Chapter 13: Patterns and Tactics

What is a Pattern?

- **An architecture pattern** is a package of design decisions that is found repeatedly in practice
- has known properties that permits reuse, and describes a class of architectures

Architectural Patterns

- **Module patterns**
  - Layered pattern
- **Component-and-Connector patterns**
  - Broker pattern
  - Model-View-Controller pattern
  - Pipe-and-Filter pattern
  - Client-Server pattern
  - Peer-to-Peer pattern
  - Service-Oriented Architecture (SOA) pattern
  - Publish-Subscribe pattern
  - Shared-Data pattern
- **Allocation patterns**
  - Map-Reduce pattern
  - Multi-tier Pattern

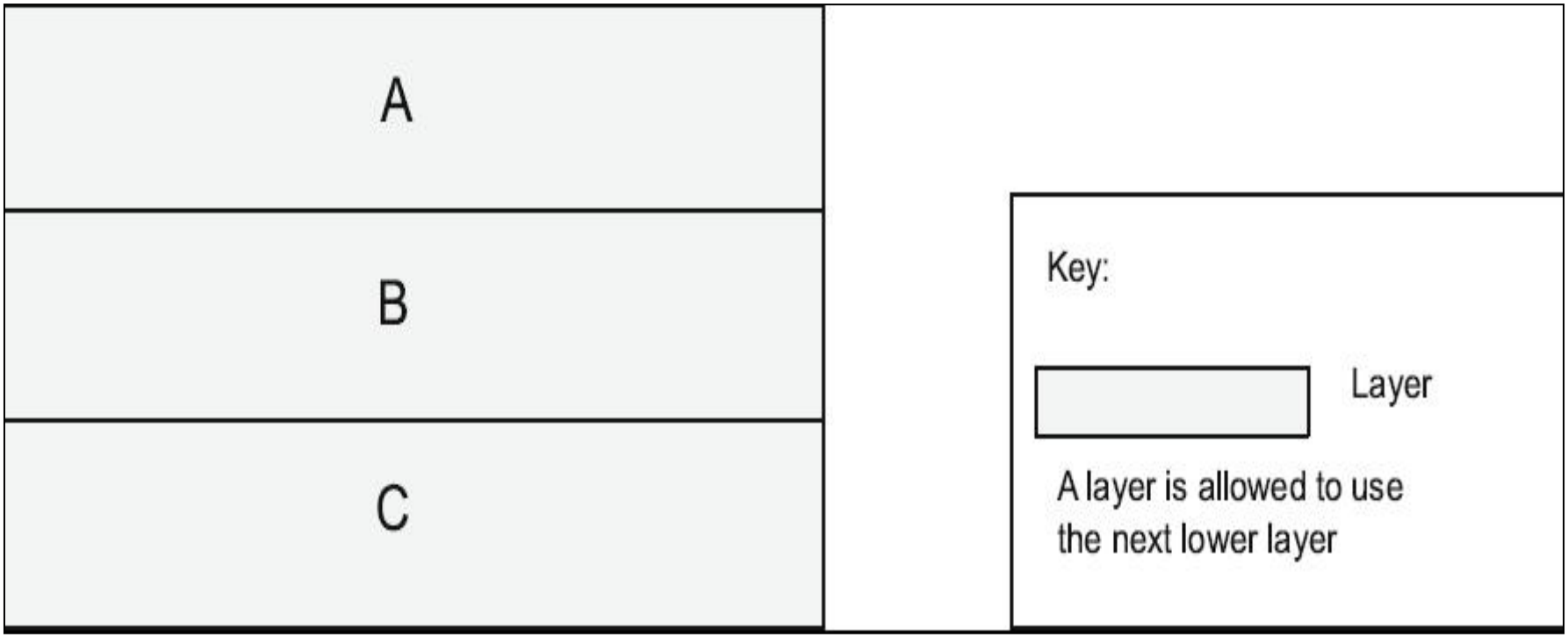
Layer Pattern

- **Context:** Modules of the system may be independently developed and maintained.
- **Problem:** To minimize the interaction among the different development organizations, and support portability, modifiability, and reuse.

Layer Pattern

- The layered pattern divides the software into units called layers.
- Each layer is a grouping of modules that offers a cohesive set of services.
- Each layer is exposed through a public interface.
- The usage must be unidirectional.

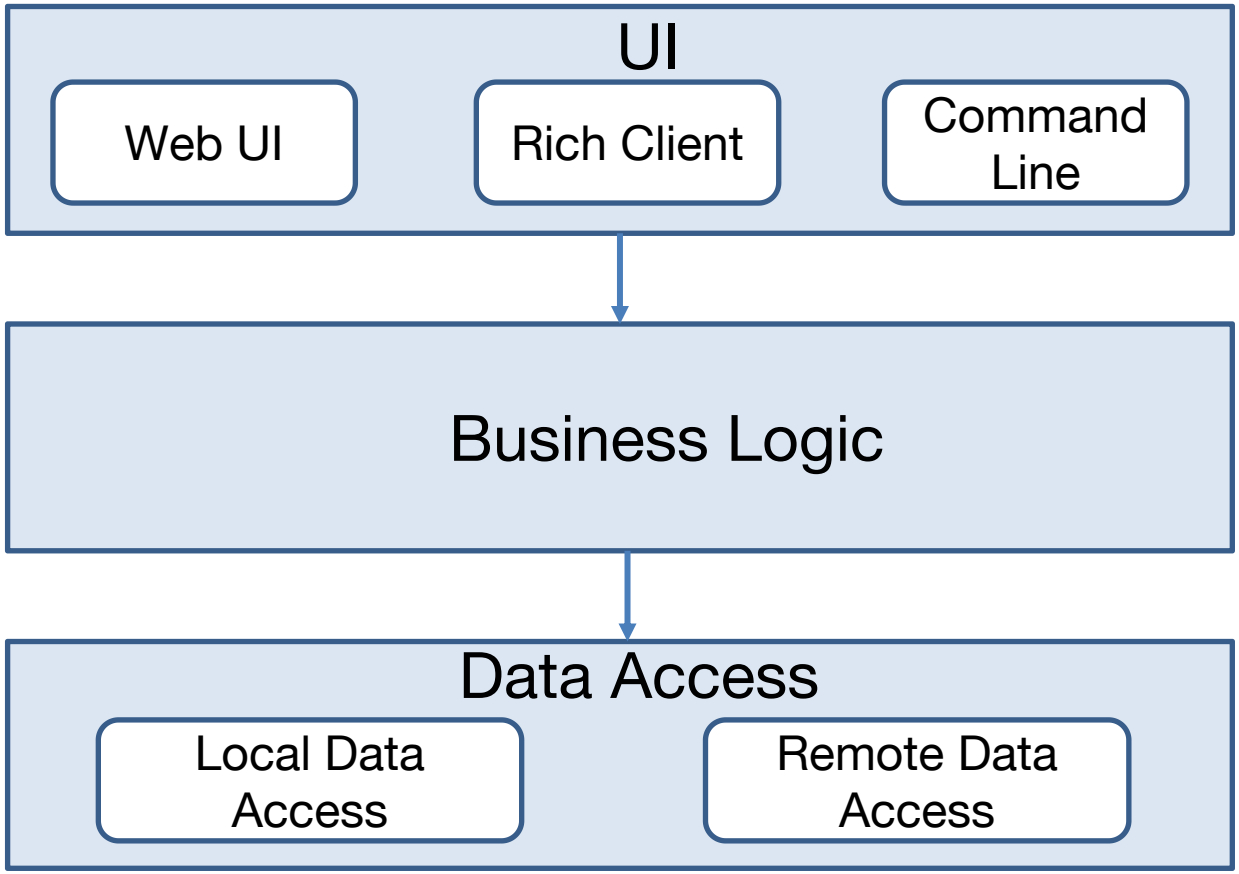
Layer Pattern Example



Layer Pattern Solution

- The layered pattern defines layers and a unidirectional *allowed-to-use* relation among the layers.
- **Constraints**:
  - Every piece of software is allocated to exactly one layer.
  - There are at least two layers
  - The *allowed-to-use* relations should not be circular
- **Weaknesses**:
  - The addition of layers adds up cost and complexity to a system.
  - Layers contribute a performance penalty.

Three layered applications



Architectural Patterns

- **Component-and-Connector patterns**
  - Broker pattern
  - Model-View-Controller pattern
  - Pipe-and-Filter pattern
  - Client-Server pattern
  - Peer-to-Peer pattern
  - Service-Oriented Architecture (SOA) pattern
  - Publish-Subscribe pattern
  - Shared-Data pattern

Broker Pattern

- **Context**: Many systems are constructed from a collection of services distributed across multiple servers
- **Problem**: How do we structure *distributed software* so that service users do not need to know the nature and location of service providers?
- **Solution**: The broker pattern separates clients from providers servers by inserting an *intermediary*, called a broker.

Broker Solution

- **Overview**: The broker pattern defines a runtime component, called a *broker*, that mediates the communication between a number of clients and servers.
- **Elements**:
  - *Client,* a requester of services
  - *Server,* a provider of services
  - *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client

Broker Solution

- **Constraints**: The client can only attach to a broker. The server can only attach to a broker.
- **Weaknesses**:
  - Brokers add latency between clients and servers, and it may be a communication bottleneck.
  - The broker can be a single point of failure.
  - A broker may be a target for security attacks.

