Programming Distributed Systems

12 Programming Models for Distributed Systems

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019
What is a Programming Model? [4]

- A programming model is some form of abstract machine
  - Provides operations to the level above
  - Requires implementations for these operations on the level(s) below
- Simplification through abstraction
- Standard interface that remains stable even if underlying architecture changes
- Provide different levels of abstraction
- Often starting point for language development

⇒ Separation of concern between software developers and framework implementors (runtime system, compiler, etc.)
Properties of good programming models

- Meaningful abstractions
- System-architecture independent
- Efficiently implementable
- Easy to understand
What kind of abstractions should a programming model for distributed systems provide?
Remote Procedure Call
Remote Procedure Call (RPC) [2]

- Rather broad classifying term with changing meaning over time
  - From client-server design to interconnected services
- Two entities (caller/callee) with different address spaces communicate over some channel in a request-response mechanism
- Examples: CORBA (Common Object Request Broker Architecture), Java RMI (Remote Method Invocation), SOAP (Simple Object Access Protocol), gRPC (Protocol Buffers), Twitter Finagle ...
Flaws of RPC

- Location transparency (i.e. request to remote service looks like local function call) masks the potential of distribution-related failures.
- RPCs might timeout, requires usually special handling such as retrying.
- Local functions do not need to deal with the problem of idempotence.
- Execution time is unpredictable.
- Passing of objects is complex (e.g. might need to serialize referenced objects).
- Translating data types between languages might rely on semantical approximation.
Aspects of modern RPC

- Language-agnostic
- Serialization (aka marshalling or pickling)
  - JSON, XML, Protocol Buffers, ...
- Load-balancing
  - SOA (Service-oriented architecture) ⇒ Microservice architectures!
- Asynchronous

⇒ RPC as term gets more and more diffuse
Futures and Promises

- “Asynchronous RPC”
- A future is a value that will eventually become available
- Two states:
  - completed: value is available
  - incomplete: computation for value is not yet complete
- Strategies: Eager vs. lazy evaluation
- Typical application: Web development and user interfaces
Example

interface ArchiveSearcher { String search(String target); }

class App {
    ExecutorService executor = ...;
    ArchiveSearcher searcher = ...;
    void showSearch(final String target)
        throws InterruptedException {
        Future<String> future
            = executor.submit(new Callable<String>(){
                public String call() {
                    return searcher.search(target);
                }
            });
        displayOtherThings(); // do other things while searching
        try {
            displayText(future.get()); // use future
        } catch (ExecutionException ex) { cleanup(); return; }
    }
}

From Oracle’s Java Documentation
Actors and Message Passing
Characteristics of Actor Model [3]

- Actors are isolated units of computation + state that can send messages asynchronously to each other.
- Messages are queued in mailbox and processed sequentially when they match against some pattern/rule.
- No assumptions on message delivery guarantees.
- (Potential) State + behavior changes upon message processing[1].
- Very close to Alan Kay’s definition of Object-Oriented Programming.
Actors in the Wild

- **Erlang**
  - Process-based
  - Pure message passing
  - Monitor and link for notification of process failure/shutdown
  - OTP (Open Telecom Platform) for generic reusable patterns

- **Akka**
  - Actor model for the JVM
  - Purges non-matching messages
  - Enforces parental supervision
  - Included in Scala standard library

- **Orleans**
  - Actors for Cloud computing
  - Scalability by replication
  - Fine-grain reconciliation of state with transactions
Message brokers

- Message-oriented middleware which stores messages temporarily and forwards them to registered recipients
- Patterns: Publish-subscribe, point-to-point
- Acts as buffer for unavailable and overloaded recipients
- Decoupling of sender and receiver(s)
- Efficient 1-to-n multicast
- Advanced Message Queuing Protocol (AMQP) standardizes queuing, routing, reliability and security
- Delivery guarantees (at-most-once, at-least-once, exactly-once)
Example: RabbitMQ

- Supports (amongst others) publish-subscribe pattern
- Typical usage: Topics as routing keys

- Q1 is interested in all the orange animals
- Q2 wants to hear everything about rabbits, and everything about lazy animals
- Messages that don’t map any binding get lost
- Messages are maintained in the queue in publication order
Stream processing

- (Infinite) Sequence of data that is incrementally made available
- Example: Sensor data, audio / video delivery, filesystem APIs, etc.
- Producers vs. Consumers
- Notions of window and time: Consumers will receive only messages after subscribing
- Here: Event stream where data item is atypically associated with timestamp
Classification of stream processing systems

1. What happens if producer sends messages faster than the consumer can handle?
   - Drop messages
   - Buffer messages
   - Apply backpressure (i.e. prevent producer from sending more)

2. What happens if nodes become unreachable?
   - Loose messages
   - Use replication and persistence to preserve non-acknowledged messages
Log-based message brokers

- Example: Kafka [https://kafka.apache.org]
- Message buffers are typically transient: Once the message is delivered, the message is deleted
- Idea: Combine durable storage with low-latency notification!
Scalability and fault-tolerance for replicated logs

- For scalability, partitioning of log on different machines
- For fault-tolerance, replication on different machines

Anatomy of a Topic

- Need to ensure same ordering on all replicas (⇒ Total-order broadcast)
- Can easily add consumers for debugging, testing, etc.
- Ideas: Event-sourcing, immutability and audits
Batch-processing

- Static data sets that has known/finite size
- Need to artificially batch data into by day, month, minute, . . .
- Typically large latencies
The Future: Distributed Programming Languages
From Model to Language

- Challenges: Partial failure, concurrency and consistency, latency, ...

1. Distributed Shared Memory
   - Runtime maps virtual addresses to physical ones
   - “Single-system” illusion

2. Actors
   - Explicit communication
   - Location of processes is transparent

3. Dataflow
   - Data transformations expressed as DAG
   - Processes are transparent
   - Example: MapReduce (Google), Dryad (Microsoft), Spark
Example: WordCount in MapReduce
Further reading

- **Material collection** by Northeastern University, CS7680 Special Topics in Computing Systems: Programming Models for Distributed Computing
Further reading I


Further reading II