Principles of Software Construction: Objects, Design, and Concurrency
Part 6: Concurrency and distributed systems

Abstractions of State

Christian Kästner  Charlie Garrod
Administrivia

• Homework 6...
• Final exam Tuesday, May 5\textsuperscript{th}, 1 – 4 p.m.  DH 2210
  – Final exam review session Sunday, May 3\textsuperscript{rd}, 4 – 6:30 p.m., Hamburg 1000
Key concepts from Tuesday
Data consistency

• Suppose \( D \) is the database for some application and \( \varphi \) is a function from database states to \{true, false\}
  – We call \( \varphi \) an integrity constraint for the application if \( \varphi(D) \) is true if the state \( D \) is "good"
  – We say a database state \( D \) is consistent if \( \varphi(D) \) is true for all integrity constraints \( \varphi \)
  – We say \( D \) is inconsistent if \( \varphi(D) \) is false for any integrity constraint \( \varphi \)

• Transaction ACID properties:
  – Atomicity: All or nothing
  – Consistency: Application-dependent as before
  – Isolation: Each transaction runs as if alone
  – Durability: Database will not abort or undo work of a transaction after it confirms the commit
Concurrent transactions and serializability

• For good performance, database interleaves operations of concurrent transactions

• Problems to avoid:
  – Lost updates
    • Another transaction overwrites your update, based on old data
  – Inconsistent retrievals
    • Reading partial writes by another transaction
    • Reading writes by another transaction that subsequently aborts

• A schedule of transaction operations is * Serializable* if it is equivalent to some serial ordering of the transactions
2PC sequence of events for a successful commit

Coordinator:

“prepared”

“committed” (persistently)

“done”

Participants:

“prepared” (persistently)

“uncertain” (objects still locked)

“committed”
Problems with two-phase commit?
Problems with two-phase commit?

- Failure assumptions are too strong
  - Real servers can fail permanently
  - Persistent storage can fail permanently
- Temporary failures can arbitrarily delay a commit
- Poor performance
  - Many round-trip messages
Aside: The CAP theorem for distributed systems

• For any distributed system you want...
  – Consistency
  – Availability
  – tolerance of network Partitions

• ...but you can support at most two of the three
Today: Abstractions of state

- State-based models of computation
  - Finite state machines (FSMs)
- The State design pattern
- A distributed application: The actor model
An aside: I need two volunteers...
Memorize the following number:

4 2
What was the number?
Memorize the following number:

4 2 9 7
What was the number?
Memorize the following number:

4  2  9  7  2  8
What was the number?
Memorize the following number:

4 2 9 7 2 8 6 1 9 3
9 1 0 2 8 4 0 0 2 8
8 2 1 0 8 2 7 3 2 3
3 3 2 8 6 6 7 1 0 0
8 0 9 1 0 8 2 8 6 4
2 8 5 6 0 9 1 7 2 8
2 7 8 1 6 8 7 2 0 9
Memorize the following number:

<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
<th>4</th>
<th>2</th>
<th>4</th>
<th>2</th>
<th>4</th>
<th>2</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
An aside's aside: Run-length encodings

4 2 (5*7 times)
What causes programming errors?
What causes programming errors?

- Knowledge problems: Inadequate, inert, heuristic, oversimplified, or interfering content or organization
- Attentional problems: Fixation, loss of situational awareness, or working memory strain
- Strategic problems: Unforeseen interactions from goal conflict resolution or bounded rationality

What causes programming errors?

- Knowledge problems: Inadequate, inert, heuristic, oversimplified, or interfering content or organization
- Attentional problems: Fixation, loss of situational awareness, or working memory strain
- Strategic problems: Unforeseen interactions from goal conflict resolution or bounded rationality

A goal: Eliminate complexity

Today: Abstractions of state

• State-based models of computation
  – Finite state machines (FSMs)
• The State design pattern
• A distributed application: The actor model
Related: Deterministic Finite Automata (DFAs)

- A simple model of computation in which input is accepted or rejected by a finite state machine
  - e.g. A DFA that accepts the input 42:
Related: Deterministic Finite Automata (DFAs)

- A simple model of computation in which input is accepted or rejected by a finite state machine
  - e.g. A DFA that accepts the input 42:
Related: Deterministic Finite Automata (DFAs)

- A simple model of computation in which input is accepted or rejected by a finite state machine
  - e.g. A DFA that accepts the input 42:
Related: Deterministic Finite Automata (DFAs)

- A simple model of computation in which input is accepted or rejected by a finite state machine
  - e.g. A DFA that accepts the input 42:

```
0-3,5-9
```

```
0-9
```

```
0-9
```

```
0,1,3-9
```

```
2
```

```
4
```

```
ε
```
Related: Turing Machines

