Relevant Technologies

Table of Contents

1 COMPUTER SCIENCE IN GENERAL

- R. L. Ashenhurst, editor. *ACM Turing Award Lectures: The First Twenty Years: 1966 to 1985*. ACM Press Anthology Series. ACM Press, 1987.
- C. A. R. Hoare et al. Grand Challenges in Computing.

2 ALGORITHMS AND THEORY OF COMPUTATION

2.1 General Textbooks

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.
- M. R. Garey and M. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

2.2 Complexity Theories

- E. Kushilevitz. Communication complexity. Manuscript, 1996.

2.3 Distributed Algorithms

- N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.
- H. Attiya. *Lecture Notes for Course #236357: Distributed Algorithms*. Department of Computer Science, The Technion, Haifa, January 1994.

- D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

3 PROGRAMMING LANGUAGES AND COMPILERS

3.1 General Textbooks

- H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.
- D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1992.
- J. C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, Cambridge, UK, 2003.
- A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.
- A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, UK, 1998.

3.2 Functional Programming Languages

- D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Proceedings of LFP' 84, ACM SIGPLAN Conference on LISP and Functional Programming*, pages 348–355, 1984.
- A. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of POPL' 89, 16th Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, 1989.
- A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- A. Appel. SSA is functional. *SIGPLAN Notices*, 1998.

3.2.1 Racket

3.3 Object-oriented Programming Languages

- M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, New York, NY, 1996.

3.4 Logic Programming Languages

- Warren abstract machine
- J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

3.5 Distributed Programming Languages

3.5.1 dRuby

3.5.2 Scala

- Scala ≡ "scalable language"
- functional JVM language (i.e. interpreter written with Java)
  - allows easy stealing of Java classes into Scala
  - *this becomes it's weakness – it's tied to Java*
- "tastefully-typed": statically-typed with type inference
- **mixin**: class which contains a combination of methods from other classes – "interface with implemented methods"
- **actor**:

```
actor {
  var sum = 0
  loop {
    // every actor has a  mailbox in
    // which incoming messages are queued
    receive {
      case Data(bytes)
        => sum += hash(bytes)
      // send = "recipient ! msg"
      case GetSum(requester)
        => requester ! sum
    }
  }
}
```

3.5.3 Erlang

- **basics**
  - world is concurrent: or *full of concurrent "processes"*
  - *message passing* is good: processes actually don't share data – shmem-based concurrency is difficult since maintaining data consistency is difficult esp. in presence of failures/delays
  - **COMMENT: message passing-based code is difficult to develop; how about letting users to shmem but we internally change it to message passing – i.e. PROTOCOL SYNTHESIS + COMMUNICATOR GENERATOR**
  - target of *message delivery*: **local process**, **remote process**
  - constructs for embracing **failures**
  - constructs for receptive code
- **three primitives for message-passing processes**
  - **spawn**: `Pid = spawn(Fun)`: create a process which executes function `Fun`
  - **send** (`Pid ! Message`): sends a message to the **mailbox** of process `Pid`
  - **receive**: remove a message from the **mailbox** of the process which executes this "receive"

```
receive
  Pattern1 [when Guard1] ->
    Expressions1;
  Pattern2 [when Guard2] ->
    Expressions2;
  ...
end
```

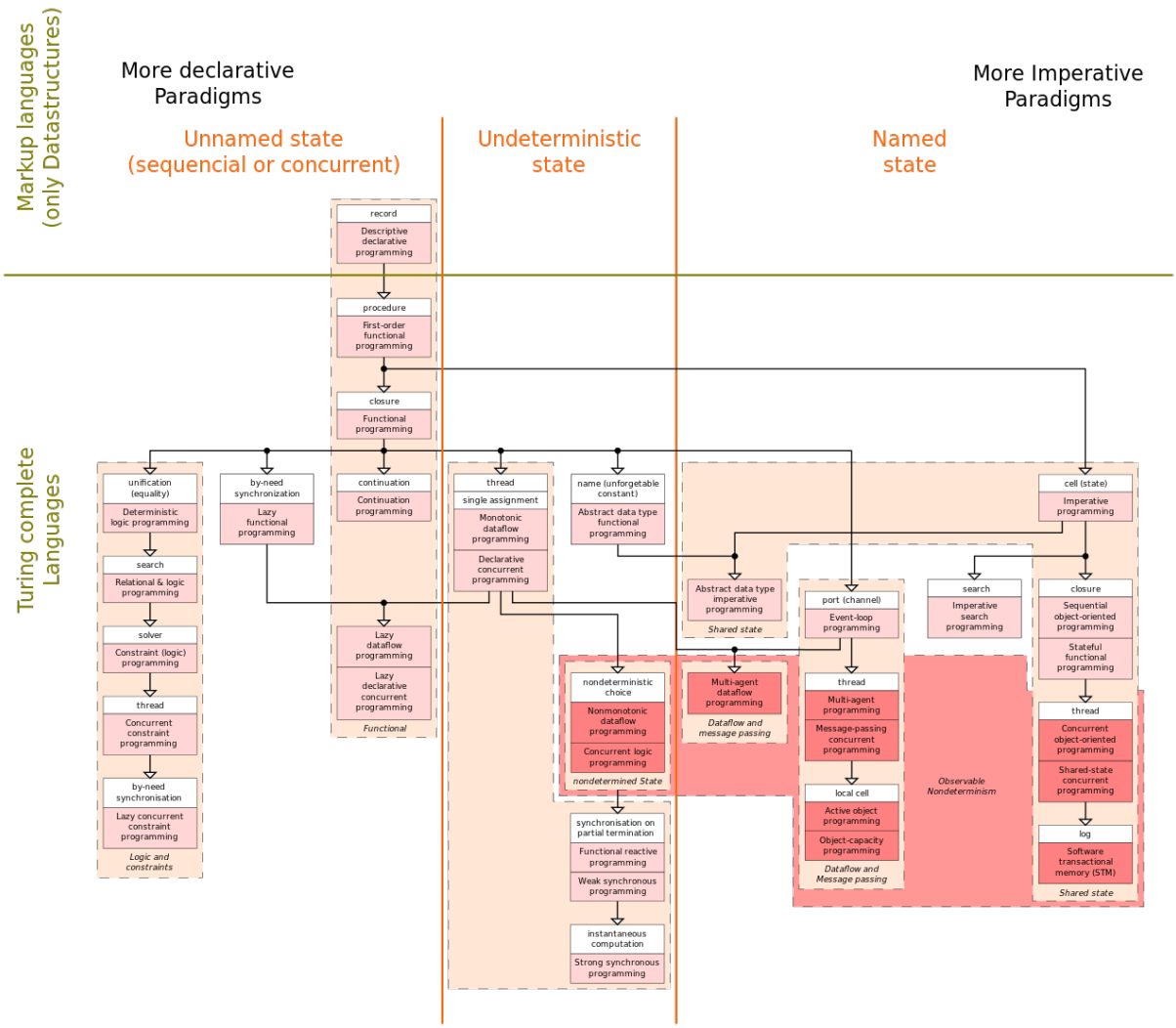
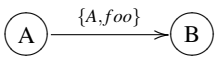


Figure 1: Programming Paradigms

- **message passing**
  - **simple message passing**

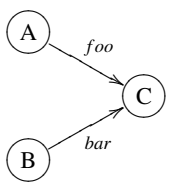
```
B!self(),foo
```




```
receive
 {From,Msg} -> Actions
end
```

  - **selective message reception**: pattern matched

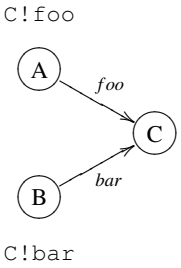
```
C!foo
```




```
C!bar
```

```
receive foo -> true end,
receive bar -> true end
```

- **selective message reception**: pattern matched




```
receive Msg
```

- **distributed Erlang**
  - **Erlang node**
  - **name server**:

3.5.4 Groovy

3.5.5 Ambit

3.5.6 Linda

3.6 Formal Semantics Of Programming Languages

- J. Stoy. *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
- S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume III, pages 1–168. Clarendon Press, 1991.

