Network programming

Roger Orr
OR/2 Ltd

Writing programs in a networked world

Goal of this presentation

- Many current programs use network communications – whether implicitly or explicitly
- What does – or should – this change in the way we write our programs?
- We'll look at a number of questions that should be asked when using networks

Why use a network?

- There are many reasons for using a net-work
  - Consumer of remote data or services
    - Time dependent
    - Expensive to copy
  - Producer of shared data
  - Access to different machines
  - Reduce need for physical proximity
  - Better performance
  - Improved resilience

Why use a network?

- There are different types of network, eg:
  - Local LAN
    - Probably TCP/IP
  - Corporate intranet
    - LAN
    - WAN
  - Internet
  - Mobile
  - Interplanetary Internet

Why use a network?

- There are also communications not involving a network
  - Leased line
  - Serial line
  - USB
- Various parts of this talk are also relevant to these scenarios

What the end user wants

- The end user of the program generally wants *transparent* use of the network
  - Indistinguishable from an isolated program
  - Problems should be sorted out without needing user interaction
- This is **not** totally achievable...
- Hiding the network at a higher level API level can also be a mistake

Costs of a network

- The main areas where a network causes issues are
  - Failure modes (connection and remote nodes)
  - Troubleshooting
  - Limitations of physical laws (latency)
  - Security
  - Scalability
  - Interoperability (standards and versioning)

Costs of a network

- Address these issues up front
- It can be expensive (or even impossible) to solve them later
- Making a program 'network aware' will usually affect the **interface** as well as the **implementation**

What can go wrong?

- A stand alone program can be debugged in isolation, or off-line from a dump file
- Networking adds the communications channel **and** independent processes
- Failure modes are more complex
- Partial failure is much more common
  - **Part** of the system is down
  - **Reduction** of performance
- Remote failures may not be in our control

Reducing the pain

- The network **interfaces** are key to good support and maintenance
  - Capturing network traffic
  - Text is easier to read than binary
  - Avoiding complicated cross-process state
  - Proxies and stubs
- Think about what pieces of the system should still work without the whole

Reducing the pain

- Consolidated tracing/logging
  - Machine/process identification
  - Universal timestamps
  - Data reduction
- The **end user** doesn't want to know about the network, but the **support** engineer does
- Can you simplify the configuration?
- How do you test failure modes?

Reducing the pain

- Some examples...
  - Grid [save network packet on failure; log client name and machine; support for local mode]
  - I&K [everything is text, so can easily be saved/replayed; central logger]

Increasing the pain

- An example...
  - Binary protocols across a number of servers
  - It was not apparent which calls were local and which ones were remote
  - No documented design of call hierarchy
  - Errors and exceptions transparently mapped to local errors, or even silently consumed
  - Configuration was sufficiently hard that some developers couldn't get a local installation to work

Troubleshooting

- Networks cause problems but do provide a clean interface to resolve problems
- Network sniffers – for example Wireshark (aka ethereal), tcpdump.
  - Provide a complete trace of the protocol exchange at the lowest level
    - Fault finding
    - Performance analysis
  - Can be hard to relate to application activity

Troubleshooting

- Proactive debugging – what is likely to go wrong and what information will I need?
  - Design communication components independently from business logic
  - 'Ping' methods to separate *connectivity* issues from *application* issues
  - Ensure target details are logged (both IP address + port number)

Limitations of physical laws

- Communication across a network will be slower than that within a process
- The two main measures are:
  - **Throughput** - the amount of digital data per time unit that is delivered over a physical or logical link
  - **Latency** - the time taken for a packet of data to be sent from one application, travel to, and be received by another application

Limitations of physical laws

- Overall throughput is (roughly) the same as the minimum throughput of each part of the communication pathway.
  - Additional throughput can often be bought
- Overall latency is (roughly) the **sum** of the individual latencies
  - Latency usually can't be reduced much
- Most non-technical people don't really un-derstand the difference ...