- A simple model of computation in which input is accepted or rejected by a finite state machine
  - Essentially a DFA with an infinite memory tape
Finite state machines (FSMs)
FSMs simply represent system behavior

- E.g., a 4-function calculator
FSMs simply represent system behavior

• E.g., a 4-function calculator
• E.g., the traffic light at Forbes and Morewood
FSMs enable precise communication

- E.g., the Transmission Control Protocol (TCP)
UML state diagrams enable richer communication

- Conditional transitions
- Independent events/actions

(example from Wikipedia...)
FSMs can help organize the implementation

• See StateMachineCalculator.java
FSMs can help organize the implementation

• See StateMachineCalculator.java
  – Warning: The StateMachineCalculator intentionally demonstrates poor design.
A calculator with the State design pattern

- See StatePatternCalculator.java
The State design pattern

- **Applicability:**
  - An object's behavior depends on its state, and it must change its behavior at run-time based on that state.
  - Transition function between states is highly state-dependent and complex.

- **Consequences:**
  - State-specific behavior is partitioned, localized, and cohesive.
  - State transitions are explicit.
  - State objects can be shared.
Recall a problem of concurrency: Shared state
MapReduce's approach to shared state

- E.g., for each word on the Web, count the number of times that word occurs
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is a word, values is a list of the number of counts of that word

```java
f1(String key1, String value):
    for each word w in value:
        EmitIntermediate(w, 1);

f2(String key2, Iterator values):
    int result = 0;
    for each v in values:
        result += v;
    Emit(key2, result);
```

Map: (key1, v1) → (key2, v2*)
Reduce: (key2, v2*) → (key3, v3*)

MapReduce: (docName, docText)* → (word, wordCount)*

Transactional approach to shared state

• For good performance, database interleaves operations of concurrent transactions

• Problems to avoid:
  – Lost updates
    • Another transaction overwrites your update, based on old data
  – Inconsistent retrievals
    • Reading partial writes by another transaction
    • Reading writes by another transaction that subsequently aborts

• A schedule of transaction operations is *serializable* if it is equivalent to some serial ordering of the transactions
Models of concurrency and parallelism

- Explicit concurrency: threads and locking
- Functional programming
- Transactions and serializability
- MapReduce and other data-centric architectures
- SIMD and data parallelism
- Communicating sequential processes
  - Message passing
  - Channels
  - The actor model
The actor model

- System is composed of independent *actors* that communicate via asynchronous messages

  i.e. concurrent function calls without return values

  sequential, no shared state
The actor model

- System is composed of independent *actors* that communicate via asynchronous messages

- Properties of actors:
  - Sequential and non-blocking
  - Non-shared, mutable state
  - Queue for incoming messages
  - Inherently concurrent
  - Extremely lightweight
  - Distributed by default

```
while true:
    process next message
```
Implementations of the actor model

• Frameworks:
  – Java: Akka
  – Python: Pykka
  – C++: CAF (C++ Actor Framework)

• Languages:
  – Scala
  – Scratch
  – Erlang
  – Elixer

• Typically provide:
  – Communication between actors
  – Distribution among servers
  – Supervisory relationships between actors
  – Lightweight management and scheduling
Processing messages

• An actor may:
  – Change its internal state
  – Send one or more messages to other actors
  – Create one or more new actors
Processing messages

• An actor may:
  – Change its internal state
  – Send one or more messages to other actors
  – Create one or more new actors

  • Defines a hierarchy of actors

(source: *Seven Concurrency Models in Seven Weeks* by Paul Butcher.)
Recall an advantage of Exceptions

- Separates normal and exceptional control flow

```java
try {
    FileInputStream fileInput = new FileInputStream(filename);
    DataInputStream dataInput = new DataInputStream(fileInput);
    int i = dataInput.readInt();
    fileInput.close();
    return i;
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + filename);
    return -1;
} catch (IOException e) {
    System.out.println("Error reading binary data from file " + filename);
    return -1;
}
```
Error handling in the actor model

• "Let it crash"
  – Resume or restart failed actors
  – Escalate errors to higher level

(source: Seven Concurrency Models in Seven Weeks by Paul Butcher.)
Trade-offs of the actor model

• Strengths:
  – Strong encapsulation via isolation and messaging
  – Fault tolerance
  – Inherently distributed and concurrent

• Weaknesses:
  – Messages expensive compared to shared, local memory
  – Subtle systemic problems, e.g. overflowing mailboxes
Next time...

- Version control systems