3.7 Practices in Programming Languages

- G. L. Steele, Jr. Growing a language. Talk given at OOPSLA'98, 1998.

4 FORMAL METHODS

4.1 Process Algebras

4.1.1 General textbooks

- H. Bowman and R. Gomez. *Concurrency Theory: Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*. Springer Verlag, 2006.

4.1.2 CSP (Communicating Sequential Processes)

- C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

4.1.3 CCS (Calculus of Communicating Systems)

- R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer Verlag, 1980.
- M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
- R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- R. Milner. Semantics of concurrent processes. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 1201–1242. MIT Press, 1990.

4.1.4 π-calculus

- R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, Cambridge, UK, 1999.
- R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.
- R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, September 1992.
- R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part II. *Information and Computation*, 100(1):41–77, September 1992.
- R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
- B. Pierce. The Pict Programming Language.

4.2 Petri Nets

- T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of The IEEE*, 77(4):541–580, April 1989.
- Each place represents one device. Token sync means "join". 1-1 mapping to "fork-join".

4.3 Dataflow Process Networks

- **Classification**
  - Data-driven vs demand-driven
  - Static vs dynamic
- **Example networks**
  - Kahn network
  - Synchronous network by Ed. Lee
  - Dataflow machine by Arvind
- G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, 1974.
- E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

4.4 Modeling of Reactive Systems

- D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- Reactive system modeling

4.5 Modal and Temporal Logics

- Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1991.

4.6 Formal Verification

4.6.1 Model checking

4.6.2 Theorem proving

4.7 Protocol Synthesis

4.7.1 LOTOS

- ISO8807. Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behavior. ISO 8807: 1989 (E), February 1989.
- J. A. Manas and T. de Miguel. From LOTOS to C. In K. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Technique (FORTE'88)*. North-Holland, 1988.

4.7.2 SDL

4.8 Protocol synthesis

- G. J. Holzmann. *Design and Validation of Computer Protocols*, chapter 10. Protocol Synthesis. Prentice Hall, 1990.

5 OPERATING SYSTEMS

5.1 Operating Systems Practices

- D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, third edition, 2006.

6 COMPUTER NETWORKS

6.1 General Textbooks

- L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, second edition, 2000.

6.2 Wireless Protocols

6.2.1 802.15.4

- physical and data link layer

6.2.2 ZigBee

- network and application layer protocols on top of 802.15.4

7 DISTRIBUTED COMPUTING

7.1 General Textbooks

- J. Saltzer and M. F. Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers, 2009.
- S. Mullender, editor. *Distributed Systems*. Addison-Wesley, 2nd edition, 1993.
- M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Verlag, 2013.

7.2 Theoretical Foundations

7.2.1 Synchronization models

- D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

7.2.2 Remote procedure calls

- A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. SRC Research Report 115, Digital Equipment Corporation, 1994.
- M. Henning. The rise and fall of CORBA. *ACM Queue*, pages 28–34, June 2006.

7.2.3 Synchronizers

- B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

7.2.4 Logical clocks and clock synchronization

- L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

7.2.5 Authentication

- B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

7.2.6 Scheduling

- M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the 22nd Symposium on Operating System Principles (SOSP'09)*, pages 261–276, 2009.

7.2.7 Distributed lookup

- **center al coordinator**: Napster, GFS
- **flooding**: send queries to a large set of machines – Gnutella
- **DHT (distributed hash table**: Chord, CAN, Tapestry, Amazon Dynamo

7.3 Practical Issues

7.3.1 UUID: Universally Unique ID

7.4 Naming and Directory Services

7.4.1 LDAP

7.4.2 JNDI (Java Naming & Directory Interface)

- Usage #1: lookup

```
printer =
  (Printer) bldg7.lookup("puffin");
printer.print(document);
```

- Usage #2: get attributes

```
String[] attrs =
  {"workphone", "cellphone"};
boolsphons =
  directory.getAttributes(
   "cn=bob o=sales c=US"/*key*/,
   attrs);
```

- Usage #3: directory search

```
bobs = directory.search("cn=bob");
```

- **existing naming services**: **LDAP, DNS, NDS, ...**
- **Naming system**: has following components
  - **naming scheme** (or naming convention): simple names, compound (hierarchical) names
  - **context**: an object whose state is a set of bindings (from name to "object") with distinct atomic names
    - provides **lookup (resolution)** operation that returns an object, and may provide operations such as for **binding** names, **unbinding** names, **listing** bound names
    - **subcontext**: an atomic name in a context can be bound to another context object, say **subcontext**, giving rise to compound names
  - **naming system**: a connected set of contexts of the same type (i.e. have the same naming convention and provides the same set of operations with the same semantics)
  - **namespace**: the set of all names in a naming system
  - **composite name**: a name that spans multiple naming systems e.g. http://plato.mv.com/home/cjeong combines the DNS naming system (plato.mv.com) and filesystem naming system
  - *every name is interpreted relative to some context*
- **directory objects**
  - primary function of naming system: **mapping names to objects**
    - object can be any final atomic object or a directory object
  - directory object can have attributes, etc.
- **JNDI API**
  - **javax.naming**:
  - **javax.naminig.directory**:
  - **javax.naming.event**: events for object created, bound, unbound, etc.
  - **javax.naming.ldap**: LDAP v3
- **Java Naming and Directory Interface (JNDI) SPI**
  - SPI is what JDNI **service providers** need to implement
  - Factories:
    - **Object factories**: transforms "objects in naming/directory service system" into "Java objects"
    - **State factories**: transforms "Java objects" to "objects in **naming/directory services**
    - **Response control factories**: for narrowing LDAP v3 response controls received from LDAP services into more user-friendly types

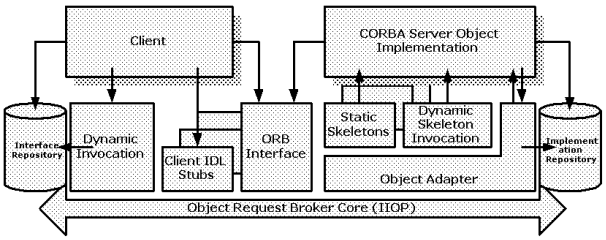
7.5 Distributed File Systems

- S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *The Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

