Principles of Software Construction

Serializability and Transactions

Josh Bloch **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 6 checkpoint due tomorrow 5 p.m.
- Final exam Tuesday, May 3rd 5:30-8:30 p.m., PH 100
  - Review session Sunday, May 1st 7-9 p.m., DH 1112

Tuesday, reprise

- (See Tuesday's slides for details)

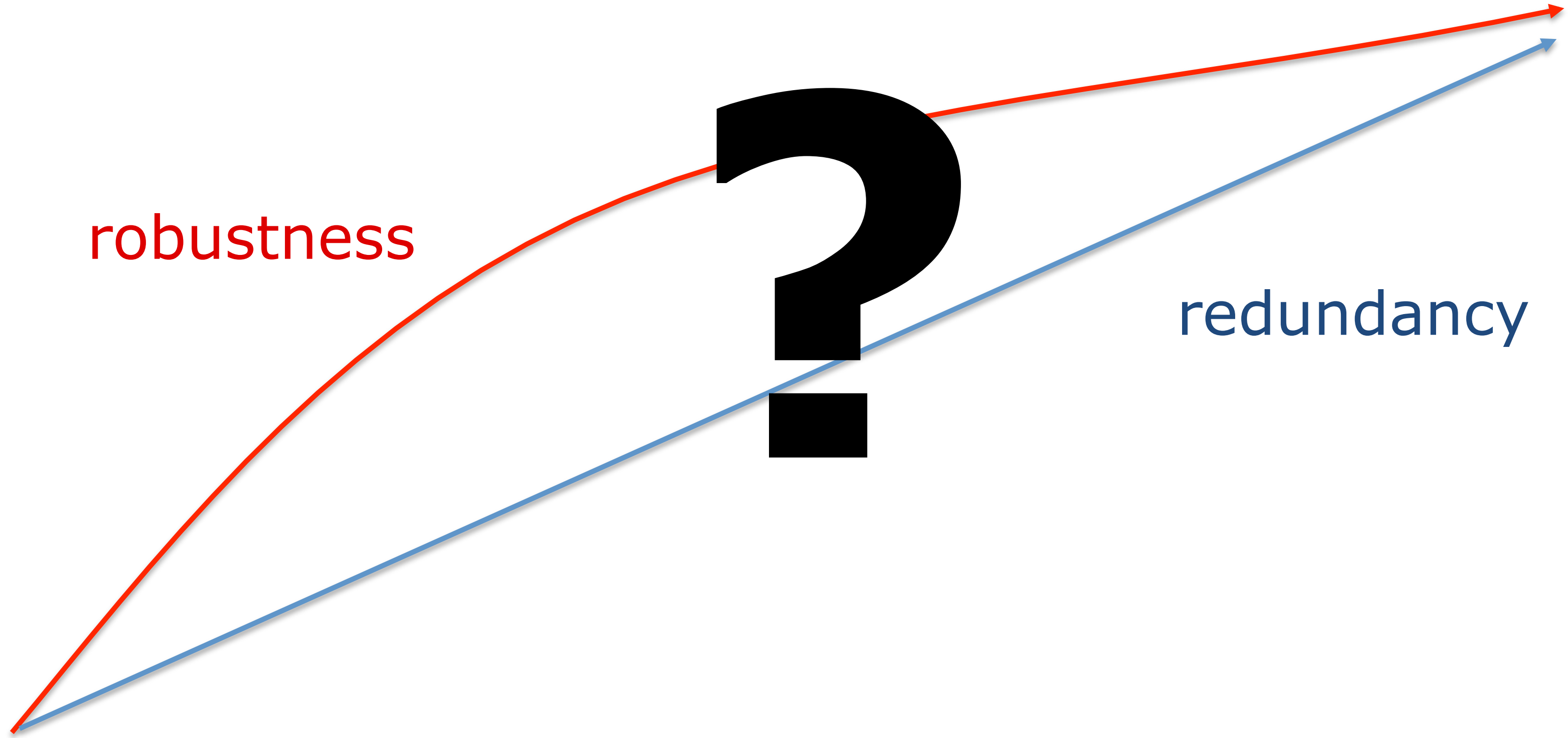
Today: Transactions and serializability

- Gang of Four design patterns, reprised
- Aside: Distributed systems principles
- A formal definition of consistency
- Introduction to transactions
- Concurrency control and serializability
- Distributed concurrency control (time permitting)
  - Two-phase commit