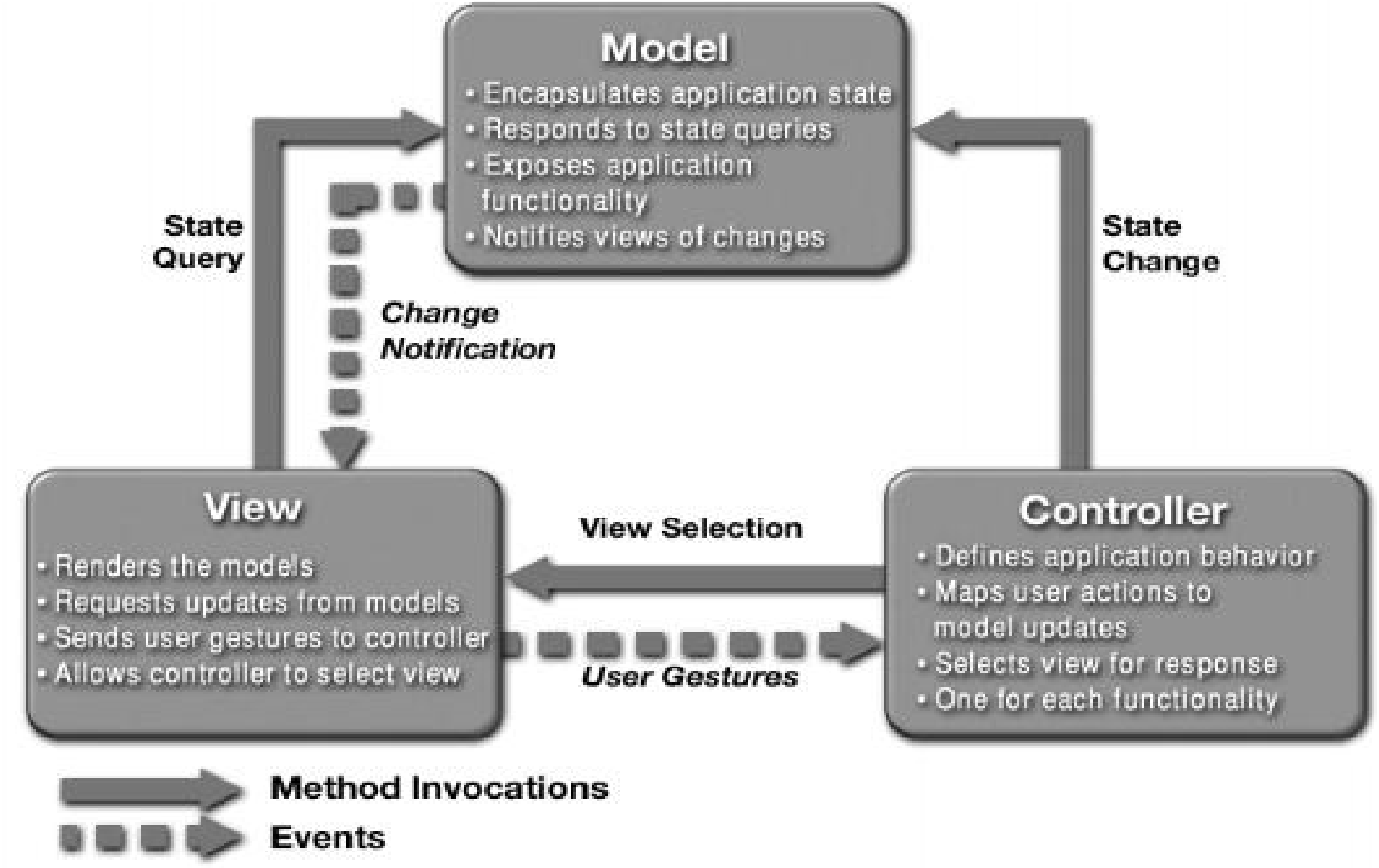
Model-View-Controller Pattern

- **Context**: *User interface software* is the most frequently modified portion of an interactive application.
- **Problem**: How can user interface functionality be kept separate from application functionality and yet still be *responsive to user input*, or *to changes* in the underlying application's data?
- And how can *multiple views of the user interface* be created, maintained, and coordinated when the underlying application data changes?

Model-View-Controller Pattern

- **Solution**: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:
  - ***A model***, which contains the application's data
  - ***A view***, which displays some portion of the underlying data and interacts with the user
  - ***A controller***, which mediates between the model and the view and manages the notifications of state changes

MVC Example

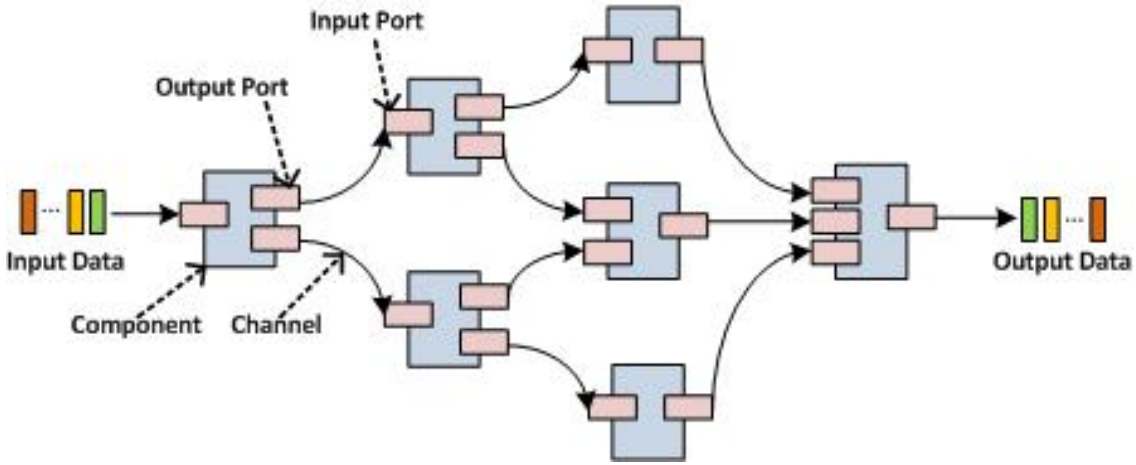

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

MVC Solution

- **Constraints**:
  - There must be at least one instance for each of model, view, and controller.
- **Weaknesses**:
  - The complexity may not be worth it for simple user interfaces.

Pipe and Filter Pattern

- **Context:** Streaming data processing
- **Problem:** How to speed up the data processing?
- **Solution:** Data arrives at a filter's input port, is transformed, and then is passed via its output port through a pipe to the next filter.
- A single filter can consume data from, or produce data to, one or more ports.



Pipe and Filter Solution

- **Elements**:
  - *Filter,* which is a component that transforms data read on its input port to data written on its output port.
  - *Pipe,* which is a connector that conveys data from a filter's output port to another filter's input port.
- **Relations**: The *attachment* relation associates the output of filters with the input of pipes and vice versa.
- **Constraints**:
  - Connected filters must agree on the type of data being passed along the connecting pipe.

Client-Server Pattern

- **Context:** There are *shared resources and services* that large numbers of distributed clients wish to access, and for which we wish to *control access or quality of service*.
- **Problem:** To improve availability by *centralizing the control* of these resources and services.
- **Solution:** Clients interact by requesting services of servers.
- There may be one central server or multiple distributed ones.

Client-Server Solution

- **Weaknesses**:
  - Server can be a performance bottleneck.
  - Server can be a single point of failure.
  - Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

Peer-to-Peer Pattern

- **Problem:** How can a set of “equal” distributed computational entities be connected to each other via a common protocol,
- such that they can organize and share their services with high availability and scalability?
- **Solution:** In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are “equal”.
- Peer-to-peer communication is typically a request/reply interaction *without the asymmetry* found in the client-server pattern.

Peer-to-Peer Solution

- **Weaknesses**:
  - Managing data consistency, data/service availability, backup, and recovery are all more complex.
  - Small peer-to-peer systems may not be able to achieve quality goals such as performance and availability.

Service Oriented Architecture Pattern

- **Problem**: How can we support *interoperability* of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?
- **Solution**: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services.

Service Oriented Architecture Solution - 1

- **Elements**:
  - *Components*:
    - *Service providers,* which provide one or more services through published interfaces.
    - *Service consumers,* which invoke services directly or through an intermediary.
  - *Enterprise Service Bus (ESB),* which is an ***intermediary element*** that can route and transform messages between service providers and consumers.
  - *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime.

Service Oriented Architecture Solution - 2

– Connectors:

- *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP.
- *REST connector,* which relies on the basic request/reply operations of the HTTP protocol.
- *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe *asynchronous message exchanges*.

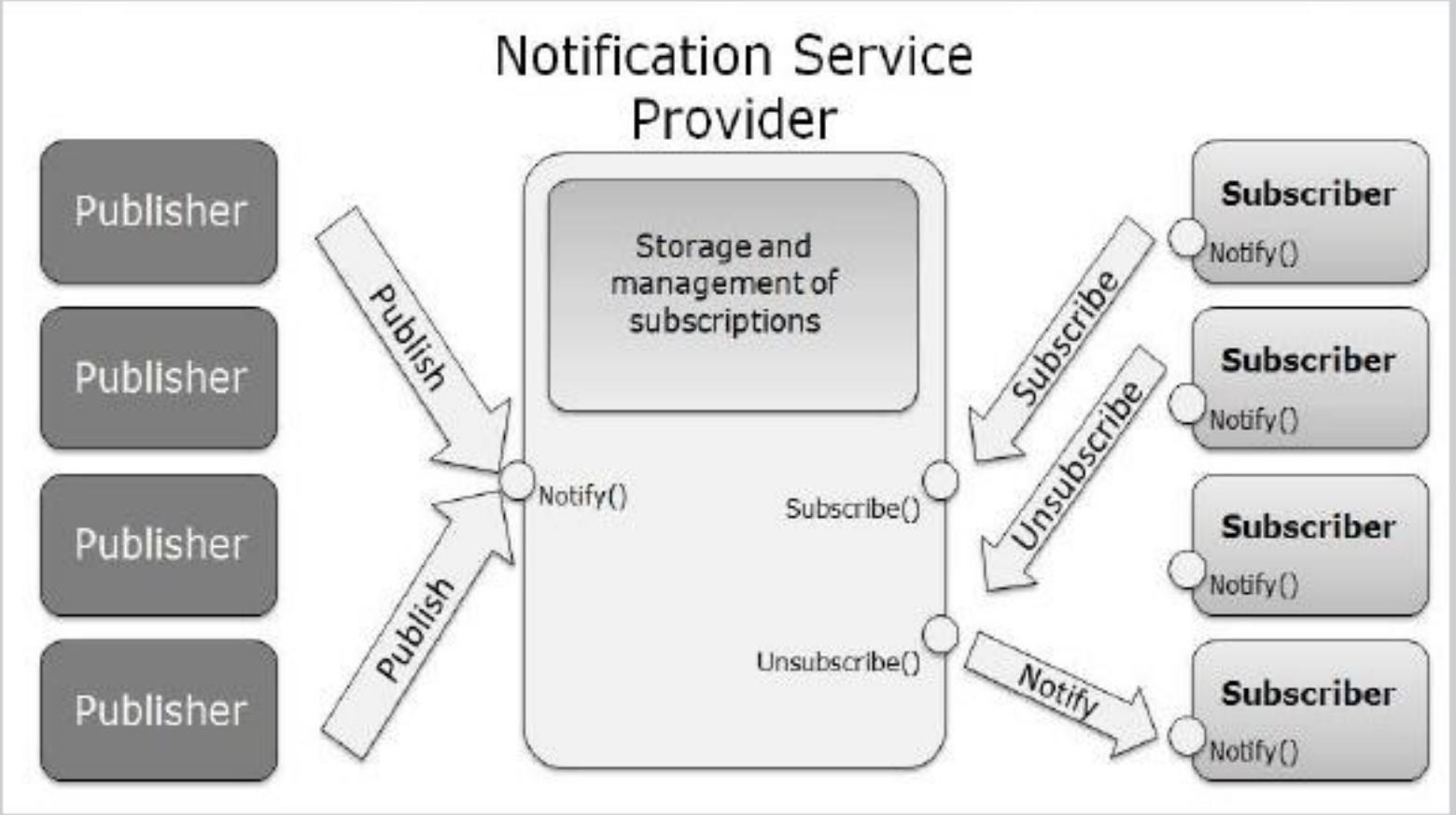
Service Oriented Architecture Solution - 3

- **Weaknesses**:
  - There is a *performance overhead associated with the middleware*, which may be performance bottlenecks, and typically do not provide performance guarantees.

Publish-Subscribe Pattern

- **Problem:** To transmit messages among the producers and consumers so they are *unaware of each other's identity, or even their existence*?

Publish-Subscribe Basic Model Overview



Advantages of Pub/Sub

- Highly suited for mobile applications, ubiquitous computing and distributed embedded systems
- Robust – Failure of publishers or subscribers does not bring down the entire system
- Scalability- Suited to build distributed applications consisting a large number of entities

Disadvantages of Pub/Sub

- Reliability – no strong guarantee on broker to deliver content to subscriber. After a publisher publishes the event, it assumes that all corresponding subscribers would receive it.

- Potential bottleneck in brokers when subscribers and publishers overload them. (Solve by load balancing techniques)

Shared-Data Pattern

- **Context:** Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.
- **Problem:** How can systems store and manipulate persistent data that is accessed by multiple independent components?
- **Solution:** In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*.
- Exchange may be initiated by the accessors or the data store. The connector type is *data reading and writing*.

