ECE 587 – Hardware/Software Co-Design
Lecture 16 Case Study II

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

March 9, 2015
Midterm Exam

- Wed. Mar. 11, 3:15 PM – 4:30 PM
- Closed book/notes, cheat sheet allowed
- We will use this classroom.
- Students registered for TV/Online/India session
  - You may take it with a proctor or with me here on-campus.
Due Date: 10:00am 05/01 Chicago time
Reading Assignment

- This lecture: Notes
- Next lecture (after Midterm exam): 5
RPC and RMI

MapReduce
Remote Procedure Call (RPC)

- A paradigm for distributed computing.
  - Allow to access resources on other machines as if they are local.
- Specification model: sequential programs
  - Two types of processes: client and server. Processes
  - Implicit communication: clients “call” servers.
- A very popular choice.
  - Communication is completely hidden at the application layer – tools will handle the details automatically.
  - RPC can be built on top of many transport layer mechanisms.
  - Many different protocols and implementations.
Remote Procedure Call (RPC)

- A paradigm for distributed computing.
  - Allow to access resources on other machines as if they are local.

- Specification model: sequential programs
  - Two types of processes: client and server. Processes
  - Implicit communication: clients “call” servers.

- A very popular choice.
  - Communication is completely hidden at the application layer – tools will handle the details automatically.
  - RPC can be built on top of many transport layer mechanisms.
  - Many different protocols and implementations.
Remote Procedure Call (RPC)

- A paradigm for distributed computing.
  - Allow to access resources on other machines as if they are local.
- Specification model: sequential programs
  - Two types of processes: client and server. Processes
  - Implicit communication: clients “call” servers.
- A very popular choice.
  - Communication is completely hidden at the application layer – tools will handle the details automatically.
  - RPC can be built on top of many transport layer mechanisms.
  - Many different protocols and implementations.
Many RPC implementations require the use of an interface description language (IDL) to specify the communication as a function call.

```plaintext
service adder {
    i32 add(1:i32 num1, 2:i32 num2)
}
```

An IDL compiler will then generate a client stub and a server stub to hide all details of communication.

- The stubs take care of implementations at the presentation layer and below.

It is possible to

- Generate stubs for multiple languages.
- Utilize different presentation layer protocols.
- Choose different transport layer implementations.
Many RPC implementations require the use of an interface description language (IDL) to specify the communication as a function call.

```plaintext
service adder {
    i32 add(1:i32 num1, 2:i32 num2)
}
```

An IDL compiler will then generate a client stub and a server stub to hide all details of communication.

- The stubs take care of implementations at the presentation layer and below.

It is possible to

- Generate stubs for multiple languages.
- Utilize different presentation layer protocols.
- Choose different transport layer implementations.
Many RPC implementations require the use of an interface description language (IDL) to specify the communication as a function call.

```
service adder {
    i32 add(1:i32 num1, 2:i32 num2)
}
```

An IDL compiler will then generate a client stub and a server stub to hide all details of communication.

- The stubs take care of implementations at the presentation layer and below.

It is possible to
- Generate stubs for multiple languages.
- Utilize different presentation layer protocols.
- Choose different transport layer implementations.
void rpc_client() {
    // communication configuration
    shared_ptr<TTransport> socket(new TSocket("localhost", 9090));
    ...
    shared_ptr<TProtocol> protocol(...);

    // adderClient is implemented in the client stub
    adderClient client(protocol);

    try {
        socket->open();

        // the computation actually happen remotely in the server
        cout << "1 + 1 = " << client.add(1, 1) << endl;

        socket->close();
    }
    catch (TException& tx) {
        // communication/server may fail
        cout << "ERROR: " << tx.what() << endl;
    }
}
// the interface adderIf is defined in the server stub
class adderHandler : public adderIf {
public:
    int32_t add(const int32_t n1, const int32_t n2) {
        cout << "add(" << n1 << ", " << n2 << ")" << endl;
        return n1 + n2;
    }
};

int main() {
    // communication configuration
    shared_ptr<adderHandler> handler(new adderHandler());
    shared_ptr<TProcessor> processor(new CalculatorProcessor(handler));
    shared_ptr<TServerTransport> serverTransport(new TServerSocket(9090));
    ...

    // The RPC library may provide additional support for server implementation.
    TSimpleServer server(processor, serverTransport, ...);
    server.serve();
    return 0;
}
A RPC call that returns something requires a pair of messages to be exchanged between the client and the server.

Sequence of events

1. The client calls the client stub locally.
2. The client stub encodes the parameters into a message and sends it to the server stub via a transport. It then waits for the server stub to reply a message that encodes the result.
3. Upon receiving a message, the server stub decodes it into parameters and calls the server implementation.
4. The server stub encodes what the server implementation returns into a message and replies it to the client stub.
5. The client stub decodes the message into the result and returns it to the client.
A RPC call that returns something requires a pair of messages to be exchanged between the client and the server.