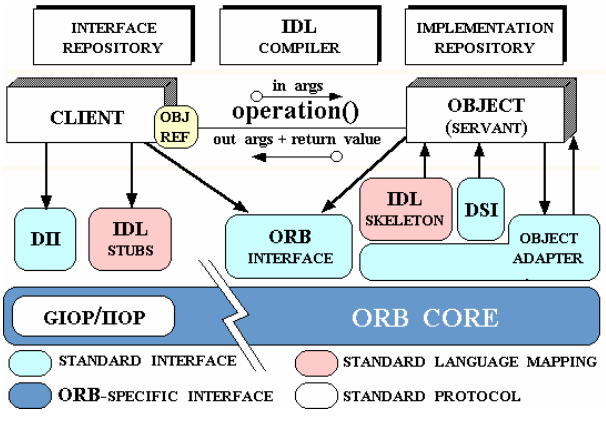
7.6 Distributed Objects

7.6.1 CORBA

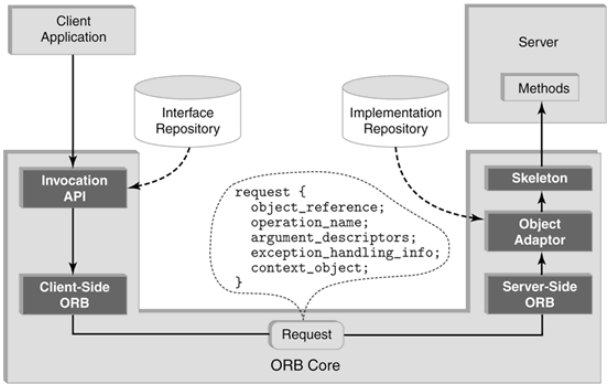
- **architecture**



- **alternate architecture**

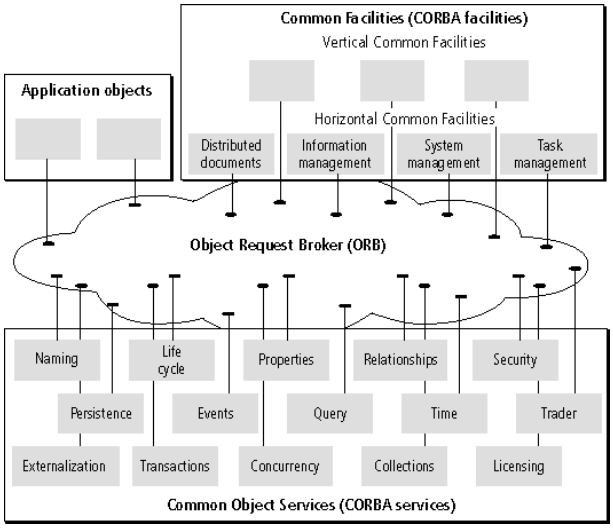


- **CORBA method invocation**

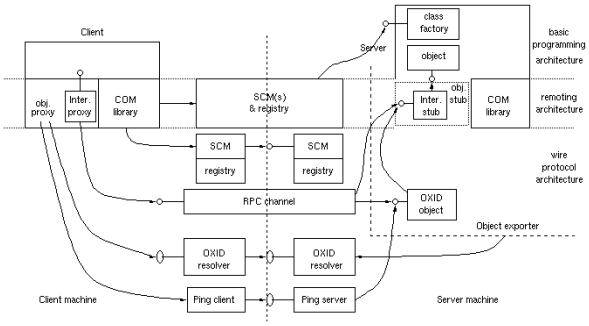


- **object references**: uniquely identify remote object instance
  - can be converted into string (but opaque)
  - consists of three information
    - **type name**: a.k.a. **repository ID**
    - **protocol and address details**: e.g. for IIOP, hostname+TCP-port-no
    - **object key**:

- **CORBA services and facilities**



7.7 DCOM



7.7.1 Java RMI

7.8 Cluster Systems

7.9 Grid Systems

7.10 P2P (Peer-to-Peer) Systems

- M. Surtani. Infinispan. In T. Armstrong, editor, *The Performance of Open Source Applications*, chapter 7. Lulu, 2013.

7.10.1 Distributed hash tables

- I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, And Protocols for Computer Communications (SIGCOMM '01)*, pages 149–160, 2001.

7.11 Jini – Apache River

7.11.1 JavaSpaces

- solves two problems:
  - distributed persistence
  - design of distributed algorithms
- A javaSpaces service holds **entries**
- An **entry** is a typed group of objects, expressed in a class for the Java platform that implements net.jini.core.entry.Entry.
- An entry can be **written** into a JavaSpaces service, which creates a copy of that entry in the space that can be used in future lookup operations
- Entry lookup can be done using **templates**, which are entry objects with all or some of its fields filled up with values to be matched exactly.
- To kinds of lookup **read** and **take**

7.12 Distributed System Performance

- B. Gregg. *Systems Performance: Enterprise and the Cloud*. John Wiley & Sons, 2013.

8 PARALLEL COMPUTING

8.1 Parallel Programming

- M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2012.
- J. S. Chase, F. G. Amador, E. D. Lazowska, H. H. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 147–158, 1989.

8.1.1 MPI

8.1.2 OpenMP

8.2 Memory Models

8.2.1 Java memory model

- JMM introduced since JSR133/Java5 – before this, when multiple threads access shared memory, all kinds of strange results occur; e.g.
  - **visibility problem**: a thread not seeing values written by other threads
  - **instruction reordering problem**: a thread observing "impossible" behavior of other threads, caused by instructions not being execution in the order expected
- JMM is a set of rules based on "**happens-before**" relation, which constrain when one memory access must happen before another, and conversely, when they are allowed to happen out of order. two examples are:
  - **the monitor lock rule**: a release of a lock happens before every subsequent ac quite of the same lock
  - **the volatile variable rule**: a write of a volatile variable happens before every subsequent read of the same volatile variable

8.2.2 C++ memory model

8.3 Threading Models

8.3.1 Pthreads

8.3.2 Java threading model

- based on shared memory and locking
- difficult to reason about (esp. when systems scale up in size and complexity)
- potential race conditions and deadlocks

8.3.3 C++ threading model

9 DATABASE SYSTEMS

9.1 Database Systems and Transaction Processing

- J. Gray and J. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, San Mateo, CA, 1993.

10 MOBILE AGENTS

10.1 General Textbooks

- J. Cao and S. K. Das, editors. *Mobile Agents in Networking and Distributed Computing*. Wiley, 2013.

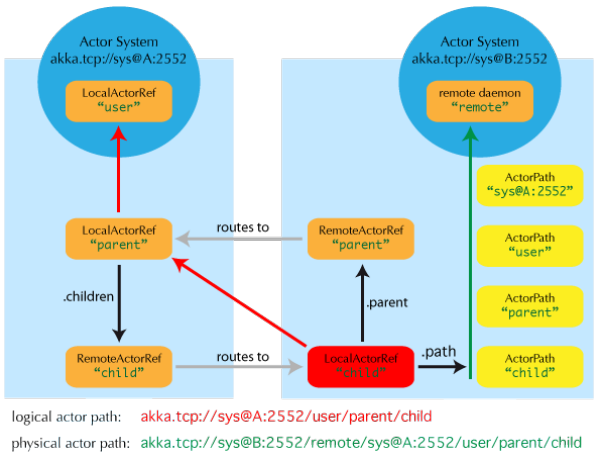
10.2 Actor Model of Computation

- C. Hewitt. Viewing control structures as patterns of passing messages. AI Memo 410, AI Laboratory, MIT, 1976.
- W. Clinger. Foundations of actor semantics. Technical Report 633, AI Laboratory, MIT, 1981.
- G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
- an **actor** is a computational entity which, in response to a **message** it receives, can concurrently:
  - send a finite number of messages to other actors; recipients are identified by **(mailing) address**
  - create a finite number of new actors
  - designate the behavior to be used for the next message it receives
- actor is based on message passing (cf. shared memory), allowing **asynchronous communication**
- **locality**: actor can send messages only to 1) addresses that it receives in the message, 2) addresses that it already had, 3) addresses that it synthesized
- **migration**: actors can change locations

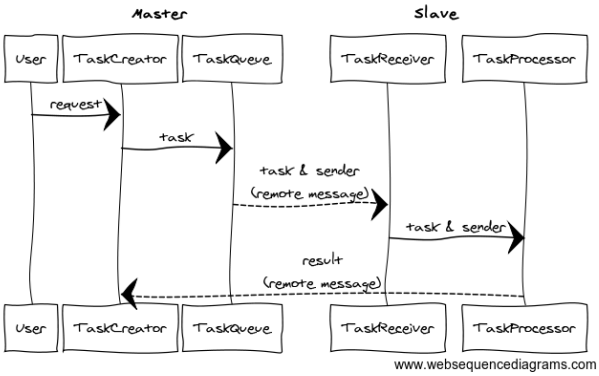
10.2.1 Scala

10.2.2 Akka

- Akka was written in Scala
- actor as very lightweight event-driven processes (roughly 2.7M actors per GB RAM; i.e. 370 bytes per process)
- can be used either as 1) library or as 2) microkernel
- See **meadow-akka-scala-actor** for details about "ActorRef", "Actor Path", and "Actor System"



