Rutherford Appleton Laboratory
Computing and Information Systems Department

Threads in a Modular World Wide Web Server

http://www.cis.rl.ac.uk/proj/www/docs/threads.ppt

Simon Dobson

S.Dobson@rl.ac.uk http://www.cis.rl.ac.uk/people/sd/contact.html

Overview

World Wide Web

The STONES architecture for distributed information servers

Threaded aspects of STONES

- protocols
- connections
- administration and advanced services

Benefits and lessons

Conclusions

Who is this man?

Senior Research Fellow in the Computing and Information Systems Department of the Rutherford Appleton Laboratory

Interests

- programming abstractions and languages
- distributed system architectures
- formal aspects, especially type systems, program analysis and transformation

Projects

- TallShiP - high-level sharing for parallel programming
- part of CCLRC's central WWW team

The World Wide Web

Conceived at CERN as a medium for storing and sharing information on large co-operative projects

Has since evolved into the primary way to provide and access data across Internet

- client/server distributed architecture on Internet scale
- hyperlinks between pages
- multimedia - text, images, videos, virtual worlds...
- large repositories of information on most subjects
- search engines for locating pages
- increasing integration with database back-ends

CCLRC's Interest in WWW

We see WWW - and what it may become - as one of the most exciting new areas of computing science

- member of W3C since Spring 1995
- co-founded ERCIM W4G in Autumn 1994

Interests:

- database integration and information retrieval
- graphics (CGM) and virtual reality (VRML)
- enabling protocols and technologies
- page and web design

All Laboratory information is accessible *via* WWW interfaces

The STONES Architecture - Rationale

We want to investigate novel applications of WWW

Existing servers tend to be optimised for performance, rather than for maintenance or extensibility

We needed a well-engineered, modular, extensible WWW server to act as an experimental testbed

- add additional services without disruption
- ease of maintenance
- portability - across platforms **and** languages
- performance is **far** less important to us than these

The STONES Architecture - Design

Different sorts of documents, different protocols

- objects and sub-typing

Configuration at the level of a document store, not a protocol

Activities within the server are mostly independent

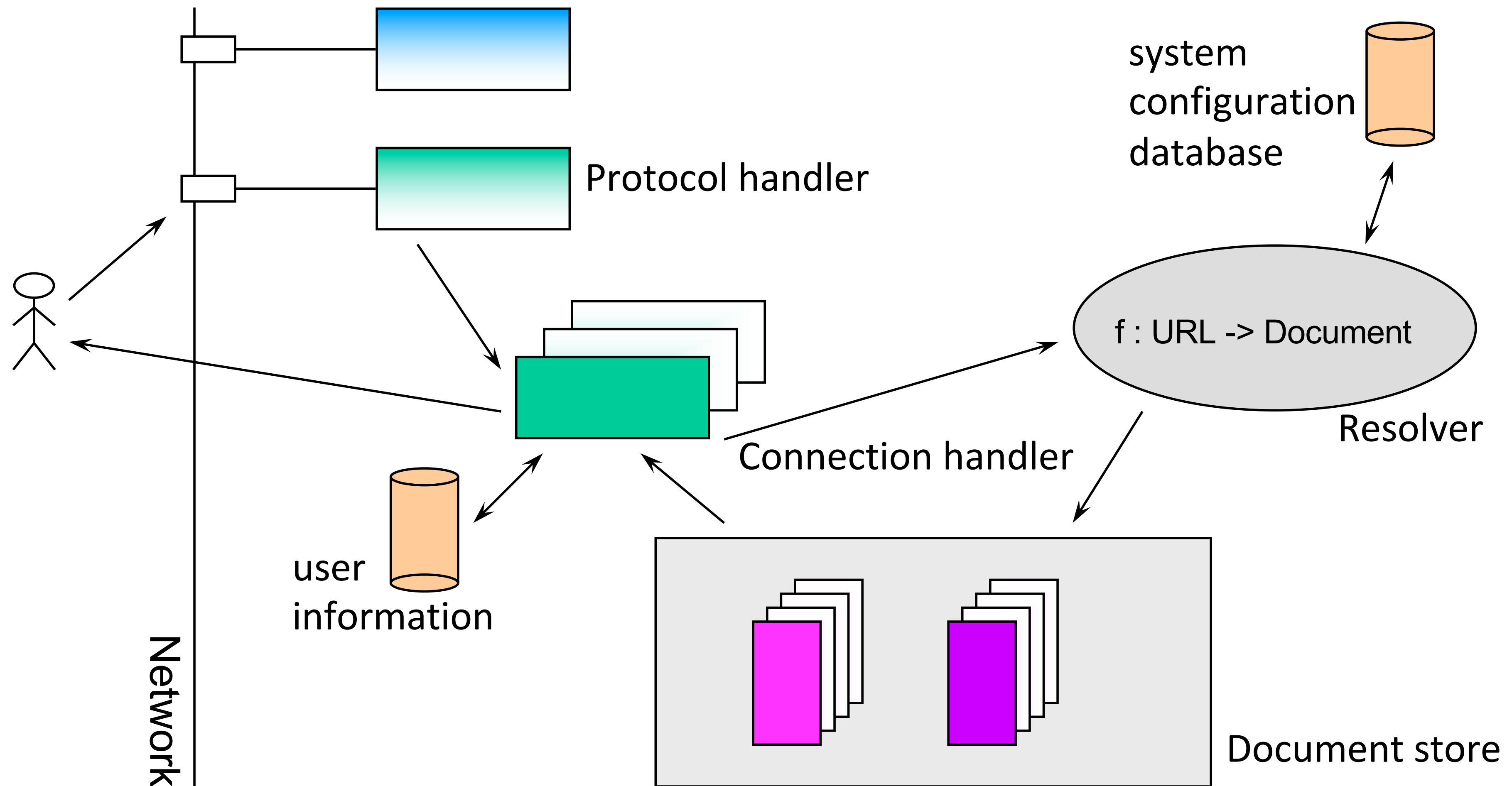
- a thread for each activity (connection, protocol)