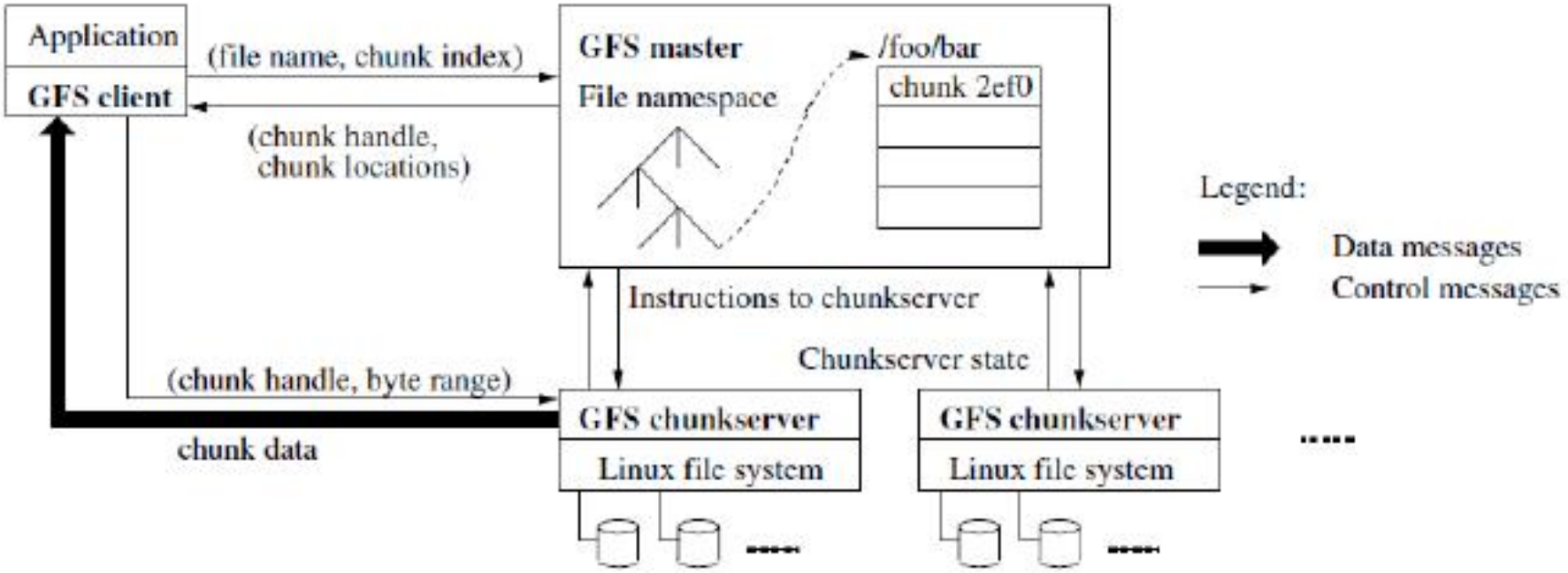
Shared Data Solution - 1

- Communication between data accessors is mediated by a shared data store.
- Data is made persistent by the data store.
- **Elements**:
  - *Shared-data store.* Concerns include types of data stored, data distribution, and number of accessors permitted.
  - *Data accessor*
  - *Data reading and writing connector*.

Shared Data Solution - 2

- **Constraints**: Data accessors interact only with the data store(s).
- **Weaknesses**:
  - The shared-data store may be a performance bottleneck.
  - The shared-data store may be a single point of failure.
  - Producers and consumers of data may be tightly coupled. E.g., concurrency control is needed in database

Google File System (GFS)



The Google File System. SOSP 2003

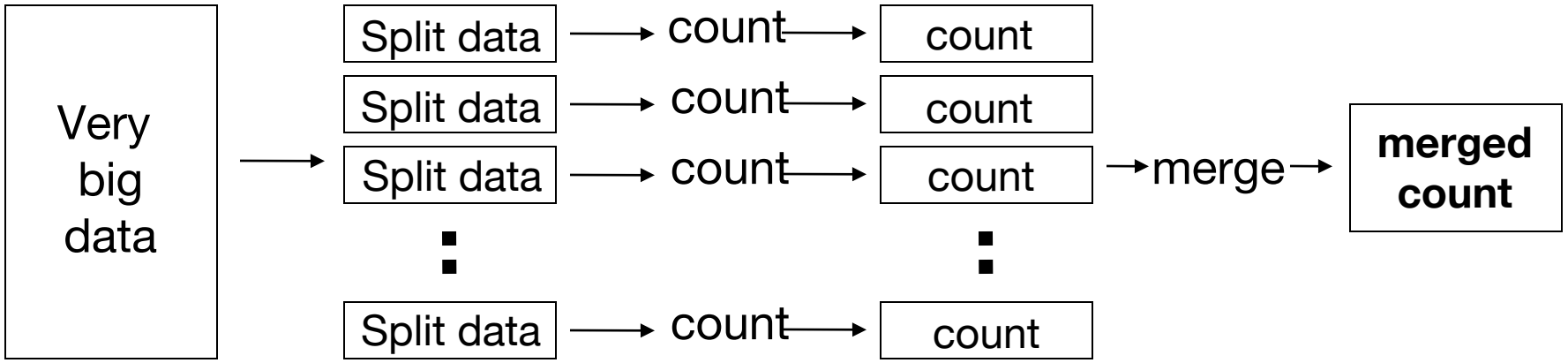
Architectural Patterns

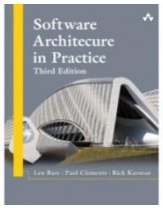
- **Module patterns**
  - Layered pattern
- **Component-and-Connector patterns**
  - Broker pattern
  - Model-View-Controller pattern
  - Pipe-and-Filter pattern
  - Client-Server pattern
  - Peer-to-Peer pattern
  - Service-Oriented Architecture (SOA) pattern
  - Publish-Subscribe pattern
  - Shared-Data pattern
- **Allocation patterns**
  - Map-Reduce pattern
  - Multi-tier Pattern

MapReduce

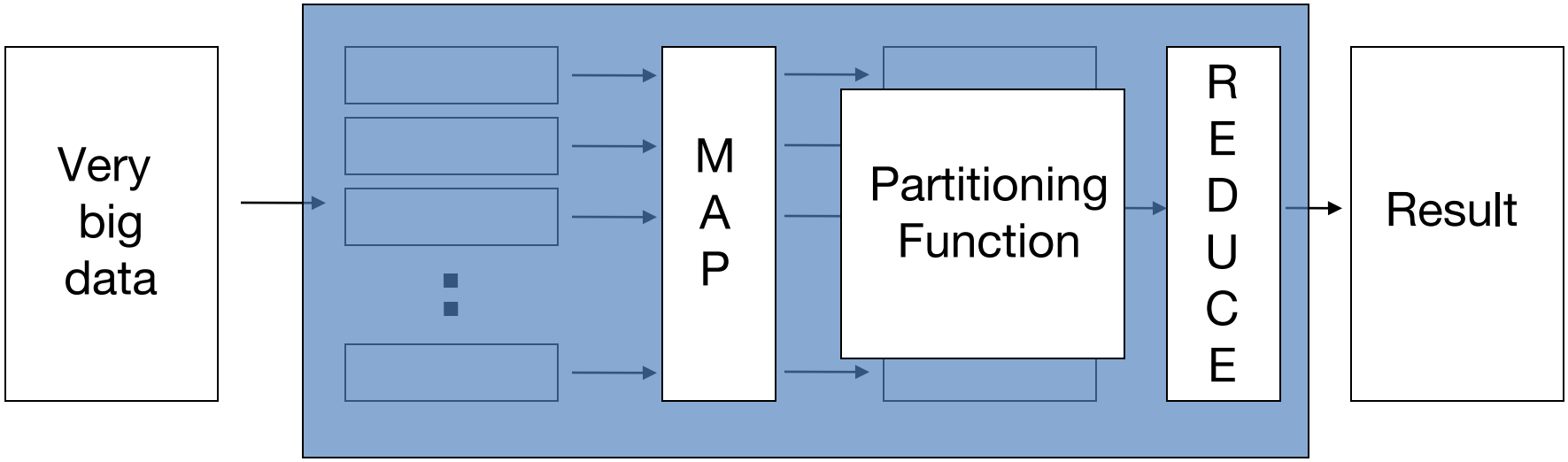
- Origin from Google, [OSDI'04]
- A simple programming model
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in distributed manner
  - Offers high availability