- Akka flow diagram



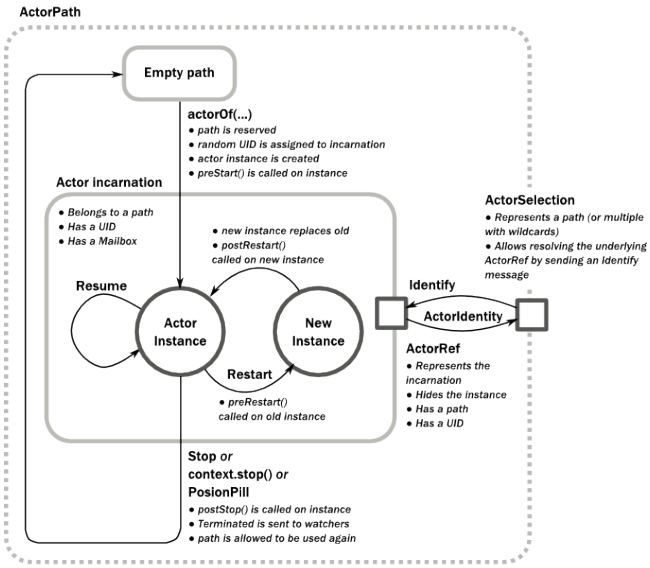
- Akka configuration

```
akka {
  actor {
    provider =
      "akka.remote.RemoteActorRefProvider"
  }
  remote {
    transport =
      "akka.remote.netty.
       NettyRemoteTransport"
    netty {
      port = 2552
    }
  }
}
```

- lookup remote actors:

```
val actor = actorSystem.actorFor(
  "akka://actorSystemName@server:"+
      "2552/user/actorName")
```

- Akka actor life-cycle



10.2.3 Typesafe activator: Reactive platform

10.2.4 Theron: C++ concurrency library

10.3 Agent Communication Languages

- B. Chaib-Draa and F. Dignum. Trends in agent communication language. *Computational Intelligence*, 18(2):89–101, 2002.

10.4 Mobile Agent Coordination

10.4.1 Temporal coupling

- Temporal coupling means that some form of synchronization needed between the interacting agents.
- Temporally-uncoupled systems: shared data space in common, used as repository for messages: *blackboard-based* or *tuple-based* systems
- **blackboard-based model**: shared space where explicitly target (destination) agent is stated
- **tuple-based (Linda-like) systems**: tuple is a structured set of typed data times and coordination between agents are performed indirectly via exchange of tuples through a shared tuple space
  - **coordination operations**
    - **rd**: read a tuple from the tuple space
    - **in**: extract a tuple from the tuple space
    - **out**: write a tuple in the tuple space
  - associative mechanism to get tuples from the space is based on **matching rule**

10.4.2 Spatial coupling

- Spatial coupled system: agents can communicate by **explicitly naming the receiving agents**
- Spatial coupling requires naming or location service.

10.4.3 Notion of roles

- Formally, a role is a relation over “agent types”
- Reminds me of **interfaces** in SystemVerilog

10.5 Resource and Service Discovery

10.6 Example Systems

10.6.1 IBM Aglets

10.6.2 D'Agents (aka Agent Tcl)

10.6.3 ObjectSpace Voyager

10.6.4 General Magic Odyssey

10.6.5 IKV Grasshopper

10.6.6 Sun JavaSpace

10.6.7 LIME (Linda in Mobile Environment)

10.6.8 SwarmLinda

10.6.9 TuCSoN

10.6.10 MARS

- programmable coordination architecture for mobile agents

10.7 Process Migration

- D. Milojičić, F. Douglis, and R. Wheeler, editors. *Mobility: Processes, Computers, and Agents*. ACM Press, 1999.
- Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 21(2):23–36, 1988.
- D. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. In *ACM Computing Surveys*, volume 32, pages 241–299, September 2000.

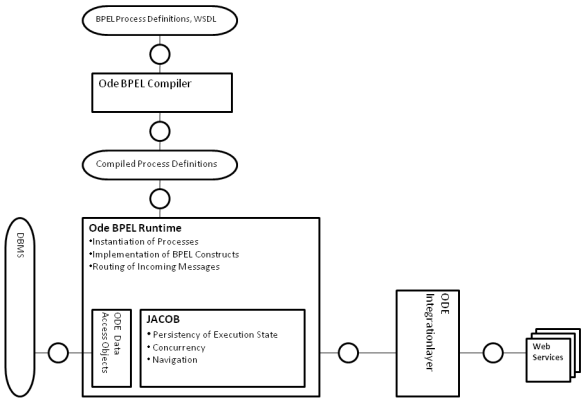
11 WEB SERVICES

11.1 Orchestration of Web Services

11.1.1 WS-BPEL (Business Process Exec Language)

- The OASIS Committee. Web services business process execution language (ws-bpel) version 2.0, 2007.

11.1.2 Apache ODE (Orchestration Director Engine)



11.1.3 More orchestration languages

- **XPDL (XML Process Def Language)**
- **XLANG**
- **Microsoft WWF (Windows Workflow Foundation)**

11.2 Choreography of Web Services

- A. Barker, C. D. Walton, and D. Robertson. Choreographing web services. *IEEE Transactions on Services Computing*, 2(2):152–166, April–June 2009.

11.2.1 WS-CDL (WS Choreography Desc Language)

- W3C. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10, 2005.

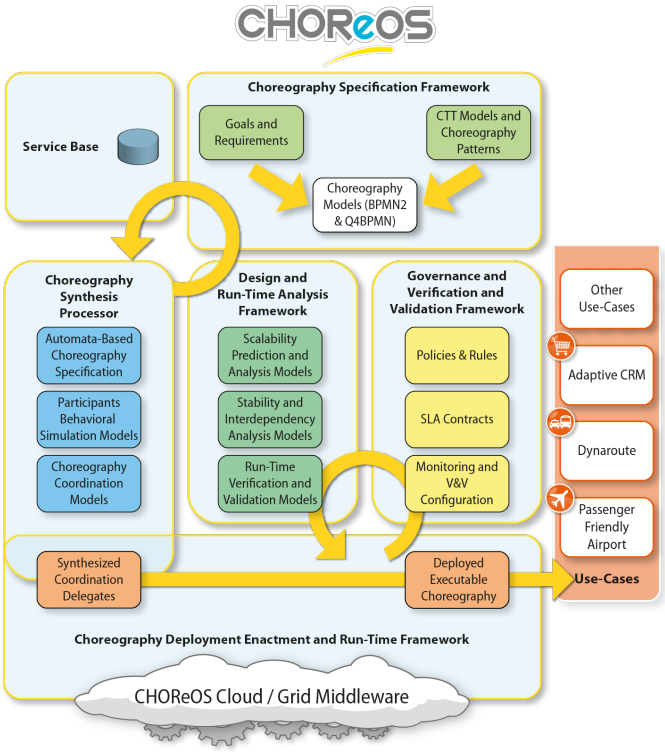
11.2.2 Projection: Choreography synthesis

- J. Mendling and M. Halfner. From inter-organizational workflows to process execution: Generating BPEL from WS-CDL. In *LNCS 3762: Proceedings of On the Move to Meaningful Internet Systems 2005*, pages 506–515, 2005.
- Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centered concurrent programming. Unpublished manuscript, 2006.

11.3 Large-Scale Choreography

11.3.1 CHOReOS

- `http://www.choreos.eu`



12 WORKFLOW SYSTEMS

12.1 General Textbooks

- W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
- L. Fischer, editor. *Workflow Handbook*. Future Strategies Inc., 2002.
- D. Hollingsworth. The workflow reference model. Document Number TC00-1003, The Workflow Management Coalition, 1995.
- B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the KEPLER system. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, 2005.