Some distributed system design goals

- The end-to-end principle
  - When possible, implement functionality at the end components (rather than middle components) of a distributed system
- The robustness principle
  - Be strict in what you send, but be liberal in what you accept from others
    - Protocols
    - Failure behaviors
- Benefit from incremental changes
- Be redundant
  - Data replication
  - Checks for correctness

Aside: The robustness vs. redundancy curve



institute for SOFTWARE RESEARCH

Today: Transactions and serializability

- Gang of Four design patterns, reprised
- Aside: Distributed systems principles
- A formal definition of consistency
- Introduction to transactions
- Concurrency control and serializability
- Distributed concurrency control (time permitting)
  - Two-phase commit

An aside: Double-entry bookkeeping

- A style of accounting where every event consists of two separate entries: a credit and a debit

```
void transfer(Account fromAcct, Account toAcct, int val) {
    fromAccount.debit(val);
    toAccount.credit(val);
}

static final Account BANK_LIABILITIES = …;

void deposit(Account toAcct, int val) {
    transfer(BANK_LIABILITIES, toAcct, val);
}

boolean withdraw(Account fromAcct, int val) {
    if (fromAcct.getBalance() < val) return false;
    transfer(fromAcct, BANK_LIABILITIES, val);
    return true;
}
```

Some properties of double-entry bookkeeping

- Redundancy!
- Sum of all accounts is static
  - Can be 0

Data consistency of an application

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

Data consistency of an application

- Suppose $\mathcal{D}$ is the database for some application and φ is a function from database states to {true, false}
  - We call φ an *integrity constraint* for the application if $\varphi(\mathcal{D})$ is true if the state $\mathcal{D}$ is "good"
  - We say a database state $\mathcal{D}$ is *consistent* if $\varphi(\mathcal{D})$ is true for all integrity constraints φ
  - We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint φ
- E.g., for a bank using double-entry bookkeeping one possible integrity constraint is:

```
def IsConsistent(D):
    If sum(all account balances in D) == 0:
        Return True
    Else:
        Return False
```

Database transactions

- A *transaction* is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another
  - "*Atomic*" ~ indivisible
- Transactions always terminate with either:
  - *Commit*: complete transaction's changes successfully
  - *Abort*: undo any partial work of the transaction

Database transactions

- A *transaction* is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another
  - "*Atomic*" ~ indivisible
- Transactions always terminate with either:
  - *Commit*: complete transaction's changes successfully
  - *Abort*: undo any partial work of the transaction

```
boolean withdraw(Account fromAcct, int val) {
    begin_transaction();
    if (fromAcct.getBalance() < val) {
        abort_transaction();
        return false;
    }
    transfer(fromAcct, BANK_LIABILITIES, val);
    commit_transaction();
    return true;
}
```

A functional view of transactions

- A transaction $\mathcal{T}$ is a function that takes the database from one state $\mathcal{D}$ to another state $\mathcal{T}(\mathcal{D})$
- In a correct application, if $\mathcal{D}$ is consistent then $\mathcal{T}(\mathcal{D})$ is consistent for all transactions $\mathcal{T}$

A functional view of transactions

- A transaction $\mathcal{T}$ is a function that takes the database from one state $\mathcal{D}$ to another state $\mathcal{T}(\mathcal{D})$
- In a correct application, if $\mathcal{D}$ is consistent then $\mathcal{T}(\mathcal{D})$ is consistent for all transactions $\mathcal{T}$
  - E.g., in a correct application any serial execution of multiple transactions takes the database from one consistent state to another consistent state

Database transactions in practice

- The application requests commit or abort, but the database may arbitrarily abort any transaction
  - Application can restart an aborted transaction
- 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

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