A worked example

Example interface in Java

```
package multiple;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Contract extends Remote {
  String read1( String key )
    throws RemoteException;
  String[] readn( String... key )
    throws RemoteException;
}
```

A worked example

```
private void testSingle( String[] keys )
  throws RemoteException
{
  String[] result = new String[ keys.length ];
  for ( int i = 0; i != keys.length; ++i ) {
    result[i] = remoteObject.read1( keys[i] );
  }
}

private void testMultiple( String[] keys )
  throws RemoteException
{
  String[] result = remoteObject.readn( keys );
}
```

A worked example

```
Public class Server implements Contract {

  public String read1( String key ) {
    return getData( key );
  }

  public String[] readn( String... key ) {
    String[] result = new String[ key.length ];
    for ( int i = 0; i != key.length; ++i ) {
      result[i] = getData( key[i] );
    }
    return result;
  }
}
```

A worked example

- So what's the difference between using **read1** and **readn**?
- For **one** object
  - A little more work to assemble an array of ob-jects for readn
  - A little more data to pass the network
- For **multiple** objects
  - The loop is written once on the server rather than once in every client
  - Less requests to pass over the network

A worked example

- So what's the difference between using **read1** and **readn**?
- What if you get an exception?
  - **read1**: only the bad requests fail – other data is available
  - **readn**: the whole request fails – may need more work to enforce this for modifications
- Could expand the interface to return an array of objects with failure status

A worked example

Local host

- java -cp . multiple.Client localhost

Single: 6.18 ms / Multiple: 6.60 ms
Single: 5.16 ms / Multiple: 5.57 ms
...
Single: 4.4 (sd 1.7) / Multiple 4.6 (sd 1.2)