12.2 Workflow Patterns

12.2.1 Control flow patterns

- **sequence**
- **parallel split** (aka fork): all branches activated
- **exclusive choice** (XOR-split, conditional routing, switch, decision): exactly one outgoing branch is taken
- **multi-choice** (OR-split): n of m branches taken – for m branches, 2^m possible combinations of taken branches
- **synchronization** (AND-join): after all incoming edges finish
- **simple merge** (XOR-join, async join, merge): exactly one of incoming branch is ever taken – choice-merge
- **synchronizing merge**: n taken branches out of m is synchronized at the point
- **multi-merge**: m branches finish at the given point, then m firing follows (no corresponding construct in SV)
- **discriminator** (join_any): only the first finisher is respected

12.2.2 Structural patterns

- **arbitrary cycles**
- **implicit termination**: no other work to do – terminated

12.2.3 Patterns involving multiple instances

- **multiple instances without synchronization**
- **multiple instances with a priori design time knowledge**
- **multiple instances with a priori run time knowledge**
- **multiple instances without a priori run time knowledge**

12.2.4 State-based patterns

12.2.5 Cancellation patterns

12.3 Workflow Examples

- **mortgage application process**: bank, applicant, financial situation of applicant, bank resource (budget)

12.4 Distributed Workflow Systems

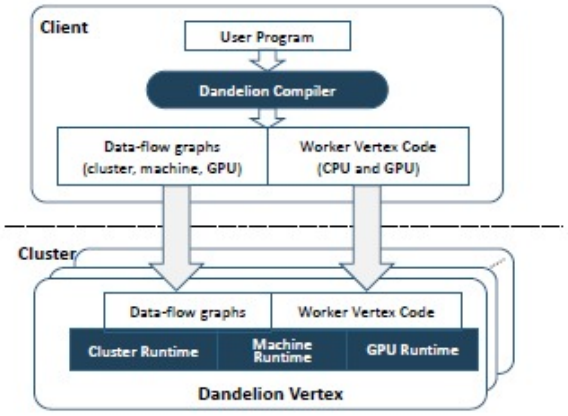
12.4.1 MapReduce

- J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*, pages 10–10, 2004.

12.4.2 Dandelion: Runtime for heterogeneous systems

- M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*, pages 59–72, 2007.

- C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*, pages 49–68, 2013.



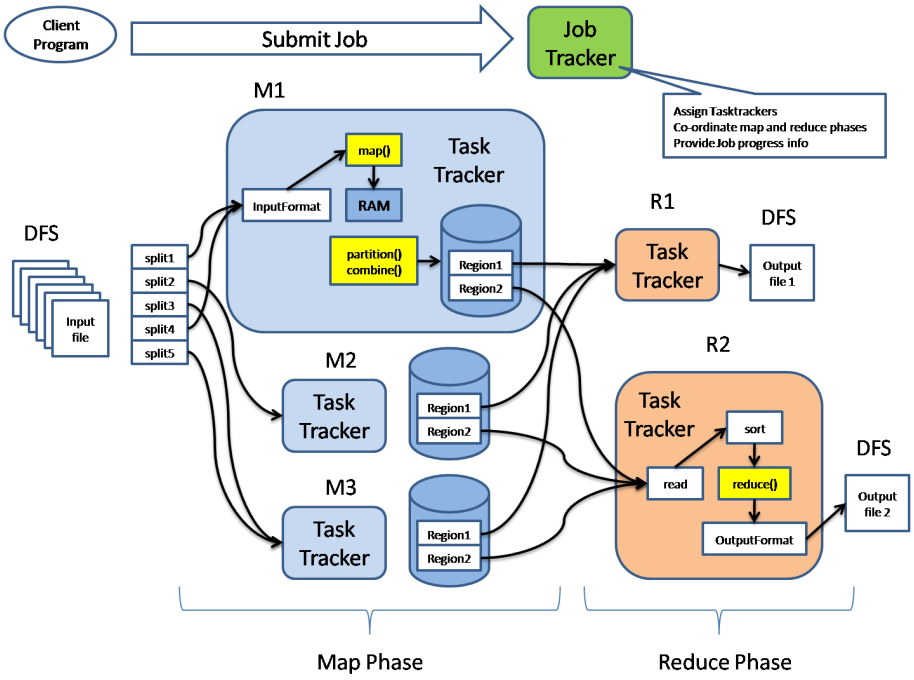
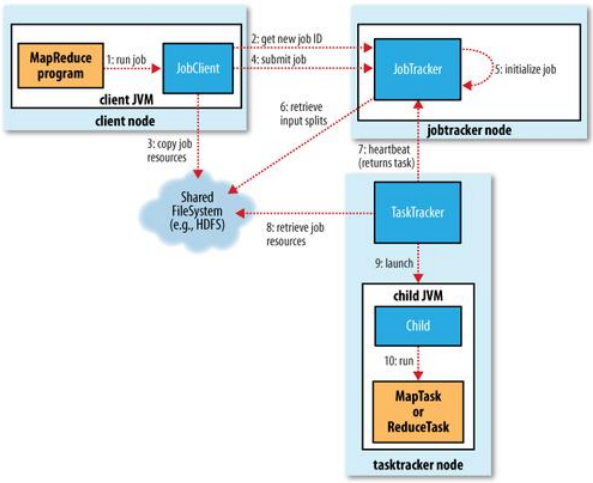


Figure 2: How Hadoop works

12.4.3 Hadoop



12.4.4 Pig: Graph-based workflow on Hadoop

12.4.5 Cascading

12.5 Scientific Workflow Systems

12.5.1 Swift: Distributed parallel scripting

- M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, September 2011.
- Some similarities to Hadoop, Pig.
- A swift script consists of apps, where a **app** is a script taking files as inputs and outputs.

12.5.2 Kepler: Scientific workflow

12.6 Enterprise Workflow Systems

12.6.1 Activiti

12.6.2 Cascading

12.6.3 OSWorkflow

13 EVENT PROCESSING SYSTEMS

13.1 Event Publish/Subscribe Systems

- A.-M. Kermarrec and P. Triantafillou. XL peer-to-peer pub/sub systems. *ACM Computing Surveys*, 46(2):16:1–16:45, November 2013.
- A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving expressiveness and scalability in an Internet-scale event notification service. In *Proceedings of the 19 th ACM Symposium on Principles of Distributed Computing (PODC'00)*, July 2000.

13.2 Active Databases

-

13.3 CEP (Complex Event Processing)

- G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3):1–69, June 2012.

- O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2011.

13.3.1 Esper

13.3.2 Oracle CEP

14 MESSAGING SYSTEMS

14.1 Enterprise Integration Patterns

- Hohpe and Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Reading, MA, 2003.

14.2 JMS (Java Message Service)

- JSR 343: Java message service 2.0, May 2013.
- **typical JMS Client**
  1. use JNDI to find a **ConnectionFactory** object
  2. use JNDI to find one or more **Destination** objects
  3. use **ConnectionFactory** to create a JMS **Connection** with message delivery inhibited
  4. use **Connection** to create one or more JMS **Session**s
  5. use **Session** and **Destination**s to create **MessageProducer**s and **MessageConsumer**s
  6. tell the **Connection** to start delivery of messages

14.3 Messaging Protocols

14.3.1 AMQP

- obsolete -> see ZMQ

14.3.2 ZMQ

- implemented in ZeroMQ system

14.3.3 MQTT

- **servers**: centralized cloud-based servers (bad idea) (e.g. Xively, Device Cloud, ...)
- **brokers**: RabbitMQ, eMQTT (Erlang MQTT broker), ActiveMQ, ...
- **client libraries**: binding in different languages (C/C++/Clojure/Erlang/Java/Objective-C/Perl/PHP/Ruby/...) for different device platforms (Arduino/mbed/Nanode/Netduino/...) allows to use MQTT protocol

14.4 Message Queues

14.4.1 ActiveMQ

14.4.2 RabbitMQ

14.4.3 ZeroMQ (also, Crossroads I/O)

- P. Hintjens. *ZeroMQ*. O'Reilly & Associates, 2013.

14.4.4 TIBCO

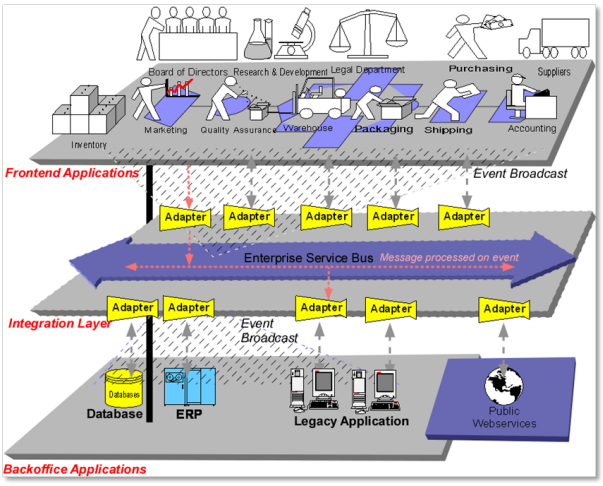
14.4.5 MSMQ (Microsoft Message Queueing)

14.4.6 Apache Camel

- provides integration framework through route building
- adding tap to components

14.5 ESB (Enterprise Service Bus)

- D. A. Chappell. *Enterprise Service Bus*. O'Reilly & Associates, 2004.
- T. Rademakers and J. Dirksen. *Open Source ESBs in Action*. Manning Publications Co., 2009.