Distributed Word Count

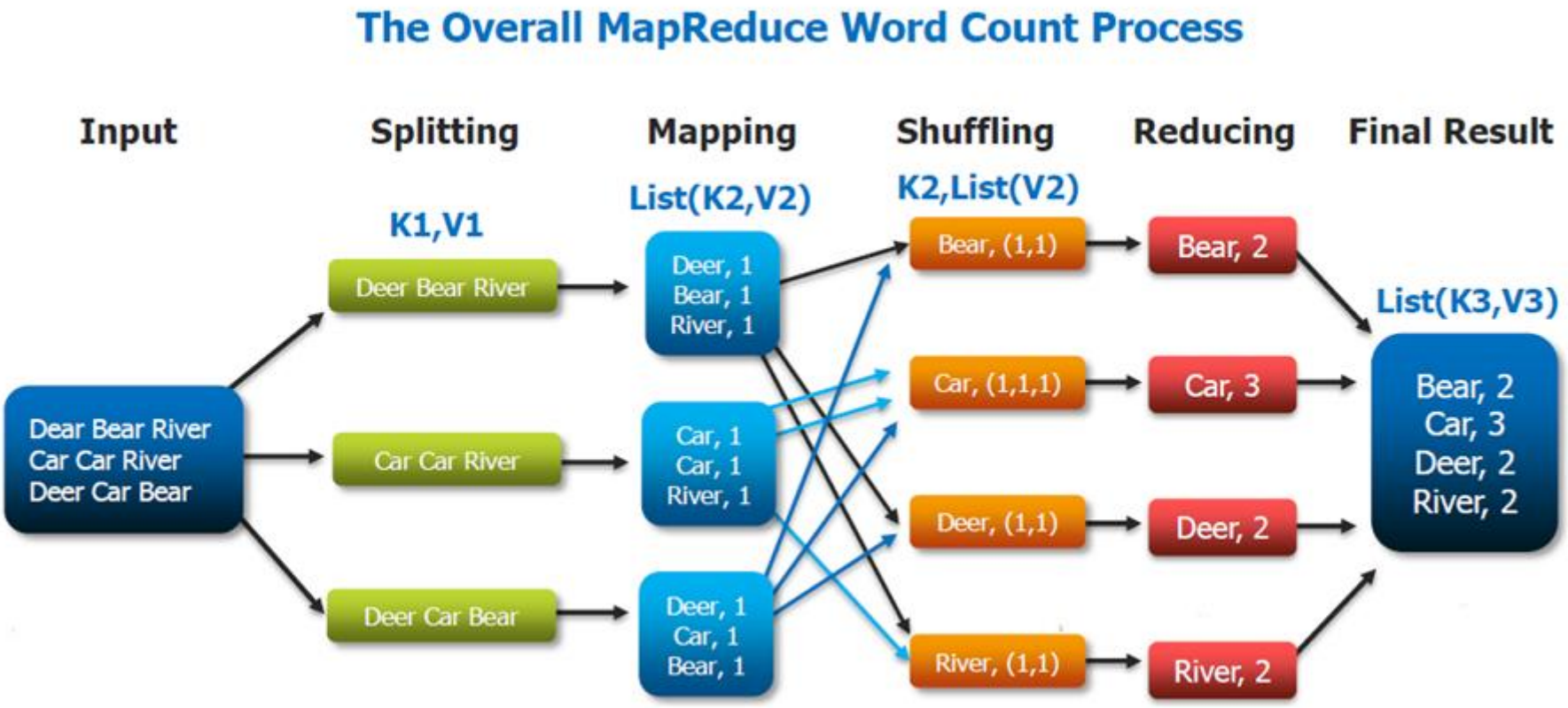




Map+Reduce



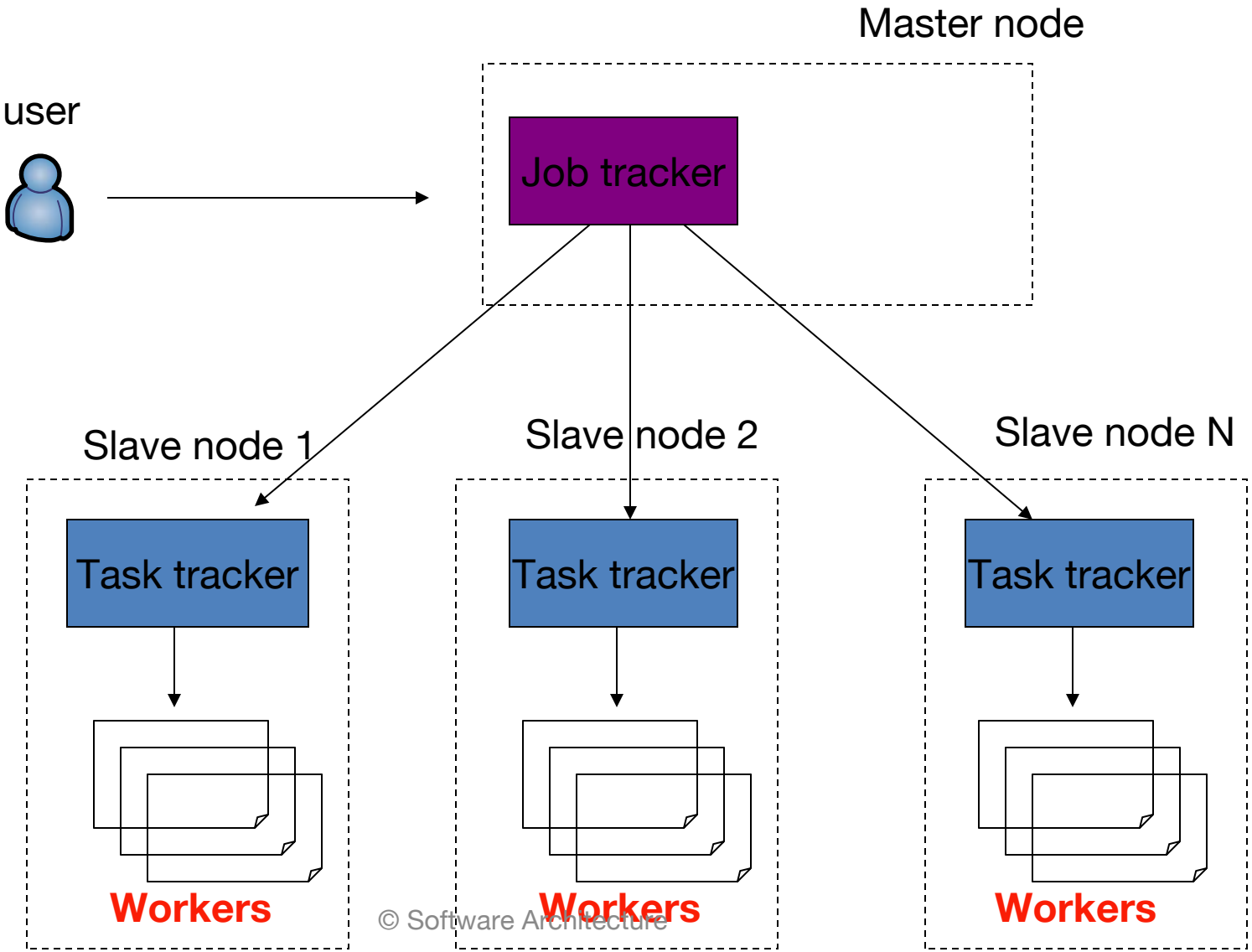
- Map:
  - Accepts *input* key/value pair
  - Emits *intermediate* key/value pair
- Reduce :
  - Accepts *intermediate* key/value* pair
  - Emits *output* key/value pair



Functions in the Model

- Map
  - Process a key/value pair to generate intermediate key/value pairs
- Reduce
  - Merge all intermediate values associated with the same key
- Partition
  - By default : `hash(key) mod R`
  - Well balanced

Architecture overview



A Simple Example

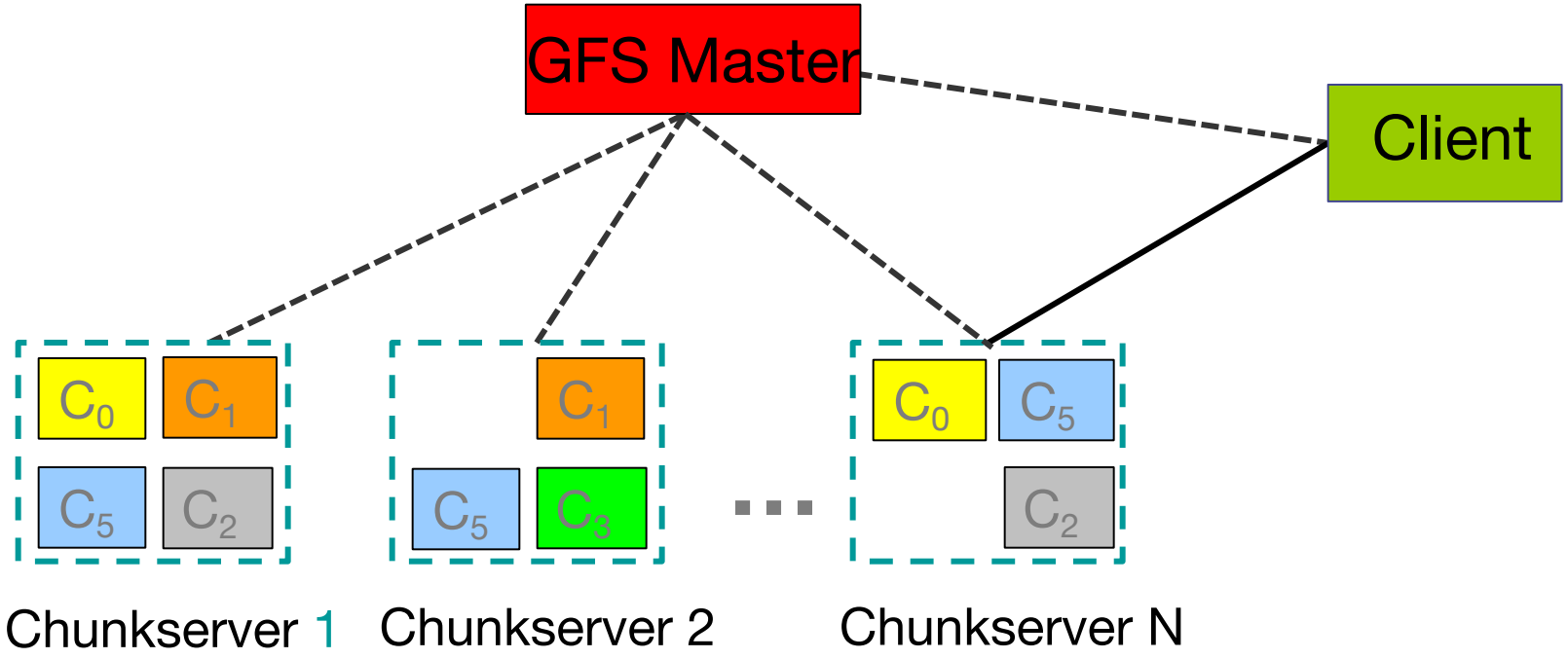
- Counting words in a large set of documents

```
map(string value)
        //key: document name
        //value: document contents
        for each word w in value
                EmitIntermediate(w, "1");

reduce(string key, iterator values)!
        //key: word
        //values: list of counts
        int results = 0;
        for each v in values
                result += ParseInt(v);
        Emit(AsString(result));
```

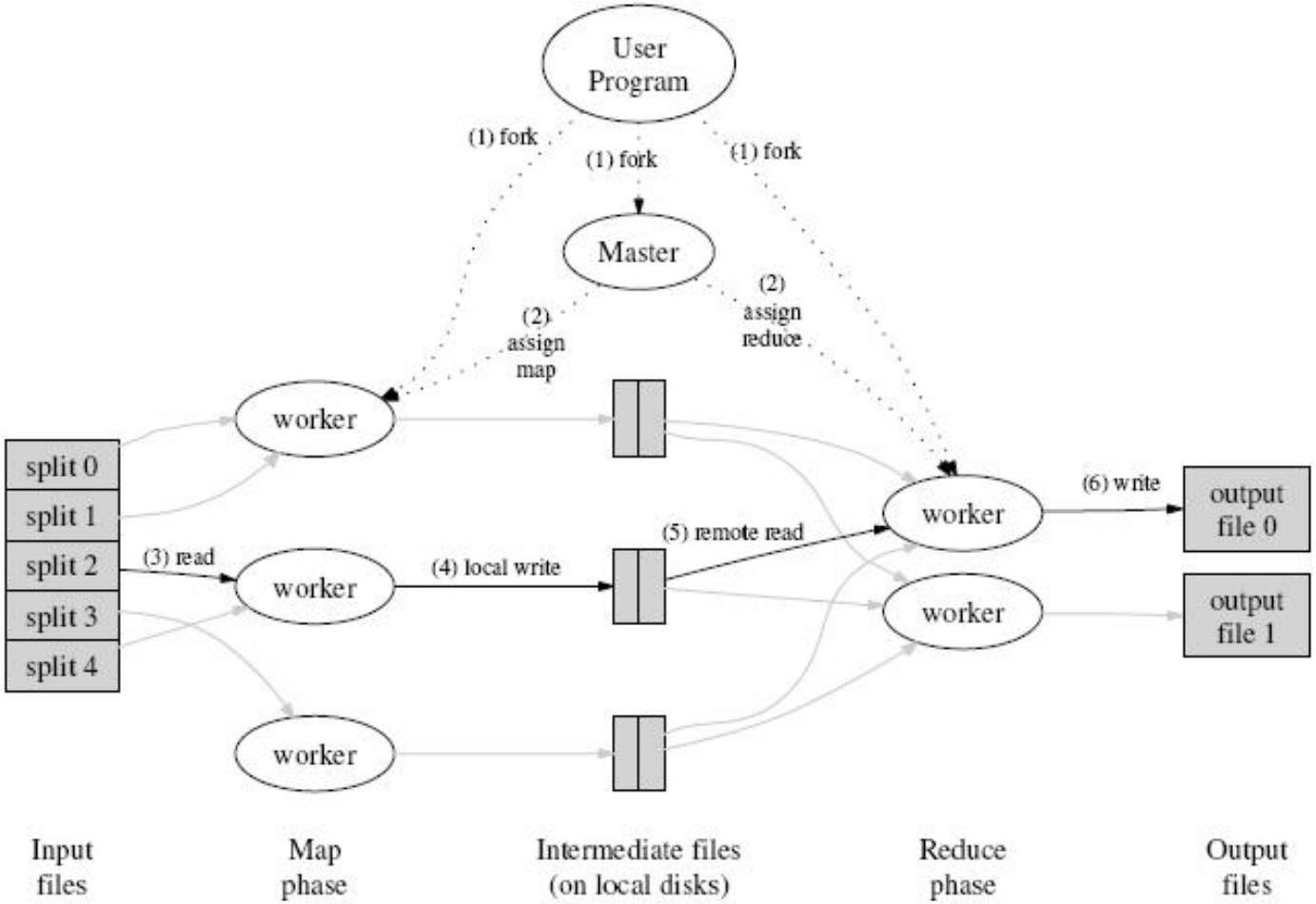
GFS: underlying storage system

- Goal
  - global view
  - make huge files available in the face of node failures
- Master Node (meta server)
  - Centralized, index all chunks on data servers
- Chunk server (data server)
  - File is split into contiguous chunks, typically 16-64MB.
  - Each chunk replicated (usually 2x or 3x).
    - Try to keep replicas in different racks.