Sequence of events

1. The client calls the client stub locally.
2. The client stub encodes the parameters into a message and sends it to the server stub via a transport. It then waits for the server stub to reply a message that encodes the result.
3. Upon receiving a message, the server stub decodes it into parameters and calls the server implementation.
4. The server stub encodes what the server implementation returns into a message and replies it to the client stub.
5. The client stub decodes the message into the result and returns it to the client.
Remote Method Invocation (RMI)

- **“Object-Oriented” RPC**
  - Mostly known as part of Java.
  - Remote resources (state bits) are grouped into objects and the functions to access/modify them become methods.
  - The clients may utilize OO to organize their programs.
  - The servers may utilize the knowledge of objects to optimize for concurrent calls.
Remote Method Invocation (RMI)

- “Object-Oriented” RPC
  - Mostly known as part of Java.
- Remote resources (state bits) are grouped into objects and the functions to access/modify them become methods.
  - The clients may utilize OO to organize their programs.
  - The servers may utilize the knowledge of objects to optimize for concurrent calls.
Remote Method Invocation (RMI)

▶ “Object-Oriented” RPC
  ▶ Mostly known as part of Java.
▶ Remote resources (state bits) are grouped into objects and the functions to access/modify them become methods.
▶ The clients may utilize OO to organize their programs.
▶ The servers may utilize the knowledge of objects to optimize for concurrent calls.
Remote Method Invocation (RMI)

- “Object-Oriented” RPC
  - Mostly known as part of Java.
- Remote resources (state bits) are grouped into objects and the functions to access/modify them become methods.
- The clients may utilize OO to organize their programs.
- The servers may utilize the knowledge of objects to optimize for concurrent calls.
RPC and RMI are intuitive extensions to sequential programs that enable distributed computing.

The overhead of the round-trip message exchange delay per RPC/RMI call cannot be ignored when many calls should be made to complete a task.

- Performance and scalability are concerns.

There are ways to hide the delay, e.g. by overlapping multiple calls.

- However, that usually requires a multi-threaded approach, which gives up the simplicity of sequential programs.

Failures are exposed to the clients and it is up to the clients to handle them explicitly.

- Abort or retry?
Discussions

- RPC and RMI are intuitive extensions to sequential programs that enable distributed computing.
- The overhead of the round-trip message exchange delay per RPC/RMI call cannot be ignored when many calls should be made to complete a task.
  - Performance and scalability are concerns.
- There are ways to hide the delay, e.g. by overlapping multiple calls.
  - However, that usually requires a multi-threaded approach, which gives up the simplicity of sequential programs.
- Failures are exposed to the clients and it is up to the clients to handle them explicitly.
  - Abort or retry?
Discussions

- RPC and RMI are intuitive extensions to sequential programs that enable distributed computing.
- The overhead of the round-trip message exchange delay per RPC/RMI call cannot be ignored when many calls should be made to complete a task.
  - Performance and scalability are concerns.
- There are ways to hide the delay, e.g. by overlapping multiple calls.
  - However, that usually requires a multi-threaded approach, which gives up the simplicity of sequential programs.
- Failures are exposed to the clients and it is up to the clients to handle them explicitly.
  - Abort or retry?
Discussions

- RPC and RMI are intuitive extensions to sequential programs that enable distributed computing.
- The overhead of the round-trip message exchange delay per RPC/RMI call cannot be ignored when many calls should be made to complete a task.
  - Performance and scalability are concerns.
- There are ways to hide the delay, e.g. by overlapping multiple calls.
  - However, that usually requires a multi-threaded approach, which gives up the simplicity of sequential programs.
- Failures are exposed to the clients and it is up to the clients to handle them explicitly.
  - Abort or retry?
Outline

RPC and RMI

MapReduce
MapReduce

- A model to specify parallel algorithms.
  - That can be readily mapped to a cluster by implementations, e.g. Hadoop.
- A few types of processes.
  - Specified as actors to factor out communications.
  - Pre-defined data dependencies. Typically input → map → combine → reduce → output.
  - Usually only the map and the reduce actors are specified by the user.
- Implicit communication: inputs/outputs of actors.
  - As (key,value) pairs where key indicates the destination and value is the payload.
- Unlike SDF where actors and communication channels are fixed, actors and channels for MapReduce are generated at runtime depending on the actual data.
A model to specify parallel algorithms.
- That can be readily mapped to a cluster by implementations, e.g. Hadoop.