14.5.1 OpenESB

14.5.2 Mule

14.5.3 Apache ServiceMix (incl. Apache Camel)

14.5.4 Apache Synapse

14.5.5 D-Bus

- **asynchronous** message buss for *interprocess communication* that forms the backbone of GNOME/KDE desktop
- shared bus architecture
- applications connect to a bus (identified by a **socket address**) and can either transmit a targeted message to another application on the bus, or broadcast a signal to all bus members
- **D-Bus daemon**: all processes communicate via the **D-Bus daemon**, which handles:
  (a) message passing
  (b) name registration
- **types of buses**
  - **system bus**: allows users to communicate with system-wide components (printer, bluetooth, H/W devices, etc.); shared by all users

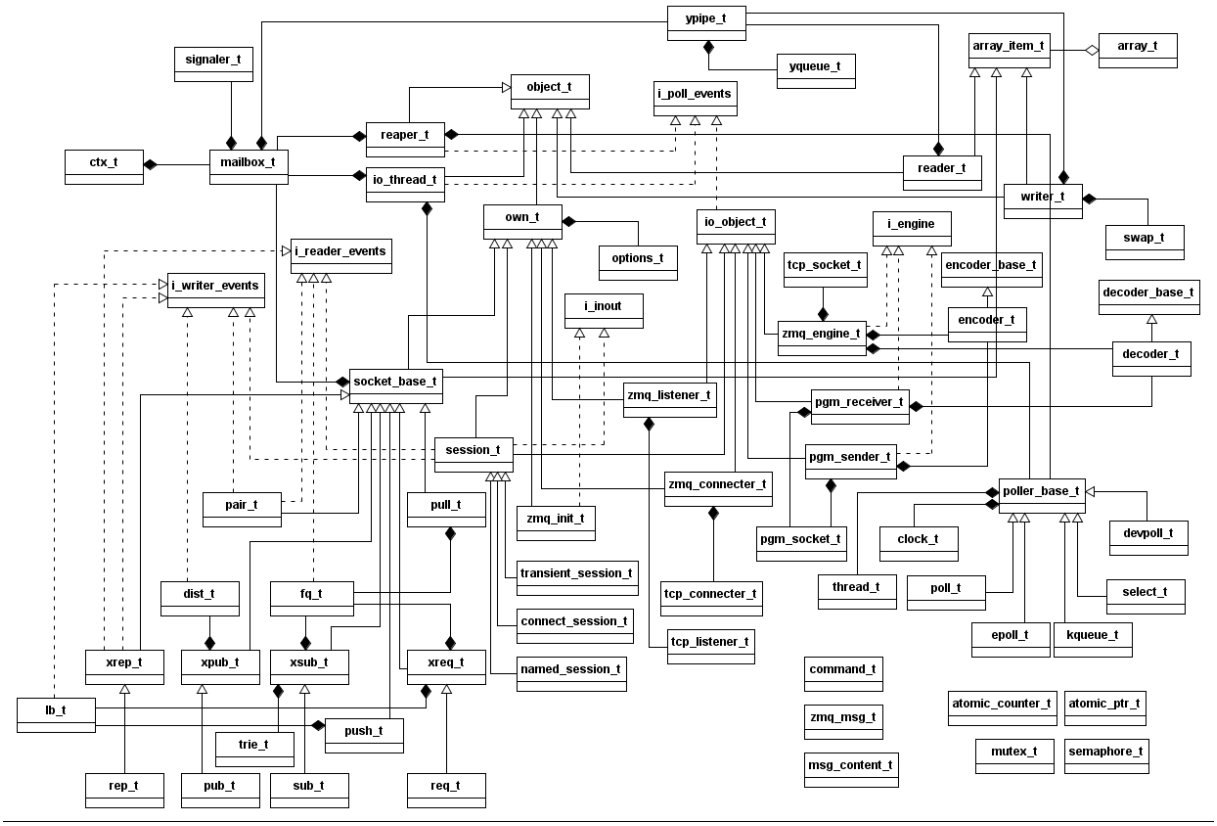


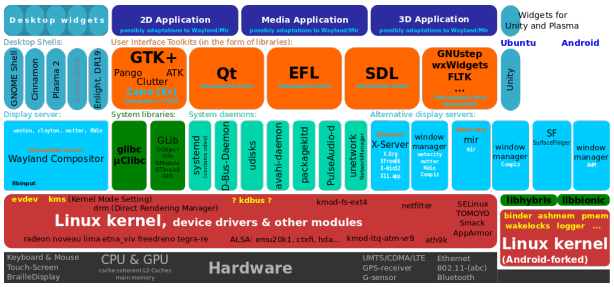
Figure 3: ZeroMQ 2.0 Classes

  - **session bus**: unique to the user (one session bus for each logged-in user); used for user's applications to communicate with each other

- for applications to adopt D-bus protocol, there are several libraries (e.g. GDBUS, libdbus, etc.), which allows:

  - to send/receive D-bus messages

  - marshalling/unmarshalling types from language's type-system to D-bus type-system



15 ENTERPRISE COMPUTING

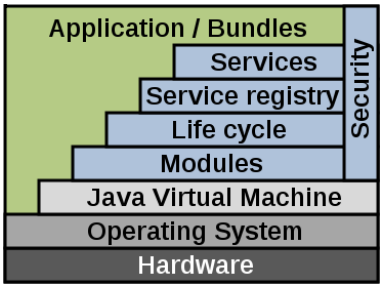
15.1 Java EE

15.2 OSGI

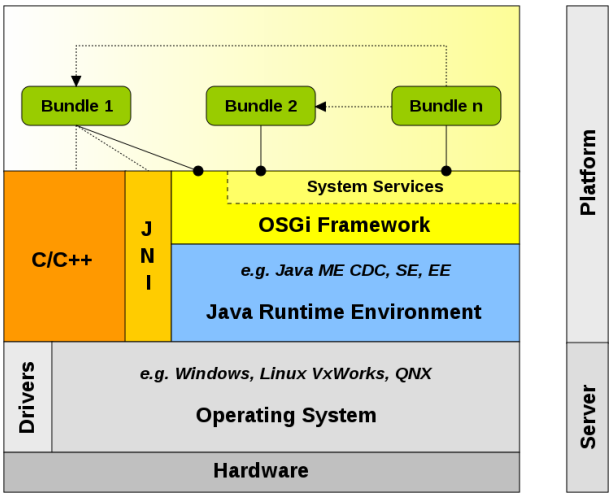
- service platform that implements complete and dynamic **component model** for Java

- components are in the form of **bundles for deployment** – remotely installed, started, stopped, updated, and uninstalled

- OSGI service gateway architecture:



- Classification: OSGI



15.3 SOA in General

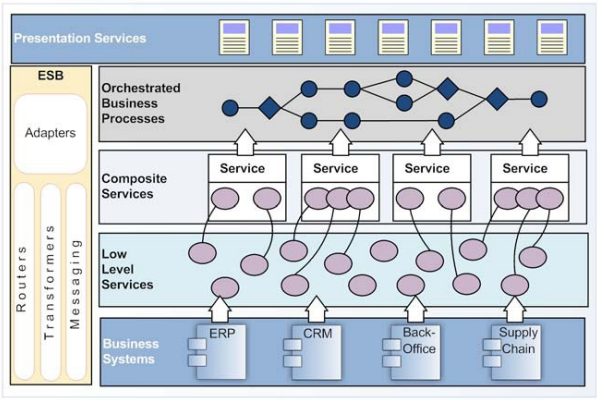


Figure 4: SOA Environment

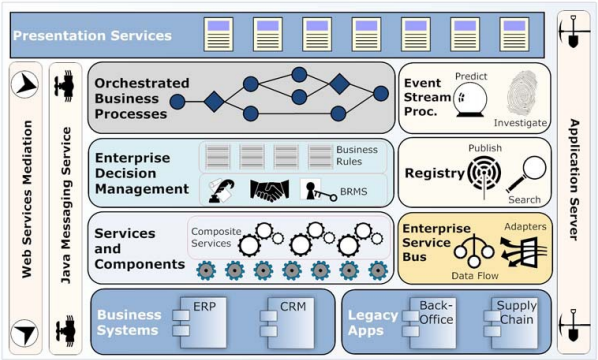


Figure 5: SOA Technology Platform

16 PERVASIVE COMPUTING

16.1 Wireless Sensor Networks

- D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, pages 59–69, Jan–Mar 2002.