GFS architecture



How does it work?



Locality issue

- Master scheduling policy
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (== GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate

Fault Tolerance

- Reactive way
  - Worker failure
    - Heartbeat, Workers are periodically pinged by master
      - NO response = failed worker
    - If the processor of a worker fails, the tasks of that worker are reassigned to another worker.
  - Master failure
    - Master writes periodic checkpoints
    - Another master can be started from the last checkpointed state
    - If eventually the master dies, the job will be aborted

Fault Tolerance

- Proactive way (**Redundant Execution**)
  - The problem of “stragglers” (slow workers)
    - Other jobs consuming resources on machine
    - Bad disks transfer data very slowly
    - Weird things: processor caches disabled (!!)
  - When computation almost done, reschedule in-progress tasks
  - Whenever either the primary or the backup executions finishes, mark it as completed

Points need to be emphasized

- No *reduce* can begin until *map* is complete
- Master must communicate locations of intermediate files
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely rerun
- MapReduce library does most of the hard work for us!

How to use it

- User to do list:
  - indicate:
    - Input/output files
    - **M**: number of map tasks
    - **R**: number of reduce tasks
    - **W**: number of machines
  - Write *map* and *reduce* functions
  - Submit the job

Detailed Example: Word Count(1)

- Map

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;

      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);
```

Detailed Example: Word Count(2)

- Reduce