A worked example

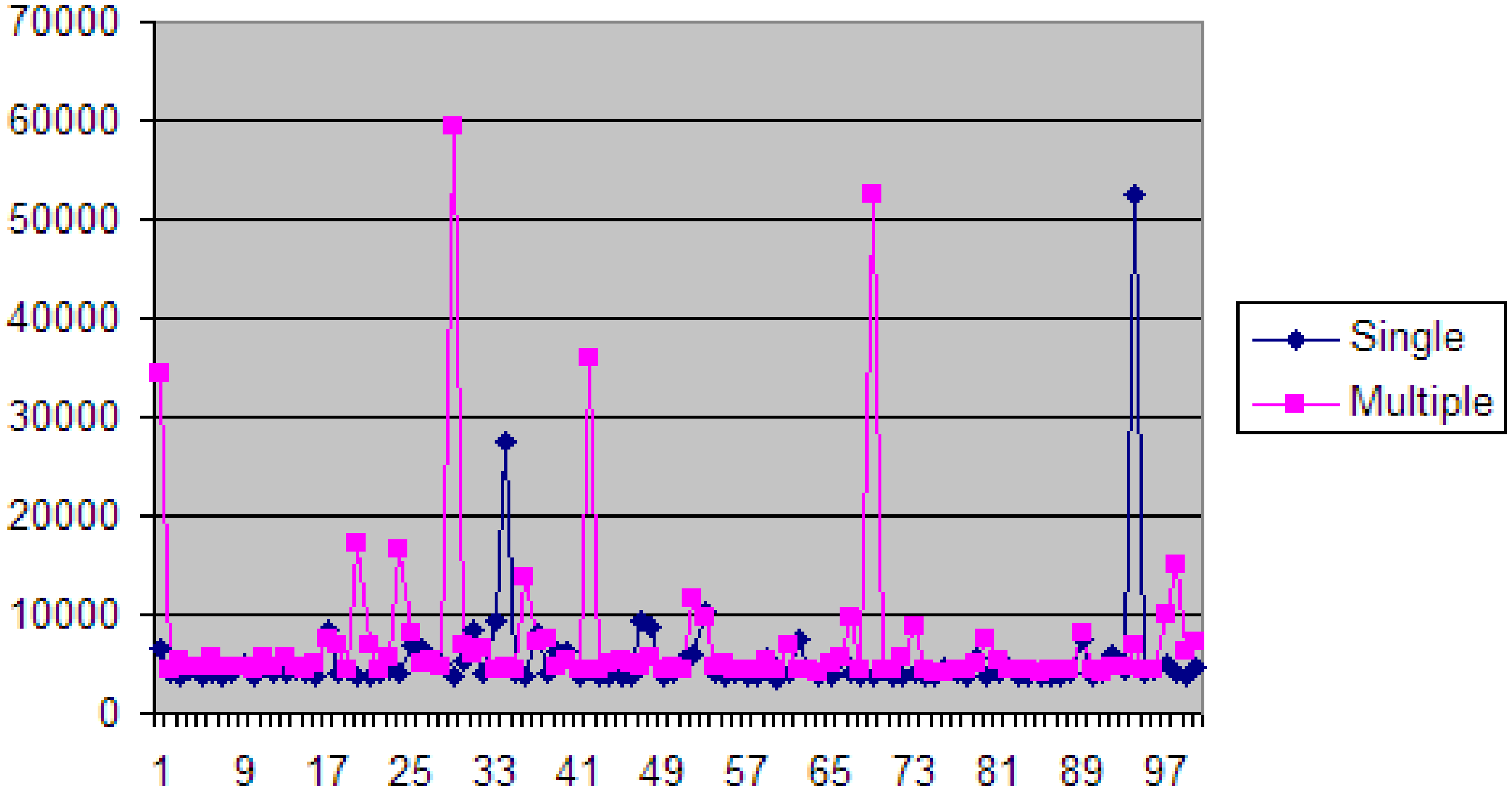
LAN connection

- java -cp . multiple.Client gordon

Single 5.7 (sd 7.4) / Multiple 5.3 (sd 3.7)

- What is going on here?
- Note the large standard deviations
- Even for one call little difference between the 'single' and 'multiple' methods

A worked example



A worked example

- The Nagle algorithm
  - RFC 896: Congestion Control in IP/TCP Inter-networks
- Solves the small-packet problem
  - (1 byte packet, 40 byte header)
  - Often what you want
  - When it isn't it can really hit you badly
- Is this the problem?
- Can I do anything about it?

A worked example

Local host – multiple calls

- java -cp . multiple.Client localhost 10

Single: 28.6ms / Multiple: 3.5ms
Single: 26.0ms / Multiple: 3.4ms
...
Single: 28.5 (sd 7) / Multiple 3.7 (sd 1.6)

A worked example

WAN connection

- java -cp . multiple.Client tokyo

Single: 259ms / Multiple: 259ms

- java -cp . multiple.Client tokyo 2

Single: 517ms / Multiple: 259ms

- java -cp . multiple.Client tokyo 20

Single: 5.2s / Multiple: 261ms

Limitations of physical laws

- Will your solution be used with local, LAN or WAN connections?
- Think about this at **design** time
- Do some simple arithmetic
  - May need to instrument to get data
- **Test early** using the worst case
- Simulate the worst case
  - Buy network simulators
  - Use a simple proxy program

Example: database connection

- Reading several thousand records from the database into cache
- When run remotely the server took over eight minutes longer to start up
  - Running a database remotely would have been an expensive solution
- JDBC supports ResultSet.setFetchSize
  - Using this pretty well restored the local performance remotely

Security

- Accepting input across a network opens up a number of security problems.
- Malicious attacks – principally on the Internet but increasingly internally too
- Data 'leaks'
  - Packet capture
  - Data may be cached locally
- Authentication/authorisation

Security

- Security is a *negative* requirement – it is hard enough to satisfy the more common positive requirements
- Security usually conflicts with other goals, such as supportability
- “There are few, if any, effective strategies to enhance security after design”
  (Wikipedia)

Security

- Obfuscation is *not* a good security choice
- Standard mechanisms are generally safer
- Security is as strong as the weakest link
- However, the weakest link varies depending on access to the system
  - “Ownership is root”
  - Man in the middle attacks
  - Danger of unsecured log files
  - `System Password changed to 'Friday1'`

Security

- Take especial care with user input
  - Cater for escaped characters/special strings
  - Most database APIs provide automatic ways to do this – **always** use them
  - Check string lengths in C-style code
  - Don't trust client side validation

User Security

- Authentication
  - Who is the user
  - How can we be sure
- Authorisation
  - What is the user allowed to do
  - Access control
- Auditing
  - Who did what, when
- Non-repudiation
  - It was me, I cannot tell a lie

User Security

- Often use LDAP access for company internal systems
  - Database probably already exists
  - Tools for many tasks already written
  - Relatively cross-platform / cross-language
- Can be harder on the Internet – lack of common infrastructure
- What might the user **do** with the data?

