - `public ChatServer(int rows, String[] names)`
 - Create a new chat server.
 - Channels identified by the `Strings` in `names`.
 - Buffer of `rows` messages for each channel.
 - `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

Listener interface

- `public class ChatListener`
 - You may assume that each `ChatListener` is used by only one thread.
 - `public String getNextMessage()`
 - Get the next message from the channel (wait if necessary).
 - `public void closeConnection()`
 - Close the `ChatListener`.



Conclusion

Conclusion

- Assignment is intended to help learn monitors and condition variables, Java threads and the use of tuple spaces for data storage, communication and synchronisation.
- Assignment description will be linked from course home page.
- Technical questions and clarification requests to the newsgroup (`opinnot.tik.rinnakkaisohjelmointi`).
- Clarifications (if any) will be posted to the newsgroup.
- Questions not intended for the public to `jlonnber@cs.hut.fi`.