```
// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);
```

Detailed Example: Word Count(3)

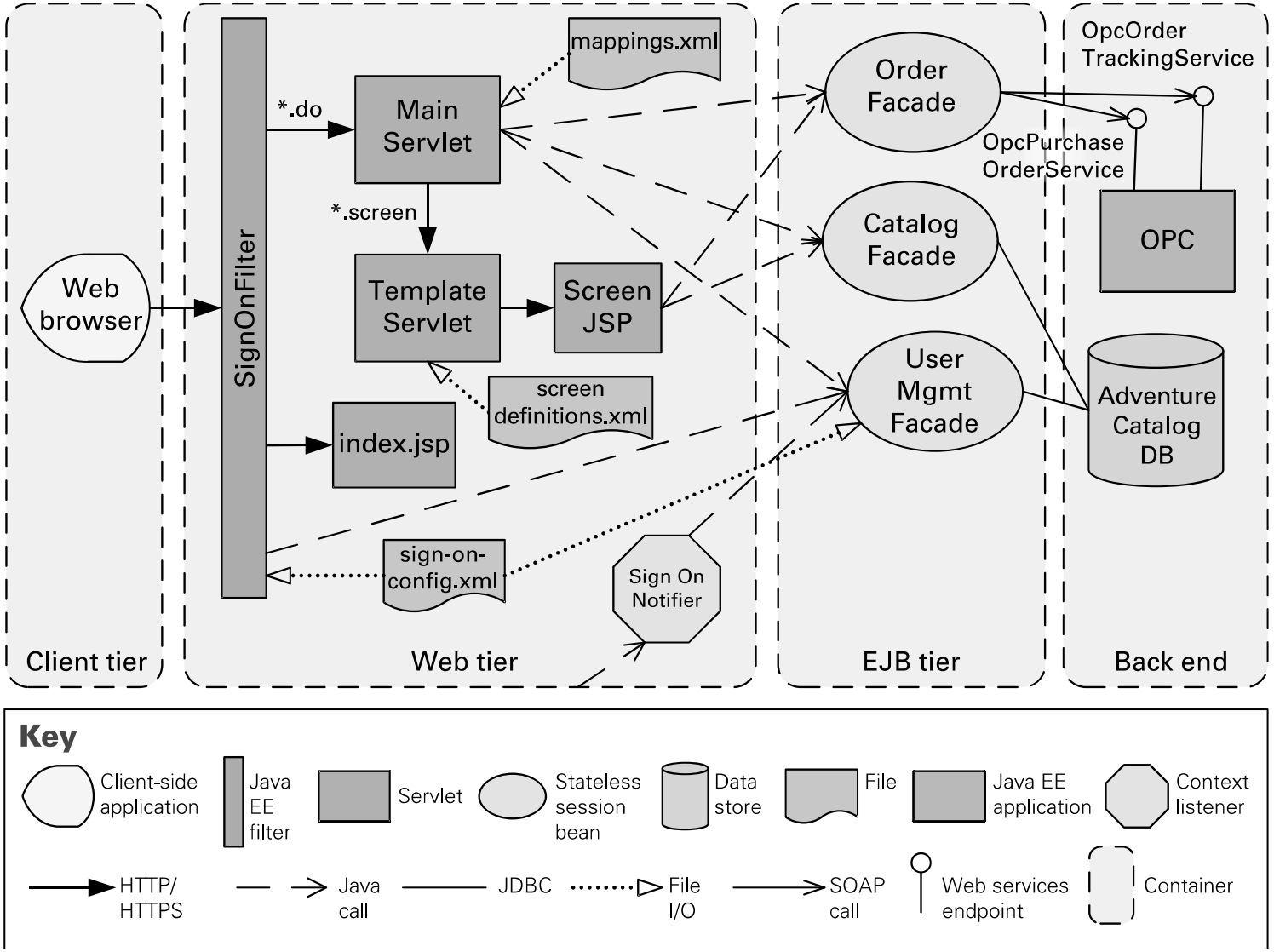
- Main

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

Multi-Tier Pattern

- The execution structures of many systems are organized as a set of logical groupings of *software and hardware*.
- Each grouping is termed a tier.
- Layer v.s. tier ?

Multi-Tier Example



Relationships Between Tactics and Patterns

- Patterns are built from tactics; if a pattern is a molecule, a tactic is an atom.
- MVC, for example utilizes the tactics:
  - Increase semantic coherence
  - Encapsulation
  - Use an intermediary

Tactics Augment Patterns

- Patterns solve a specific problem but may have weaknesses with respect to other qualities.
- Consider the broker pattern
  - May have performance bottlenecks
  - May have a single point of failure
- Using tactics such as
  - Increase resources will help performance
  - Maintain multiple copies will help availability

Tactics and Interactions

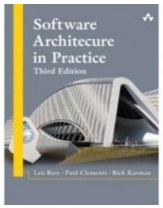
- Each tactic has pluses (its reason for being) and minuses – side effects.
- Use of tactics can help alleviate the minuses.
- But nothing is free…

Tactics and Interactions - 2

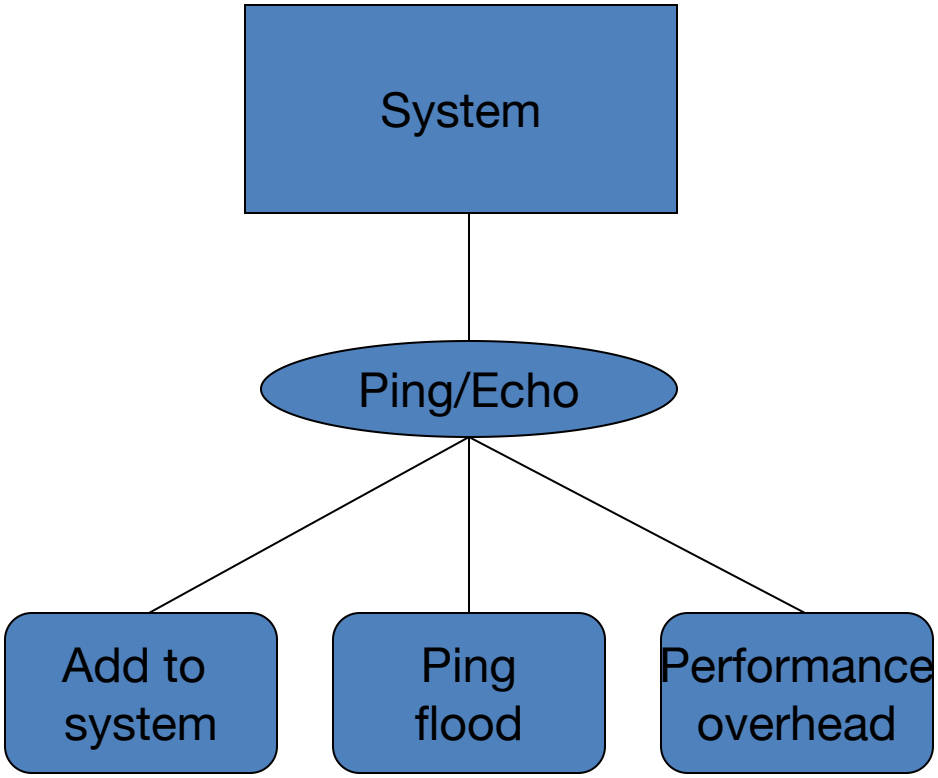
A common tactic for detecting faults is Ping/Echo.

Common side-effects of Ping/Echo are:

- security: how to prevent a ping flood attack?
- performance: how to ensure that the performance overhead of ping/echo is small?
- modifiability: how to add ping/echo to the existing architecture?



Tactics and Interactions - 3

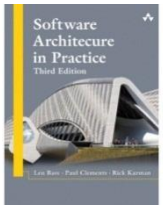


Tactics and Interactions - 4

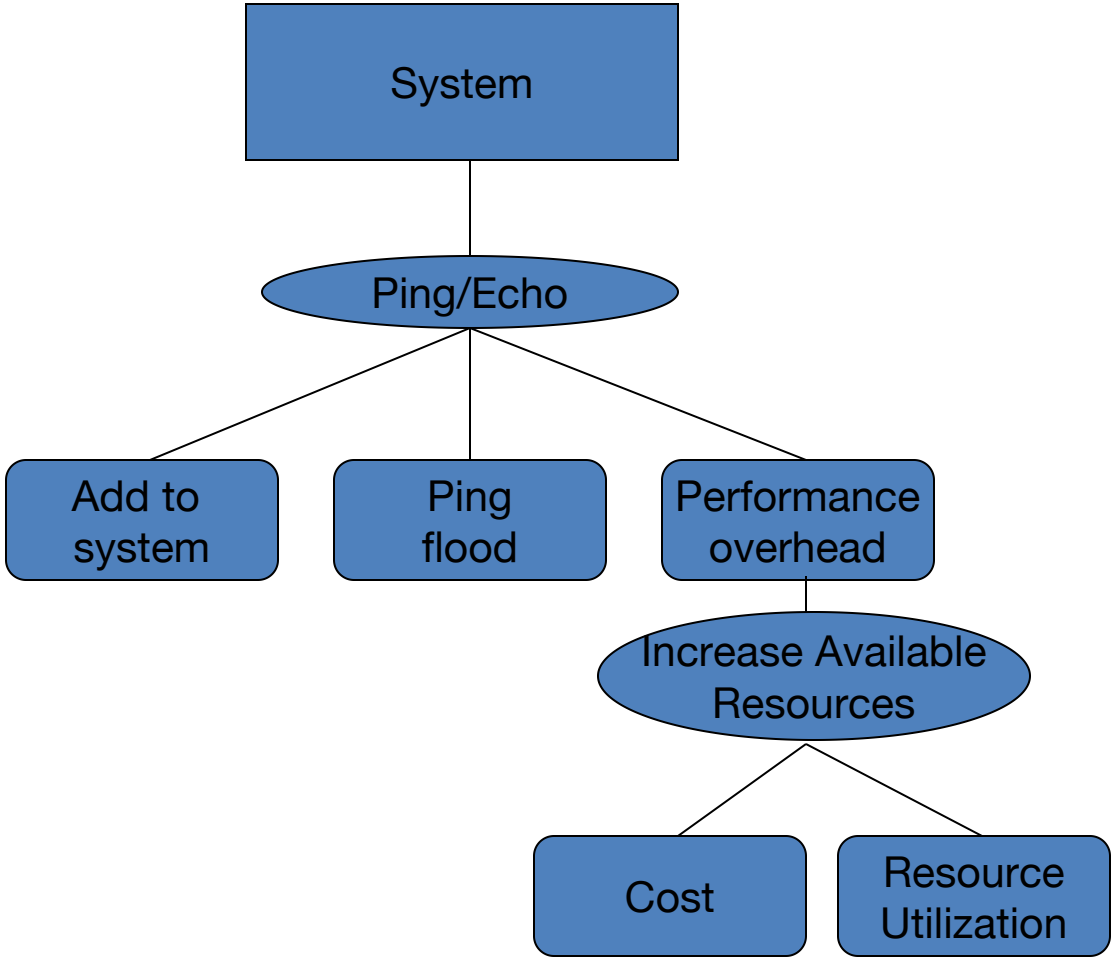
A tactic to address the performance side-effect is “Increase Available Resources”.

Common side effects of Increase Available Resources are:

- cost: increased resources cost more
- performance: how to utilize the increase resources efficiently?



Tactics and Interactions - 5

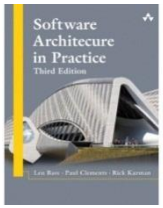


Tactics and Interactions - 6

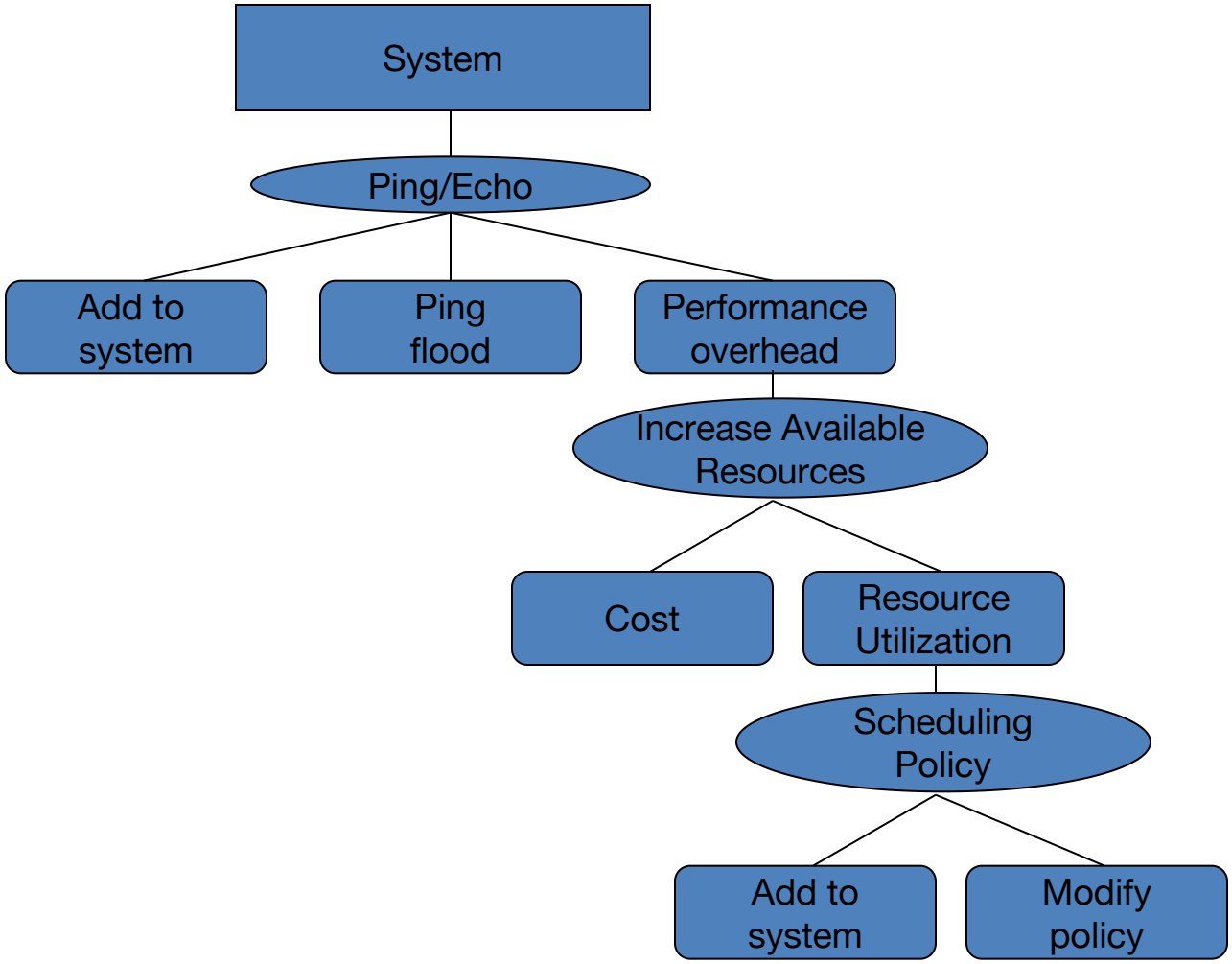
A tactic to address the efficient use of resources side-effect is “Scheduling Policy”.

Common side effects of Scheduling Policy are:

- modifiability: how to add the scheduling policy to the existing architecture
- modifiability: how to change the scheduling policy in the future?



Tactics and Interactions - 7



Tactics and Interactions - 8

A tactic to address the addition of the scheduler to the system is “Use an Intermediary”.

Common side effects of Use an Intermediary are:

- modifiability: how to ensure that all communication passes through the intermediary?

How Does This Process End?

- Each use of tactic introduces new concerns.
- Each new concern causes new tactics to be added.
- Are we in an infinite progression?
- No. Eventually the side-effects of each tactic become small enough to ignore.

Summary

- An architectural pattern
  - is a package of design decisions that is found repeatedly in practice,
  - has known properties that permit reuse, and
  - describes a *class* of architectures.
- Tactics are simpler than patterns