Security

- How does the system cope with overuse?
  - Denial of service attacks
  - 'Black Monday' market days
  - Run-away success of your product
- Design-in ways to handle such loads
  - Couple of points are covered below
- Test the system behaves properly – the component that fails may not be the one you expected

Scalability

- Networked programs can give advantages of increased scalability
  - Run processes on separate machines
  - Run multiple copies of key processes
- How do we ensure this works?
- Amdahl's law applies here – anything done serially won't scale
- Additionally there is a cost sending the work to another process

Scalability

- Identify the bottlenecks
  - Little point in writing a complex multi-process networked application to update a database if the database is the limiting factor
- Ideal candidate tasks are independent with small 'surface area' (network packet size)
- Cache unchanging data locally
- Shared volatile data is more problematic

Scalability

- Establish some benchmarks using a similar network topology to that proposed
- Decide what is the right behaviour under high load
  - No special treatment (ostrich approach)
  - Prioritize tasks
  - Coalesce tasks
  - Fail certain classes of task
    - Web site falling back to text-only mode
    - Database allowing simple queries only

Interoperability - standards

- Adopting standards for networking is a good thing
  - Good protocol design is hard - or so it seems
  - A lot of corner cases to consider (holes still exist in NetBIOS, DDE and FIX, for example)
  - Lower level code libraries may exist
  - Common protocols may already be supported by protocol analysers

Interoperability - standards

- The Postel dictum:

  “Be liberal in what you accept
  and conservative in what you send”

- Try to accept as wide an interpretation of possible on input
- Try to stick to commonest cases on output

Interoperability - standards

- "The good thing about standards is that there are so many to choose from"
  (A. Tanenbaum)
- Avoid re-inventing the wheel (eg reliable communication on top of UDP)
- May automatically provide possibilities for cooperation
- Prefer higher level abstractions allowing for multiple potential transport protocols

Interoperability - versioning

- Versioning *will* hit you and can be expensive to identify and hard to solve
- Unless you have explicit control over both ends you will end up connecting different versions of the protocol at each end
- A full solution with backward and forward compatibility is difficult: do you need it?

Interoperability - versioning

- Simplest **non** solution – no checks
- Can cause strange behaviour – for example
  - a new parameter is added to a method and old clients implicitly pass in a `null`
  - unrecognised messages may be ignored by the server leaving the client in a pending state
  - Artifacts from a rebuild do not communicate with older objects – *implicit* versioning

Interoperability - versioning

- Simplest solution – check for and reject any connection with the wrong version
- Prefer explicit up-front checks to avoid
  - Delayed failure
  - Callback failure
- This style means all programs must be updated to the correct version simultane-ously (and reverting can be hard)
- Must remember to change the version

Interoperability - versioning

- Multi version server-side solution – for example allow clients to connect using the current or the previous version
- Allows gradual rollout once the server-side components are upgraded
- Increased burden on testing and can be hard to ensure the previous protocol is actually supported consistently
- Must remember to change the version

Interoperability – versioning

- One useful subset is to *add* extra releases that support changes in the protocol (eg extra fields) but do not require them
- This allows two phase update
- Phase I - all components *use* the new protocol version but support both the old and the new versions
- Phase 2 – change some components to *require* the new protocol version

Interoperability example

- The SOAP `mustUnderstand` attribute
- Allows the new version to include some mandatory changes and other optional changes
- Examine the use cases of the interface as may end up with an interface not actually providing any useful functionality to older clients

Interoperability - platforms

- Which network types?
  - For example, will this *only* run on TCP/IP?
- Which Operating System?
  - Word size and byte order issues
  - Support for some protocols better than others
- Which language?
  - Some techniques are inherently multi-language (often using text-based protocols)
  - Single language solutions may support wider functionality (object transport, exceptions)

Interoperability - platforms

- Which language?
  - Some techniques are inherently multi-language (often using text-based protocols) and some standards have multiple language bindings
  - Single language solutions may support wider functionality
    - Local/remote proxying
    - Object transport
    - Remote class loading
    - Transparent handling of exceptions

Conclusion

- Network programming is becoming very common but it needs to be explicitly thought about at the design stage
  - Failure modes (connection and remote nodes)
  - Troubleshooting
  - Limitations of physical laws (latency)
  - Security
  - Scalability

Conclusion – questions

- What are the reasons for using a network in *this* application?
- What might go wrong?
  - What graceful degradation can we offer?
  - How easily will it be to find and fix problems?
- What latency and bandwidth is needed?
- How are we handling security?
- What standards could/should we use?
- What versioning model will we support?