Highly changeable configuration

- a thread controlling each “module”
- protect (and minimise) shared data

Threads prominent in the architecture from the beginning

The STONES Architecture - Overview



Multiple Protocols

An information server needs to be able to interact using a number of network protocols

Protocols are independent, in the sense that they can run together, but share a common configuration

Each protocol handler runs in its own thread, creating protocol connection objects as required

- each protocol completely independent - simultaneous support
- shared document store and shared configuration

Adding or removing protocols is trivial

Multiple Connections

Each connection is handled by its own thread

- requests proceed independently once accepted
- latency hiding for document store/network accesses
- time-consuming requests need not impact on other clients

Threads sequence a **single** connection but separate **multiple** connections

Long-lived connections (*i.e.* under FTP) may accumulate information about the client, for better processing. This information is local to the thread handling the interaction

Advanced Services

Consider other services, such as cache maintenance or on-the-fly system optimisation

These are completely independent of request servicing

Without threads one would need to control the interleaving of requests, to prevent a service monopolising the system at the expense of other activities

With threads, one may run such services in parallel with normal request servicing **without change**

Entire server needs to be thread-aware, *e.g.* cache manager shouldn't change a cached document while it's in use

Abstraction

Threads are a vitally important piece of abstraction - as important as procedures, types or objects

A thread isolates a locus of activity, without shutting it off from the application's shared data

Each thread can proceed independently, without other threads being aware of them…

…as long as care is taken so that no data structure is corrupted through concurrent accesses

Threads combine well with event loops - direct events to the appropriate thread, and there process them sequentially

Design Decisions and Trade-offs

It is very easy to get carried away with all this freedom, so remember…***!! Threads cost !!***

Example design decision: do we create a new thread for each new HTTP connection accepted, or have a pool of threads and assign connections to them?

- the former leads to maximum parallelism, but the server can become saturated under heavy load
- the latter is less parallel but easier to manage, as the number of workers is bounded

Note that this is **not** an architectural issue *per se*

Programming with Threads

Most languages and libraries provide laughably inadequate support for programming with threads

What you want:

- high-level management of threads, especially pools
- thread-safe objects and collections

What you get

- a thread handle
- a standard library that isn't thread-safe
- semaphores, or monitors if you're **really** lucky
- a sincere hatred of concurrency

Portability - or lack of it

Different operating systems, and different languages, provide different thread facilities

- threads in the language layer can lead to inadvertent lock-outs, *i.e.* on blocking file accesses
- threads can't always be suspended
- overheads vary wildly

A language with thread support built in (*i.e.* Modula-3) at least gives portability across platforms, but will often only support the lowest common denominator of capabilities

The basic problem is the low-level use of threads

The Good Thread Guide

1. Clearly identify each independent resource and locus of activity **at design time**
2. Create a thread to manage each shared resource
3. Create (or acquire) a thread for each independent activity
4. Unless something is absolutely, definitely, forever private, make sure it’s thread-safe
5. Consider using pools of threads, to simplify management
6. Consider having a thread to watch over your application
7. Remember that threads can unravel

Conclusions

Threads have helped greatly in creating a modular, extensible WWW server

Threads were included from the very start of design, as an aid to abstraction **not** as an aid to performance

In a complex system, threads will almost certainly simplify the management of independent activities…

…and may improve performance too, given the right problem running on the right hardware

Current programming languages just aren't adequate for real applications programming