A few types of processes.
- Specified as actors to factor out communications.
- Pre-defined data dependencies. Typically input → map → combine → reduce → output.
- Usually only the map and the reduce actors are specified by the user.

Implicit communication: inputs/outputs of actors.
- As (key, value) pairs where key indicates the destination and value is the payload.

Unlike SDF where actors and communication channels are fixed, actors and channels for MapReduce are generated at runtime depending on the actual data.
MapReduce

- A model to specify parallel algorithms.
  - That can be readily mapped to a cluster by implementations, e.g. Hadoop.

- A few types of processes.
  - Specified as actors to factor out communications.
  - Pre-defined data dependencies. Typically input $\rightarrow$ map $\rightarrow$ combine $\rightarrow$ reduce $\rightarrow$ output.
  - Usually only the map and the reduce actors are specified by the user.

- Implicit communication: inputs/outputs of actors.
  - As (key,value) pairs where key indicates the destination and value is the payload.

- Unlike SDF where actors and communication channels are fixed, actors and channels for MapReduce are generated at runtime depending on the actual data.
MapReduce

- A model to specify parallel algorithms.
  - That can be readily mapped to a cluster by implementations, e.g. Hadoop.
- A few types of processes.
  - Specified as actors to factor out communications.
  - Pre-defined data dependencies. Typically input → map → combine → reduce → output.
  - Usually only the map and the reduce actors are specified by the user.
- Implicit communication: inputs/outputs of actors.
  - As (key,value) pairs where key indicates the destination and value is the payload.
- Unlike SDF where actors and communication channels are fixed, actors and channels for MapReduce are generated at runtime depending on the actual data.
class Map_WordCount extends ... {
    public void map(
        LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {

        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            output.collect(
                new Text(tokenizer.nextToken()),
                new IntWritable(1));
        }
    }
}

A map actor consumes what an input actor generates and outputs pairs to combine actors.
class Reduce_WordCount extends ... {
    public void reduce(
        Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

- Combine actors group output pairs from map actors by the key.
- Reduce actors take what combine actors output and generate pairs for output actors as results.
class Reduce_WordCount extends ... {
    public void reduce(
        Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

- Combine actors group output pairs from map actors by the key.
- Reduce actors take what combine actors output and generate pairs for output actors as results.
Discussions

- Simplify design and implementation of many parallel algorithms.
- For MapReduce implementations, pre-defined data dependencies makes it easy to identify parallelism among actors and to schedule them accordingly.
  - Allow to use computational resources from many computers.
  - Can we allow more flexible data dependencies to support a larger set of algorithms?
- Stateless actors can be re-executed if failure happens.
  - Allow MapReduce implementations to be failure resilient.
- The system states are encoded in the (key,value) pairs.
  - Current MapReduce implementations store them on disks – could be a performance bottleneck.
  - Can we only store a portion of the pairs and recalculate the rest when failure happens for better performance?
Discussions

- Simplify design and implementation of many parallel algorithms.
- For MapReduce implementations, pre-defined data dependencies makes it easy to identify parallelism among actors and to schedule them accordingly.
  - Allow to use computational resources from many computers.
  - Can we allow more flexible data dependencies to support a larger set of algorithms?
- Stateless actors can be re-executed if failure happens.
  - Allow MapReduce implementations to be failure resilient.
- The system states are encoded in the (key,value) pairs.
  - Current MapReduce implementations store them on disks – could be a performance bottleneck.
  - Can we only store a portion of the pairs and recalculate the rest when failure happens for better performance?
Discussions

- Simplify design and implementation of many parallel algorithms.
- For MapReduce implementations, pre-defined data dependencies makes it easy to identify parallelism among actors and to schedule them accordingly.
  - Allow to use computational resources from many computers.
  - Can we allow more flexible data dependencies to support a larger set of algorithms?
- Stateless actors can be re-executed if failure happens.
  - Allow MapReduce implementations to be failure resilient.
- The system states are encoded in the (key,value) pairs.
  - Current MapReduce implementations store them on disks – could be a performance bottleneck.
  - Can we only store a portion of the pairs and recalculate the rest when failure happens for better performance?
Discussions

- Simplify design and implementation of many parallel algorithms.
- For MapReduce implementations, pre-defined data dependencies makes it easy to identify parallelism among actors and to schedule them accordingly.
  - Allow to use computational resources from many computers.
  - Can we allow more flexible data dependencies to support a larger set of algorithms?
- Stateless actors can be re-executed if failure happens.
  - Allow MapReduce implementations to be failure resilient.
- The system states are encoded in the (key,value) pairs.
  - Current MapReduce implementations store them on disks – could be a performance bottleneck.
  - Can we only store a portion of the pairs and recalculate the rest when failure happens for better performance?