16.2 Operating Systems for Devices

16.2.1 TinyOS

- microthreaded OS with small size (200-400B)
- TinyOS = **tiny scheduler + a graph of components**, where each *component* consists of four parts:
  (a) a set of *command handlers*
  (b) a set of *event handlers*
  (c) an encapsulated fixed-size *frame*
  (d) a bundle of simple *tasks*
- tasks, commands, handlers execute in the context of the **frame** and operate on its **state**
  - **frame**:
  - **command**: nonblocking request made to lower-level components; command will deposit request parameters into its frame and conditionally post a task for later execution; it may also invoke lower commands, but it must not wait for long or indeterminate latency actions to take place; a command must provide feedback to its caller by returning status indicating whether it was successful or not, e.g., buffer overrun.
  - **event handler**: invoked to deal with *hardware events*, either directly or indirectly
  - **task**: primary work performed atomically (i.e. run to completion), though can be preempted by events;
    1. call lower-level commands
    2. signal higher-level events
    3. schedule other tasks within a component
  - **task scheduler**:

16.2.2 Mote

16.2.3 ArdOS for Arduino

16.3 Database Systems for Devices

16.3.1 TinyDB

16.4 Programming Languages for Devices

16.4.1 nesC

- event-driven execution, flexible concurrency model, and component-oriented application design
- supports *compile-time data race detection*
- **assumptions**:
  1. all resources are known statically

2. rather than employing a general-purpose OS, applications are built from a suite of reusable system components coupled with application-specific code.

17 ROBOTS, CARS, INDUSTRIAL AUTOMATION

17.1 CAN Bus

-

17.2 V2V

-

17.3 Robot Operating Systems

17.3.1 ROS

- **ROS middleware**: the lowest level of ROS software stack – offers a message passing interface that provides means for inter-process communication
  - **publish/subscribe anonymous message passing**: asynchronous call
  - **recording and playback of messages**:
  - **request/response remote procedure calls**: for synchronous call – preemptible
  - **distributed parameter system**: global key-value store
- **Robot description language**
- **Robot-specific library**

17.3.2 OPRoS

17.4 OPC (Open Platform Communication)

17.4.1 Overview

OPC (`http://opcfoundation.org`) is the interoperability standard for the secure and reliable exchange of data in the industrial automation space and in other industries. It is platform independent and ensures the seamless flow of information among devices from multiple vendors. The OPC Foundation is responsible for the development and maintenance of this standard.

17.4.2 OPC UA (Unified Architecture)

The OPC Unified Architecture (UA), released in 2008, is a platform independent service-oriented architecture that integrates all the functionality of the individual OPC Classic specifications into one extensible framework.

This multi-layered approach accomplishes the original design specification goals of:

- **Functional equivalence**: all COM OPC Classic specifications are mapped to UA
- **Platform independence**: from an embedded micro-controller to cloud-based infrastructure
- **Secure**: encryption, authentication, and auditing
- **Extensible**: ability to add new features without affecting existing applications
- **Comprehensive information modeling**: for defining complex information

Functional Equivalence Building on the success of OPC Classic, OPC UA was designed to enhance and surpass the capabilities of the OPC Classic specifications. OPC UA is functionally equivalent to OPC Classic, yet capable of much more:

- **Discovery**: find the availability of OPC Servers on local PCs and/or networks
- **Address space**: all data is represented hierarchically (e.g. files and folders) allowing for simple and complex structures to be discovered and utilized by OPC Clients
- **On-demand**: read and write data/information based on access-permissions
- **Subscriptions**: monitor data/information and report-by-exception when values change based on a client's criteria
- **Events**: notify important information based on client's criteria
- **Methods**: clients can execute programs, etc. based on methods defined on the server

Integration between OPC UA products and OPC Classic products is easily accomplished with COM/Proxy wrappers that are available in the download section.

Platform Independence Given the wide array of available hardware platforms and operating systems, platform independence is essential. OPC UA functions on any of the following and more:

- **Hardware platforms**: traditional PC hardware, cloud-based servers, PLCs, micro-controllers (ARM etc.)
- **Operating Systems**: Microsoft Windows, Apple OSX, Android, or any distribution of Linux, etc.

OPC UA provides the necessary infrastructure for interoperability across the enterprise, from machine-to-machine, machine-to-enterprise and everything in-between.

Security One of the most important considerations in choosing a technology is security. OPC UA is firewall-friendly while addressing security concerns by providing a suite of controls:

- **Transport**: numerous protocols are defined providing options such as the ultra-fast OPC-binary transport or the more universally compatible SOAP-HTTPS, for example
- **Session Encryption**: messages are transmitted securely at 128 or 256 bit encryption levels
- **Message Signing**: messages are received exactly as they were sent
- **Sequenced Packets**: exposure to message replay attacks is eliminated with sequencing
- **Authentication**: each UA client and server is identified through OpenSSL certificates providing control over which applications and systems are permitted to connect with each other
- **User Control**: applications can require users to authenticate (login credentials, certificate, etc.) and can further restrict and enhance their capabilities with access rights and address-space "views"
- **Auditing**: activities by user and/or system are logged providing an access audit trail

Extensible The multi-layered architecture of OPC UA provides a "future proof" framework. Innovative technologies and methodologies such as new transport protocols, security algorithms, encoding standards, or application-services can be incorporated into OPC UA while maintaining backwards compatibility for existing products. UA products built today will work with the products of tomorrow.

17.4.3 OPC Classic

- **OPC Data Access (OPC DA)**: The OPC DA specification defines the exchange of data including values, time and quality information.
- **OPC Alarms & Events (OPC A&E)**: The OPC A&E specification defines the exchange of alarm and event type message information, as well as variable states and state management.
- **OPC Historical Data Access (OPC HDA)**: The OPC HDA specification defines query methods and analytics that may be applied to historical, time-stamped data.

18 DIGITAL SYSTEMS

18.1 Hardware Description Languages

18.1.1 Verilog

- IEEE Std 1364-2001. IEEE Standard for Verilog Hardware Description Language, 2005.

18.1.2 Esterel

18.2 Asynchronous Circuit Synthesis

18.2.1 Phillips handshake circuits

18.2.2 BALSA

18.2.3 Petrify

19 VIRTUALIZATION

19.1 Software Defined Networking (SDN)

19.1.1 Summary

- a.k.a. Network Virtualization (NV)
- provides access to the forwarding plane of the network switch
- OLD: packet arrives → routing table lookup → automatically forwarded
- NEW: with SDN installed on routers and switches, users can see flow table and software-configure the network layout and traffic flow
  - e.g. netadmin can control the priority of packet switches (video packet preferred over emails

19.1.2 OpenFlow

19.2 Software Defined Storage

- a.k.a. Storage Virtualization (SV)

19.2.1 NetApp

19.3 Software Defined Data Center

19.4 Software Defined Radio

20 SOFTWARE ARCHITECTURE AND PARADIGMS

20.1 Event-Based Programming Model

20.1.1 JavaScript

20.1.2 Twisted

20.1.3 X-Windows

20.2 Continuations

20.3 Coroutines

- D. Beazley. A curious course on coroutines and concurrency. http://dabeaz.com/coroutines, 2010.
- After all, talking distributed applications are coroutines.

