Distributed Computing & Middleware

- Info on CptS 464
- Good background in case you never take 464
Introduction

• A **distributed system** is “one in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing”
  - Very broad definition
  - Lots of examples
  - Lots of kinds

• Abbreviations
  - “Distributed System” by “DS”,
  - “Distributed Computing” is “DC”

• “You know you have one when the crash of a computer you’ve never heard of stops you from getting any work done.” Leslie Lamport
Example Local Call

**Caller:**
// declare and init stuff
x = new int [100];
y = new util;  // create and rtn ref
flag = y.sort(x, 100);

**Callee:**
// declare and init stuff (not shown)
int util:sort(int [] a, int max) {
   // implementation of sort... bubble sort, quicksort, …
   return status;
}

Example Local Call (BACKUP SLIDE)

**Caller:**
// declare and init stuff
x = new int [100];
y = new util;
flag = y.sort(x, 100);

**Callee:**
// declare and init stuff
int util::sort(int [] a, int max) {
    // implementation of sort...
    return status;
}

• Potential assumptions:
  - Object Invocation conventions between caller (“client”) and callee
  - In same address space (on same computer)
  - In same programming language (usually)
  - Written by same programmer (often, not always)
  - Same operating system for both caller and callee
  - Same CPU type for both caller and callee
  - Can transfer data and control quickly, effectively in zero time
  - Both fail, or neither do (for the most part)

• None of these assumptions are always true in a distributed system!
Reminder: Assembler

Equivalent assembler (vars on stack)

Example C-like call

\[ X = 4 + \left( \left( Y \times 4 \right) / \left( A + B \right) \right) \]

\[ \text{ldr r1, [sp, Y]} \quad !\text{load Y} \]
\[ \text{mul r1, r1, #4} \quad !Y \times 4 \]
\[ \text{ldr r2, [sp, A]} \quad !\text{load A} \]
\[ \text{ldr r3, [sp, B]} \quad !\text{load B} \]
\[ \text{add r2, r2, r3} \quad !A + B \]
\[ \text{div r1, r1, r2} \quad !\text{divide the two} \]
\[ \text{add r1, r1, #4} \quad !\text{add four to result} \]
\[ \text{str r1, [sp, X]} \quad !\text{store result in X on stack} \]
Reminder: Calling Conventions

- Calling conventions define this for a given compiler/language
- High-level language compilers do all this for you
- Have to program yourself if using assembler

```c
int main() {
    int x = 1;
    int y = 2;
    int z = myFunc(x, y);
}

int myFunc(int x, int y) {
    return x + y
}
```
Reminder: Calling Conventions

**myFunc:**

```assembly
movl %edi, -4(%rbp) !grab x off stack
movl %esi, -8(%rbp) !grab y off stack
add %esi, %edi !add x and y
movl %esi, %eax !return x + y
ret
```

```assembly
.globl main
main:
  movl $1, -4(%rbp) !x = 1
  movl $2, -8(%rbp) !y = 2
  call myFunc
  ret
```
Example Local Call (2)

- Potential assumptions between caller and callee:
  - Assembler calling conventions
  - In same address space (on same computer)
  - In same programming language (usually)
  - Same operating system
  - Same CPU type
  - Can transfer data and control quickly, effectively in zero time
  - Both fail, or neither do (for the most part)

- None of these assumptions are always true in a distributed system!
**Example Remote Call**

**Caller:**
// declare and init stuff
x = new int [100];
Y = new util.lookup(...);
Flag = y.sort(x, 100);
...

// "proxy" or "stub"
// generated by middleware
int util:sort(int[] a, int max){
  // put a[], max into struct
  // send message with struct
  // wait: message w/ struct
  // copy from struct to a[],
  // status
  return status;
}

**Callee:**
// declare and init stuff
int util_impl:sort(int[] a, int max){
  // implementation of sort
  return status;
}

// "skeleton" generated
// by middleware compiler
...

// receive message with struct
// copy from struct to a[], max
flag = z.sort(a, max)

// copy a[], flag into struct
// send message with struct
Many Local Call Assumptions don’t Hold!

- Not a local object Invocation, so need more help
  - Need remote equivalent of local (assembler) calling conventions
    - In this class we will come to understand this “plumbing” much better
- Not in same programming language (can’t assume)
- Not written by same programmer
- Not running same operating system for caller and callee
- Not same CPU type for caller and callee
- …
Many Local Call Assumptions don’t Hold! (2)

- Not always in the same administrative domain
- Latency for transfer of control and data can be large and, worse, unpredictable
- Partial failures
- Membership of the system (the computers in its collection) can change
- Unreliable or insecure communication
Context: (Most) Technology Marches On

- Hardware technology’s progress phenomenal in last few decades
  - Moore’s Law
  - Metcalf’s Law
  - Graphics processing power

- Software technology’s progress is much more spotty
  - “Software crisis”
  - Yet SW is a large and increasing part of complex apps/systems!