- Patterns are underspecified with respect to real systems so they have to be augmented with tactics.
  - Augmentation ends when requirements for a specific system are satisfied.

Chapter 14 Quality Attribute Modeling and Analysis

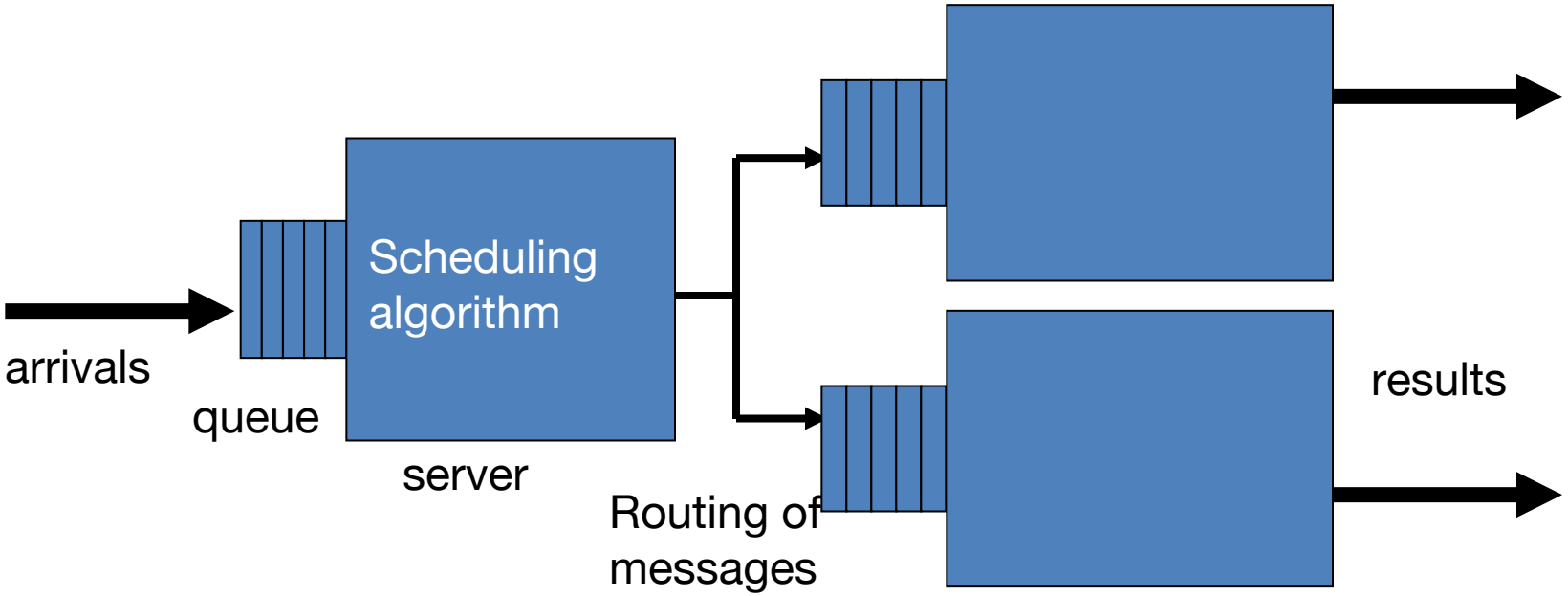
How do we know the quality attributes of a software?

- Experiment and measurement
- Analytic model
- Simulations

Modeling Architectures to Enable Quality Attribute Analysis

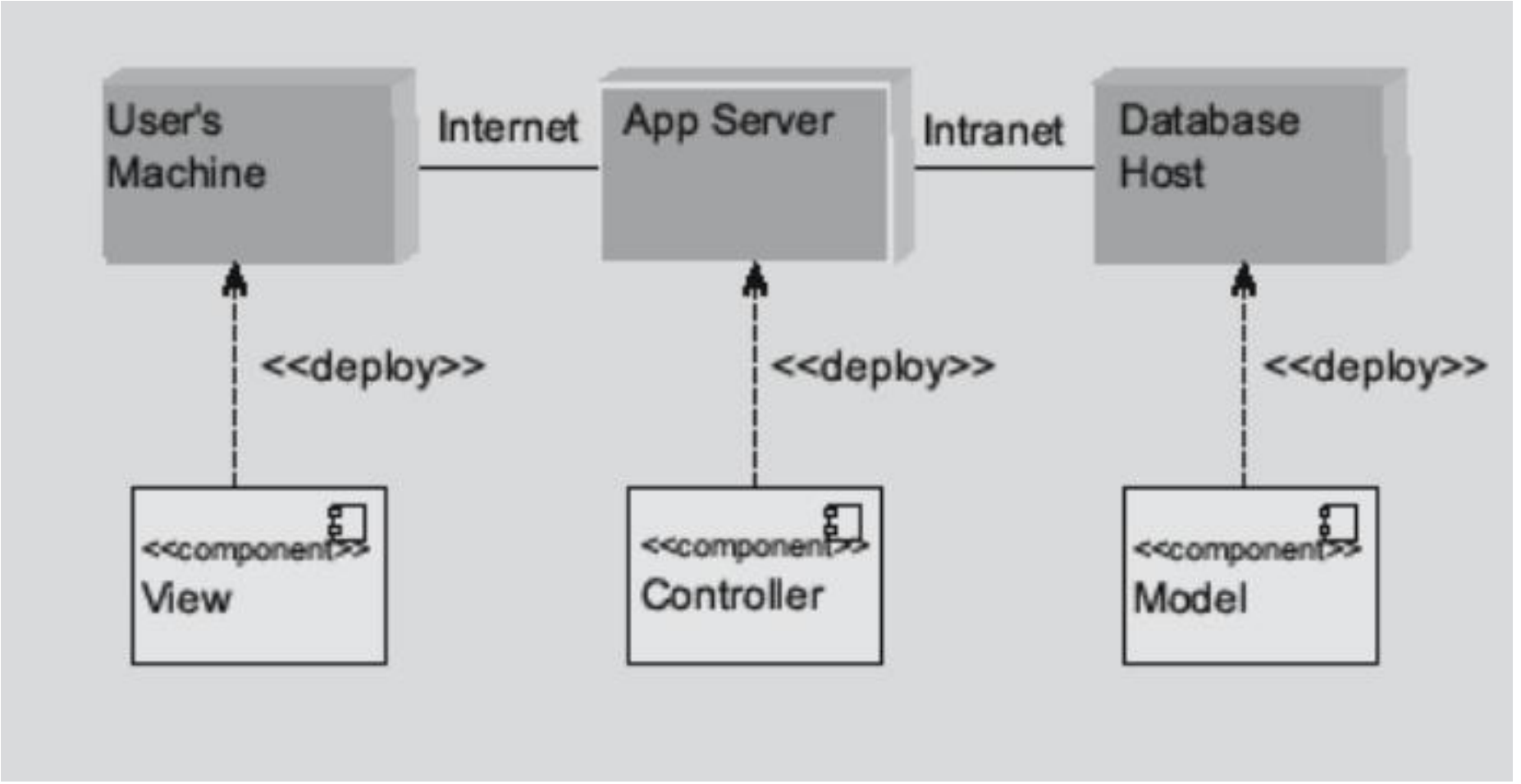
- Some quality attributes, most notably performance and availability, have well-understood, time-tested *analytic models* that can be used to assist in an analysis.
- By *analytic model*, we mean one that supports quantitative analysis.

Performance Models

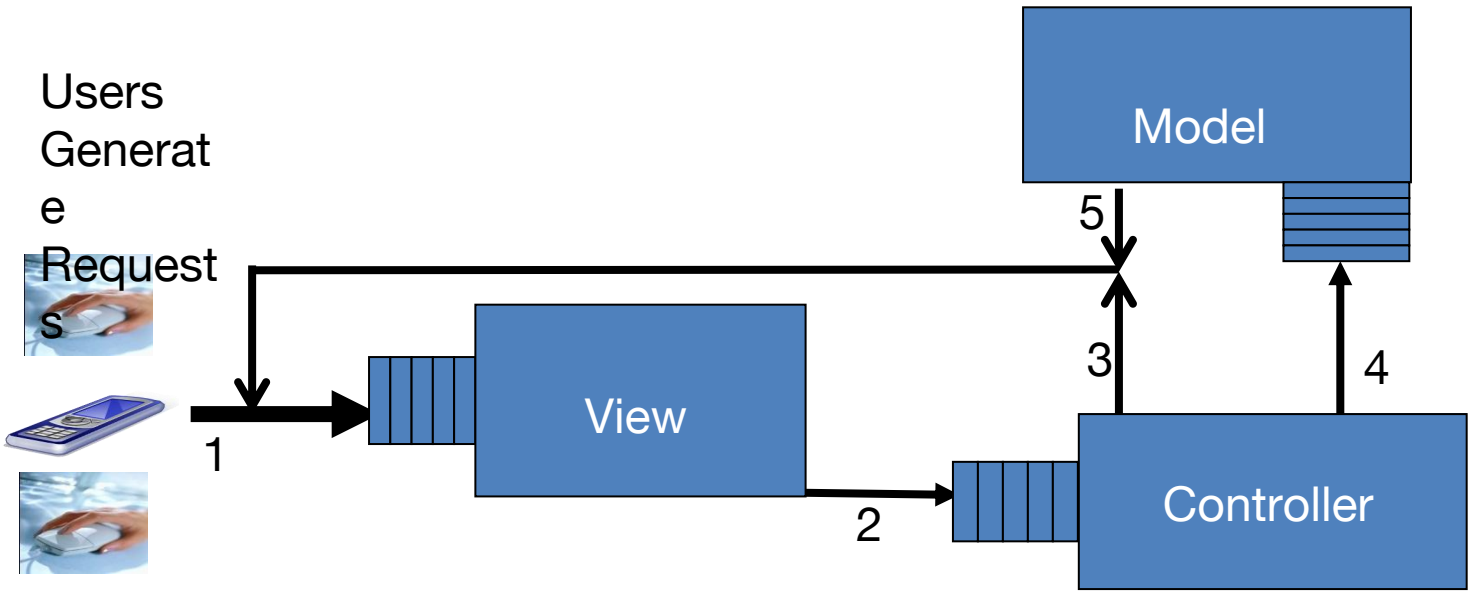


- **Parameters**: arrival rate of events, chosen queuing discipline, chosen scheduling algorithm, network topology, network bandwidth, routing algorithm chosen, service time for events

Allocation Model for MVC



Queuing Model for MVC



1. Arrivals
2. View sends requests to Controller
3. Actions returned to View
4. Actions returned to model
5. Model sends actions to View

Parameters

- To solve a queuing model for MVC performance, the following parameters must be known or estimated:
  - The frequency of arrivals from outside the system
  - The queuing discipline used at the view queue
  - The time to process a message within the view
  - The number and size of messages that the view sends to the controller
  - The bandwidth of the network that connects the view and the controller
  - The queuing discipline used by the controller
  - The time to process a message within the controller
  - The number and size of messages that the controller sends back to the view
  - The bandwidth of the network from the controller to the view
  - The number and size of messages that the controller sends to the model
  - The queuing discipline used by the model
  - The time to process a message within the model
  - The number and size of messages the model sends to the view
  - The bandwidth of the network connecting the model and the view

Cost/benefit of Performance Modeling

- Cost: determining the parameters previously mentioned
- Benefit: estimate of the latency
- The more accurately the parameters can be estimated, the better the predication of latency.
- This is worthwhile when latency is important and questionable.
- This is not worthwhile when it is obvious there is sufficient capacity to satisfy the demand.

Availability Modeling

- Another quality attribute with a well-understood analytic framework is availability.
- Modeling availability is to determine the failure rates and the recovery times of the components.
- **Steady-State Availability**

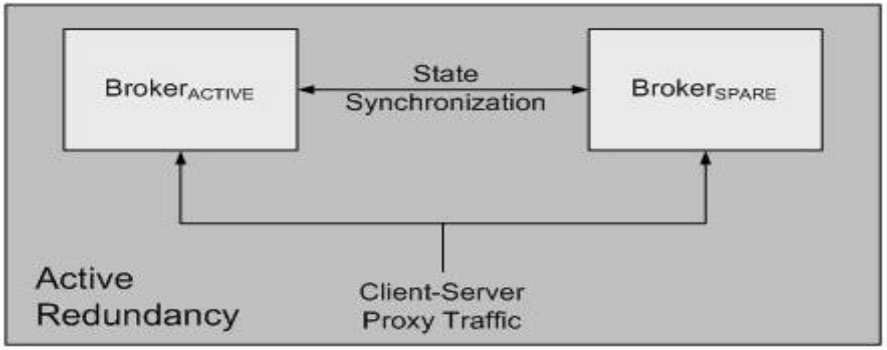
$$\frac{MTBF}{MTBF + MTTR}$$

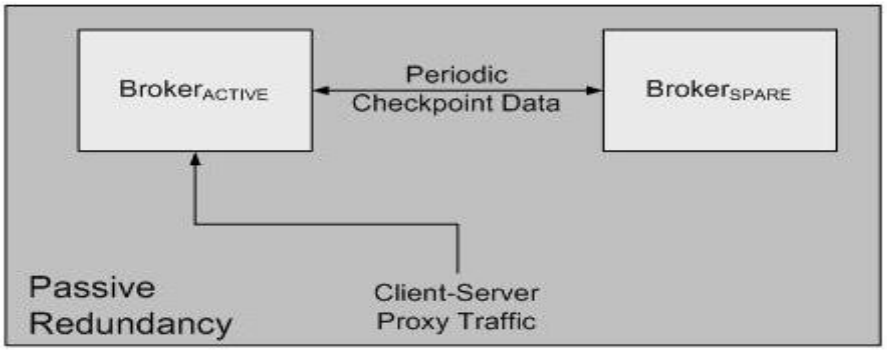
  - MTBF is the *mean time between failure*
  - MTTR refers to the *mean time to repair*

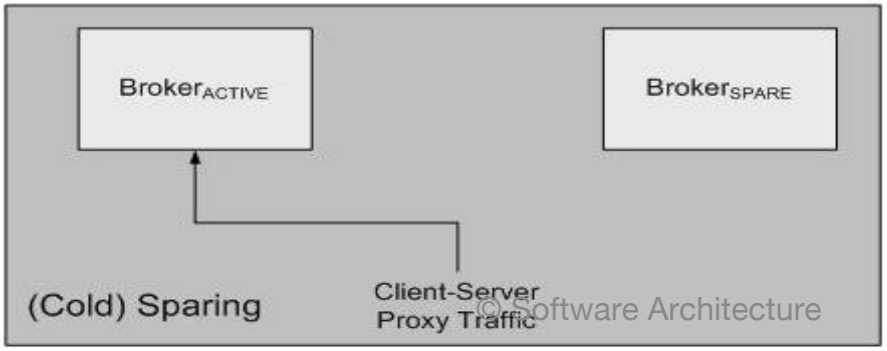
Availability Modeling

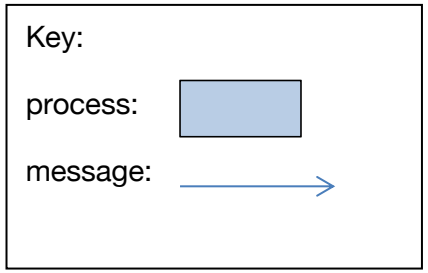
- Three different tactics for increasing the availability of the broker are:
  - active redundancy (hot spare)
  - passive redundancy (warm spare)