20.4 Reactor/Proactor Design Pattern

20.4.1 C10K problem

`http://www.kegel.com/c10k.html`

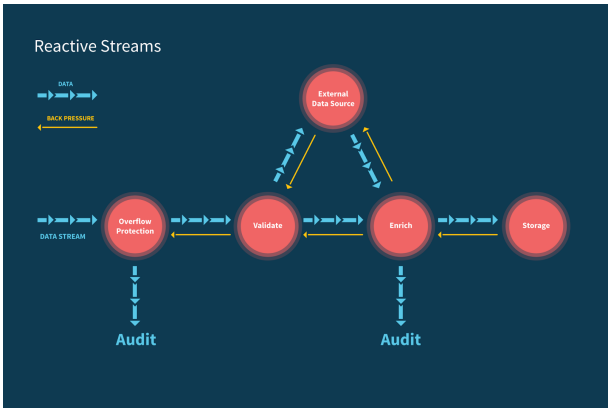
20.4.2 C libevent

20.4.3 Python gevent

20.4.4 Akka I/O module

20.4.5 Boost.asio

20.4.6 Akka: Reactive stream



20.5 Aspect-oriented Programming

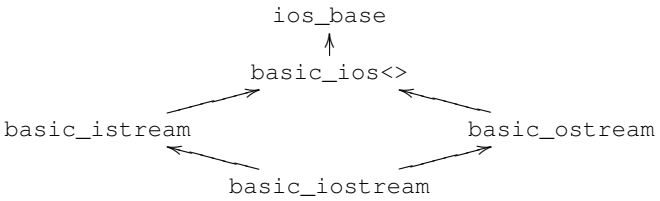
- cross-cutting concerns.
- it's just making function calls "hookable"; event generation when function call begins and ends
- NOTE: AOP was proposed by Kiczales, the inventor of Meta-Object Protocol (CommonLoops), which explains much of the philosophy of AOP

20.6 I/O systems

20.6.1 Java IO and NIO

20.6.2 C++ Streams I/O

- A **stream** is either
  1. a standard stream such as **cout**, **cin**, and **cerr**, or
  2. one created over a *file* or *device*
- **buffered stream** is a stream where *buffering* is enabled;
- **buffering** has following benefits:
  1. allows to cope with speed mismatch between producer and consumer
  2. allows producer to be nonblocking (no need to wait until the value is consumed)



- **C++ output streams**
  - **ostream** converts "values of various types" into **sequences of characters** (or **byte sequence**)

20.7 Telepathy: Instant messaging

- "communications as a service" (c.f. "printing as a service" – each application don't need to implement printing functionality; it can just use printing service)
- abstracting "communication" outside of an application
- **ConnectionManager**: one for each service (e.g. IRC, SIP, XMPP, etc.)
- **AccountManager**: for storing user's communication accounts and establishing a connection to each account via the appropriate connectino manager when requested
- **ChannelDispatcher**: listens for incoming channels signaled by each ConnectinoManager and dispatch them to clients that indicate their ability to handle that type of Channel (such as text, voice, video, file transfer, tubes)
  - also provides a service so that applications can request outgoing channels and have them handled locally by the appropriate client
- **Telepathy clients**
- **Connection**
  - A **Connection** will contain multiple **Channel**s
  - **Channel**: the mechanism through which the communications are carried out (e.g. IM conversion, file transfer, voice or video call)
- How Telepathy uses D-Bus
  - every **service** publishes a **object path**, like `org/freedesktop/Telepathy/AccntMgr`
  - A service implements multiple **interfaces**
    - a interface is strictly namespaced, like `org.freedesktop.DBus.properties` or `oftT.connection`
    - a interface provides **methods**, **signals**, or **properties that you can call/listen-to/request**
  - **publishing D-Bus object** (= service registration)
    - add new (object-path, object) pair to the map of (object-path, object) pairs
  - interfaces, methods, signals, properties are detailed in **XML-based IDL**
    - can be used to generate stubs/wrappers and language bindings

21 EXAMPLE SYSTEMS

21.1 Communication Protocols

Some are **request-response**-type of protocols while others are **publish-subscribe**-type of protocols. We need both.

21.1.1 HTTP

21.1.2 RESTful HTTP

21.1.3 CoAP

21.1.4 MQTT

21.1.5 AMQP

21.1.6 AMQP-based DDS

21.1.7 SSI

21.1.8 XMPP

21.1.9 SSI

21.2 Protocol Standards Organization

21.2.1 OASIS

21.3 Middleware

21.3.1 Qualcomm AllJoyn

- **Allseen Alliance**: Qualcomm, Microsoft, LG, Haier, Panasonic, Sharp, Technicolor, Silicon Image

21.3.2 Open Interconnect Consortium

- Intel, Samsung, Atmel, Broadcom, Dell, Wind River

21.3.3 Thread Group

- Google Nest Labs, ARM, Freescasle, Samsung, Yale Security, Silicon Labs

21.3.4 Apple HomeKit

21.3.5 Google Glass

- https://developers.google.com/glass/
- Voice command: "Get direction to San Francisco"

```
// navigation_trigger.xml
<trigger command="GET_DIRECTIONS_TO">
  <constraints
    network="true"
  />
  <input />
</trigger>

// AndroidManifest.xml
<activity android:name=
          "NavigationActivity" />
  ...
  <intent-filter>
    <action android:name=
     "com.google.android.glass.
          action.VOICE_TRIGGER"/>
  </intent-filter>
  <meta-data
    android:name=
      "com.google.android.
           glass.action.VoiceTrigger"
    android:resource=
      "@xml/navigation_trigger"/>
</activity>
```

21.3.6 Google Fit

21.3.7 Google Nest

21.3.8 Samsung SmartThings

- Cloud-based,
- Users create apps (to be executed in Cloud) using Groovy language

21.3.9 Electric Imp

- Squirrel language

21.3.10 Zonoff

21.4 Prototyping Boards

21.4.1 Arduino

- Atmel Atmega MCU

21.4.2 Raspberry Pi

21.4.3 TI Beaglebone

21.4.4 Intel Galileo

21.4.5 Spark Core

- http://spark.hackster.io
- 32-bit ARM CPU
- 2MB flash memory
- supports wireless deployment of code

21.5 Devices

21.5.1 Beacon

21.5.2 Fitbit

- TI: MSP430F261T low-power MCU
- Nordic: nRF24AP1 proprietary 2.4GHz RF chip with ANT low-power protocol
- Freescale: MMA7341L 3-axis MEMS accelerometer

21.5.3 Google Nest

21.5.4 Tagg GPS Pet Tracker

21.5.5 Aria weight scale

21.6 Misc systems

21.6.1 DIoTY.co

- a cloud based platform to help you experiment faster with your IoT projects.
- DIoTY currently provides you with two services,
  - an MQTT broker
  - a Node-RED development tool.
- **MQ Telemetry Transport (MQTT)**: lightweight broker-based publish/subscribe messaging protocol. MQTT was originally developed by IBM but is currently standardized by OASIS and is both free and royalty free. It is rapidly becoming one of the standards for IoT/M2M.
  - Clients (sensors, mobile apps,...) connect to a broker.
  - Clients communicate by sending and receiving messages to/from the broker. For the broker, a messages is just a chuck of data.
  - A client publishes a message to a topic (ego: /home/livingroom/temperature).
  - A client can subscribe to many topics. It will then start receiving all messages send to those topic(s).
- As you can easily see, an MQTT broker as provided by DIoTY can remove the need to run your own web server at home. Your sensors publish to the MQTT broker, your mobile app subscribes to the topics of interest and you're done...
- Node-RED is a visual tool for wiring the internet of things. Node-RED is a creation of IBM emerging technologies, open source licensed under Apache 2.0.
- With the Node-RED tool provided by DIoTY you can subscribe and publish to the MQTT broker. You can alter the messages by applying functions to it (eg: subscribe to /home/livingroom/temperature/c ; convert the temperature from Celsius to Fahrenheit and then publish again to the topic /home/livingroom/temperature/f).

- You can also interact over other protocols like http, websockets,... to retreive for example weather information from the bbc website and push it to your mobile application.
- Finally, with the twitter node, building your own twittering house becomes as easy as pie.