- Apps and systems are rapidly becoming (more) networked
  - Oops, distributed software is much harder yet to get right…

- Middleware a promising technology for programmability of distributed systems
Why Middleware?
• Middleware == “A layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system”
• Middleware exists to help manage the complexity and heterogeneity inherent in distributed systems
• Middleware provides higher-level building blocks (“abstractions”) for programmers than the OS provides
  ▪ Can make code much more portable
  ▪ Can make them much more productive
  ▪ Can make the resulting code have fewer errors
  ▪ Analogy — MW:sockets ≈ HOL:assembler
• Middleware sometimes is informally called “plumbing”
  ▪ Connects parts of a distributed application with “data pipes” and passes data between them
Middleware in Context

Distributed Application

Middleware API

Operating System API

Comm.  Processing  Storage

Network

Distributed Application

Middleware API

Operating System API

Comm.  Processing  Storage

Client

Server
Middleware Benefit: Masking Heterogeneity

• Middleware’s programming building blocks mask heterogeneity
  ▪ Makes programmer’s life much easier!!
• Kinds of heterogeneity masked by middleware (MW) frameworks
  ▪ All MW masks heterogeneity in network technology
  ▪ All MW masks heterogeneity in host CPU
  ▪ Almost all MW masks heterogeneity in operating system (or family thereof)
    – Notable exception: Microsoft middleware (de facto; not de jure or de fiat)
  ▪ Almost all MW masks heterogeneity in programming language
    – Noteable exception: Java RMI
  ▪ Some MW masks heterogeneity in vendor implementations
    – Object Management Group (omg.org) best here: CORBA (object-oriented), DDS (publish-subscribe)
Middleware Benefit: Transparency

- Middleware can provide useful transparencies:
  - Access Transparency
  - Location transparency
  - Concurrency transparency
  - Replication transparency
  - Failure transparency
  - Mobility transparency

- Masking heterogeneity and providing transparency makes programming distributed systems much easier to do!
Middleware Perspective

• “Middleware is like underwear: it is absolutely essential, but it should never be seen in public.” unknown witticist

• Background info (only first page required): http://www.eecs.wsu.edu/~bakken/middleware.pdf
Context: (Most) Technology Marches On

• Hardware technology’s progress phenomenal in last few decades
  ▪ Moore’s Law
  ▪ Metcalf’s Law
  ▪ Graphics processing power

• Software technology’s progress is much more spotty
  ▪ “Software crisis”
  ▪ Yet SW is a large and increasing part of complex apps/systems!

• Apps and systems are rapidly becoming (more) networked
  ▪ Oops, distributed software is much harder yet to get right…

• Middleware a promising technology for programmability of distributed systems
Why Middleware?

• Middleware == “A layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system”

• Middleware exists to help manage the complexity and heterogeneity inherent in distributed systems

• Middleware provides higher-level building blocks (“abstractions”) for programmers than the OS provides
  ▪ Can make code much more portable
  ▪ Can make them much more productive
  ▪ Can make the resulting code have fewer errors
  ▪ Analogy — MW:sockets ≈ HOL:assembler

• Middleware sometimes is informally called “plumbing”
  ▪ Connects parts of a distributed application with “data pipes” and passes data between them
Middleware in Context

Distributed Application

Middleware API

Operating System API

Comm. Processing Storage

Client

Distributed Application

Middleware API

Operating System API

Comm. Processing Storage

Server

Network
Middleware Benefit: Masking Heterogeneity

- Middleware’s programming building blocks mask heterogeneity
  - Makes programmer’s life much easier!!
- Kinds of heterogeneity masked by middleware (MW) frameworks
  - All MW masks heterogeneity in network technology
  - All MW masks heterogeneity in host CPU
  - Almost all MW masks heterogeneity in operating system (or family thereof)
    - Notable exception: Microsoft middleware (*de facto*; not *de jure* or *de fiat*)
  - Almost all MW masks heterogeneity in programming language
    - Notable exception: Java RMI
  - Some MW masks heterogeneity in vendor implementations
    - Object Management Group (omg.org) best here: CORBA (object-oriented), DDS (publish-subscribe)
Middleware Benefit: Transparency

• Middleware can provide useful transparencies:
  ▪ Access Transparency
  ▪ Location transparency
  ▪ Concurrency transparency
  ▪ Replication transparency
  ▪ Failure transparency
  ▪ Mobility transparency

• Masking heterogeneity and providing transparency makes programming distributed systems much easier to do!
Middleware and Legacy Systems

- Legacy systems are a huge problem (and asset) in industry and military domains!

- Middleware often called a “glue” technology: integrated “legacy” components
  - Much distributed programming involves integrating components, not building them from scratch!

- Middleware’s abstractions are general enough to allow legacy systems to be “wrapped”
  - Distributed objects are best here because more general
  - End result: a very high-level “lowest common denominator” of interoperability
Middleware vs. Sockets

- Middleware is much easier to program!
- Example interface from CORBA (OMG) IDL:

```idl
module HelloApp {
    interface Hello {
        bool MyFunction(in float a, in string b, in int c, in string d, in float e, out double ret);
    }
};
```
Middleware vs. Sockets(2)

- Calling that interface in C++ with CORBA

- `helloImpl (classname) = CORBA_NAME(…..)

  ```c++
  boolean success = helloImpl.MyFunction(3.3, “hello”, 2345, “bakken!”, 67.34, doubleBox);
  ```
float a; char b[5]; int c; char d[7]; float e; double rval; int success

//Ignore read errors. Hardcode field size, assume all systems are same CPU arch. and bit size
read(socket, &a, sizeof(float));
read(socket, b, sizeof(char) * 5);
read(socket, &c, sizeof(int));
read(socket, d, sizeof(char)*7);
read(socket, &e, sizeof(float));

// ... continued on next slide ...
Middleware vs. Sockets (4)

// ... continued from previous slide ...

... calculating return values etc goes here ...

// send back return value
write(socket, &rval, sizeof(double));

// cant tell if it actually was received, or if socket is broken
write(socket, &success, sizeof(int));

// again, no error checking
Middleware vs. Sockets (5)

• This socket code ignored all of the following:
  • Errors with the socket
  • Differences in CPU architecture (endianness)
  • Differences in representation of data types between languages
  • I/O errors
  • Type checking of data variables

• All of the above (and much more) are handled by middleware

• Middleware’s programming building blocks (abstractions) mask heterogeneity
  • Makes programmer’s life much easier!!