Outline

- Web Services: SOAP, WSDL, BPEL, UDDI
- JAX-WS, JAXB
- REST, JAX-RS
- Futures / Promises
- MPI
- Clouds
  - IaaS
  - PaaS
  - SaaS
- Google App Engine
- Storage
  - AppScale Data Stores
  - CouchDB == REST + MapReduce + NoSQL
Web Services

- Integration of information systems across the Internet
- **SOAP** – Simple Object Access Protocol
- **WSDL** – Web Services Description Language
  - DataType, BindingType, Address
  - Input { DataType }, Output { DataType }
  - Operation { Input, Output }
  - Interface { Operation ... }
  - Binding { BindingType, Interface, Operation ... }
  - Endpoint { Binding, Address }
  - Service { Endpoint ... }
- **UDDI** – Universal Description Discovery and Integration
  - Web Service registering other Web Services
  - White Page(s), Yellow Pages, Green Pages
Web Services

- **BPEL** - Business Process Execution Language
  - Service Orchestration (vs. Choreography)
  - Programming in the Large
- **JAX-WS** - Java API for XML Web Services
  - Standard for implementing WSDL interfaces in Java
  - Data model: JAXB (based on XML Schema) rather than SOAP
- **JAXB** - Java Architecture for XML Binding
- **JAX-RS** - Java API for RESTful Web Services
  - Maps Java classes to web resources using annotations
  - REST will be described later
Representational State Transfer

- Architecture of large web applications (WWW)
- Goals
  - Scalability, generality of interfaces
  - Independent components, intermediary components
- Constraints
  - Client (does not store) / Server (does not present)
  - Stateless
  - Cacheable
  - Layered system + Uniform interfaces across layers
  - “Code on Demand” (optional)
- CRUD – Create, Read, Update, Delete
  - Possible mapping to HTTP: POST, GET, PUT, DELETE
**Representational State Transfer**

- Full representation of resources transferred
- Resources identified by URIs
- Clients' progress driven by hyperlinks
- Messages are self-descriptive (stateless communication)
- **JAX-RS** – Implementing RESTful APIs in Java based on annotations
  - @Path
  - @GET, @PUT, @POST, @DELETE
  - @Produce, @Consume
- REST examples
  - World Wide Web
  - Google Data
  - CouchDB – Cluster of Unreliable Commodity Hardware
Futures and Promises

- Results of RPC have to be waited for...
- But what if they are not needed at that moment?
  - Stored for later use
  - Used (only) as arguments to further remote calls
- Future / Promise
  - A placeholder for output from remote computations
  - Wait-By-Necessity
  - Implicit or Explicit (sometimes language-dependent)
- Goals
  - Parallel execution, **Pipelining**
- **ProActive** – a middleware implementing the Active Object pattern
  - Active objects == “body” + standard Java object
  - Implicit threads of execution, generation of futures
Message Passing Interface

- A rich framework for parallel and distributed computations
- Special tools for compilation and process launching
- **OpenMPI** – a multiplatform implementation
- **Peer-to-Peer communication**
  - Multiple flavors with different synchronization
  - Extensible message typing
- **Remote memory access**
  - “memory windows”
  - Put, Get, Fence, ...
- **Group communication**
  - Scatter / Gather
  - Scan, **Reduce**
  - Barrier, ...
MapReduce

- API designed by Google based on well-known LISP primitives
- \texttt{map}(k_1, v_1) \rightarrow \texttt{list}(k_2, v_2)
- \texttt{reduce}(k_2, \texttt{list}(v_2)) \rightarrow \texttt{list}(v_2)
- Common workloads that can be expressed as MapReduce
  - Distributed grep
  - Inverted index
  - Distributed sort
  - Reverse web-link graph
- Used by Google; indexing == A sequence of 15-20 MapReduce operations
- Simple parallel execution for unexperienced programmers, small code size ;-)
- Eager execution, fault tolerance (killing 10% of tasks slowed 1 TB sort by 5%)
- Strong consistency guarantees for commutative and associative \texttt{reduce}()
- Multiple implementations (\texttt{Hadoop})
- Needs distributed storage (GFS, HDFS)
Classification of Cloud Applications

- **IaaS** – Infrastructure as a Service
  - EC2, Eucalyptus
  - Customer's virtual machine images run on provider's hosts
  - Implementation: XEN, KVM, VMWare and the like

- **PaaS** – Platform as a Service
  - Data processing – **MapReduce**, …
  - Data storage – Cassandra, ROSE, …
  - At least 7 distributed databases supported by **AppScale**
  - Customer's data stored or processed by provider's hosts

- **SaaS** – Software as a Service
  - Google AppEngine, AppScale
  - Customer's applications hosted on the provider's cloud
What are we interested in? ("Wisconsin Benchmark")

- **Sizeup**
  - Constant hardware configuration
  - Varying data size
  - How does data size affect response time?
  - Wanted: *sublinear* dependency

- **Speedup**
  - Varying hardware configuration
  - Constant data size
  - How does hardware configuration affect response time?
  - Wanted: *(super)*linear dependency (of old_time / new_time on #nodes)

- **Scaleup**
  - Constant ratio between data size and hardware configuration
  - Varying data size and hardware configuration
  - How does it change the response time?
  - Wanted: *constant* response time
What are we interested in?

- Wisconsin benchmark classics: **sizeup**, **speedup**, **scaleup**
- Since 1990, some new requirements emerged...
- **Resource isolation**
  - CPU sharing, **disk sharing**, network sharing
- **Fault tolerance**
  - How quickly are faults detected? Do they increase latency?
- **Load balancing**
  - What are the costs, risks and benefits of **live migration**?
- **Elasticity**
  - Are dynamic reconfigurations (especially growth...) viable?
- **Replication**
  - What's the impact of replication on read / write latency?
  - What guarantees do we require?
Questions?