  - spare (cold spare).

Making Broker More Available









$$\frac{MTBF}{MTBF + MTTR}$$

Experiments and Prototypes

- Many tools can help perform experiments to determine behavior of a design
  - Request generators can create synthetic loads to test scalability
  - Monitors can perform non-intrusive resource usage detection.
- These depend on having a partial or *prototype* implementation.
  - Prototype alternatives for the most important decisions
  - If possible, implement prototype in a fashion so that some of it can be re-used.
  - Fault injection tools can induce faults to determine response of system under failure conditions.

Summary

- Analysis is always a cost/benefit activity
  - Cost is measure of creating and executing the analysis models and tools
  - Benefit depends on
    - Accuracy of analysis
    - Importance of what is being analyzed
- Analysis can be done through
  - Models for some attributes
  - Measurement
  - Simulations
  - Prototypes
  - Thought experiments

Cost-aware Service Placement and Load Dispatching in Mobile Cloud Computing

- **Modeling the performance of content delivery applications in MCC**
- **Design an offline heuristic to achieve an optimal trade-off between the access latency and service cost**
- **Develop an online algorithm to service placement and load dispatching under the mobility of users**

"Cost-aware service placement and load dispatching in mobile cloud computing", IEEE Trans. On Computers, 2016

广东省工业和信息化厅文件

粤工信信软〔2020〕73号

广东省工业和信息化厅关于印发广东省5G基站和数据中心总体布局规划（2021-2025年）的通知

各地级以上市人民政府，省政府各部门、各直属机构：

《广东省5G基站和数据中心总体布局规划（2021-2025年）》已经省人民政府同意，现印发给你们，请认真组织实施。实施过程中遇到的问题，请径向省工业和信息化厅反映。

广东省工业和信息化厅

2020年6月9日

四、数据中心规划

（一）布局依据

1. 规模分级

超大型数据中心：10000（含）个以上标准机架的数据中心。

大型数据中心：3000~10000个标准机架的数据中心。

中型数据中心：1000~3000（含）个标准机架的数据中心。

小型数据中心：少于1000（含）个标准机架的数据中心，包括通用数据中心、边缘计算数据中心。

2. 能耗指标

参照国际先进水平制定。

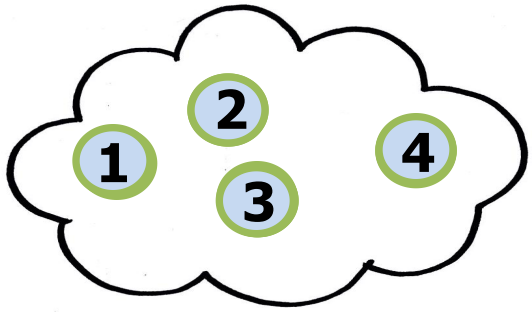
PUE≤1.25：优先支持新建和扩建。

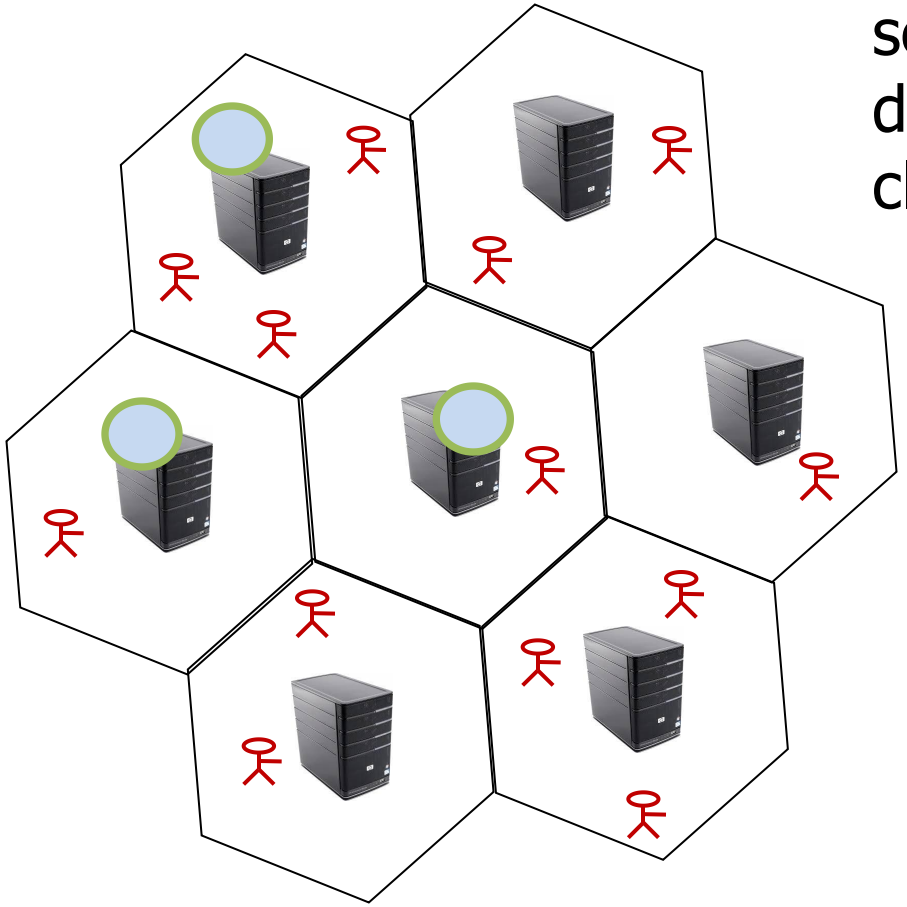
1.25＜PUE≤1.3：支持新建和扩建。

1.3＜PUE≤1.5：严控改建，不支持新建、扩建。

PUE＞1.5：禁止新建、扩建和改建。

Content Delevery





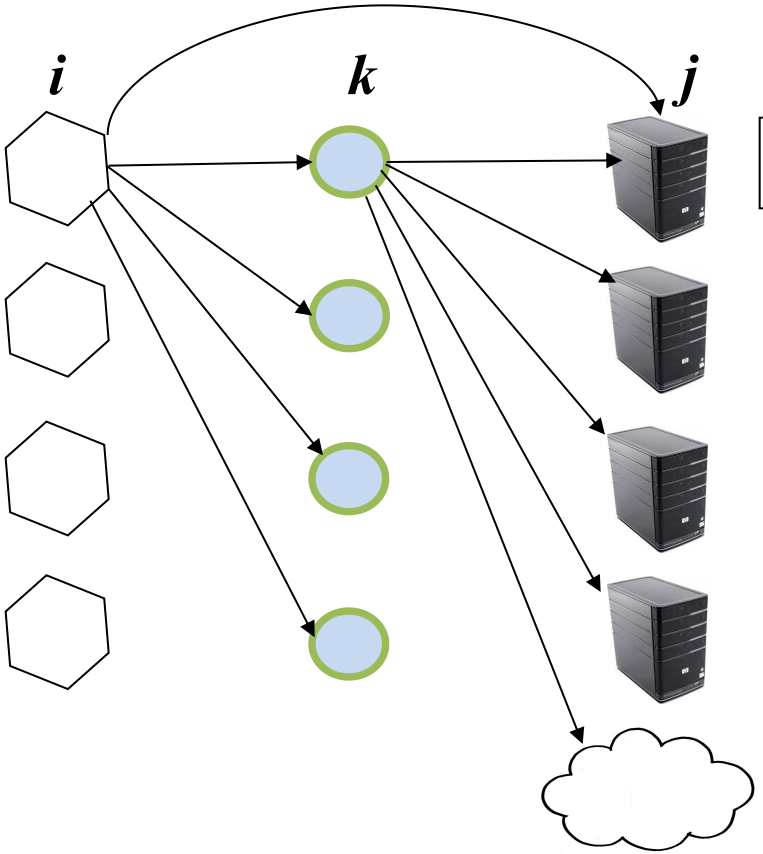
1. Joint optimization of the service placement and load dispatching in cloudlets and cloud

2. Predict the dynamic loads caused by users' mobility, and consider the cost of changing the service placement, and design an effective online solution

Two Problems

- Offline Problem
  - Basic Service Placement Problem (BSPP)
  - Load balancing
- Online Problem
  - Time-slotted Cost aware Service Placement Problem (CSPP)

Problem Definition of BSPP

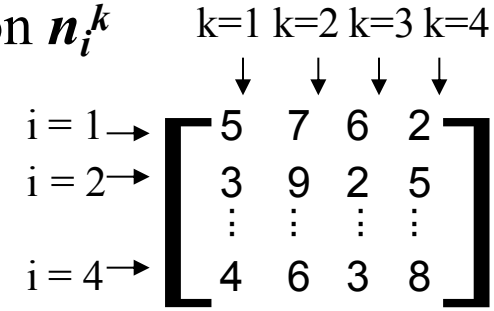


Given:

A set of K services, N regions, N cloudlets

Deploying each service k requires storage s_k

Requests distribution n_i^k



Access latency matrix d_{ij}

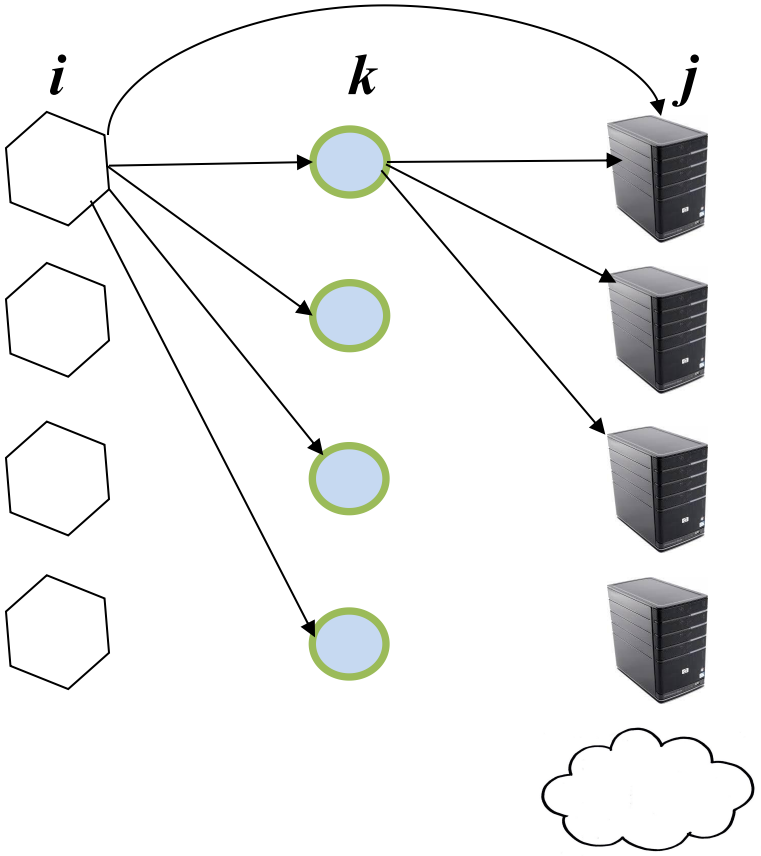
Objective: Service placement a_j^k and load dispatching β_{ij}^k to minimize the average access latency for all the loads

Constraints: Each cloudlet j has storage capacity u_j

$$\sum_j \beta_{ij}^k = 1$$

Each cloudlet j can serve at most c_j load

Problem Formulation of BSPP



Minimize:

$$\sum_{k=1}^{K}\sum_{i=1}^{N}\sum_{j=1}^{N+1}\beta_{ij}^{k}\times n_{i}^{k}\times d_{ij}$$

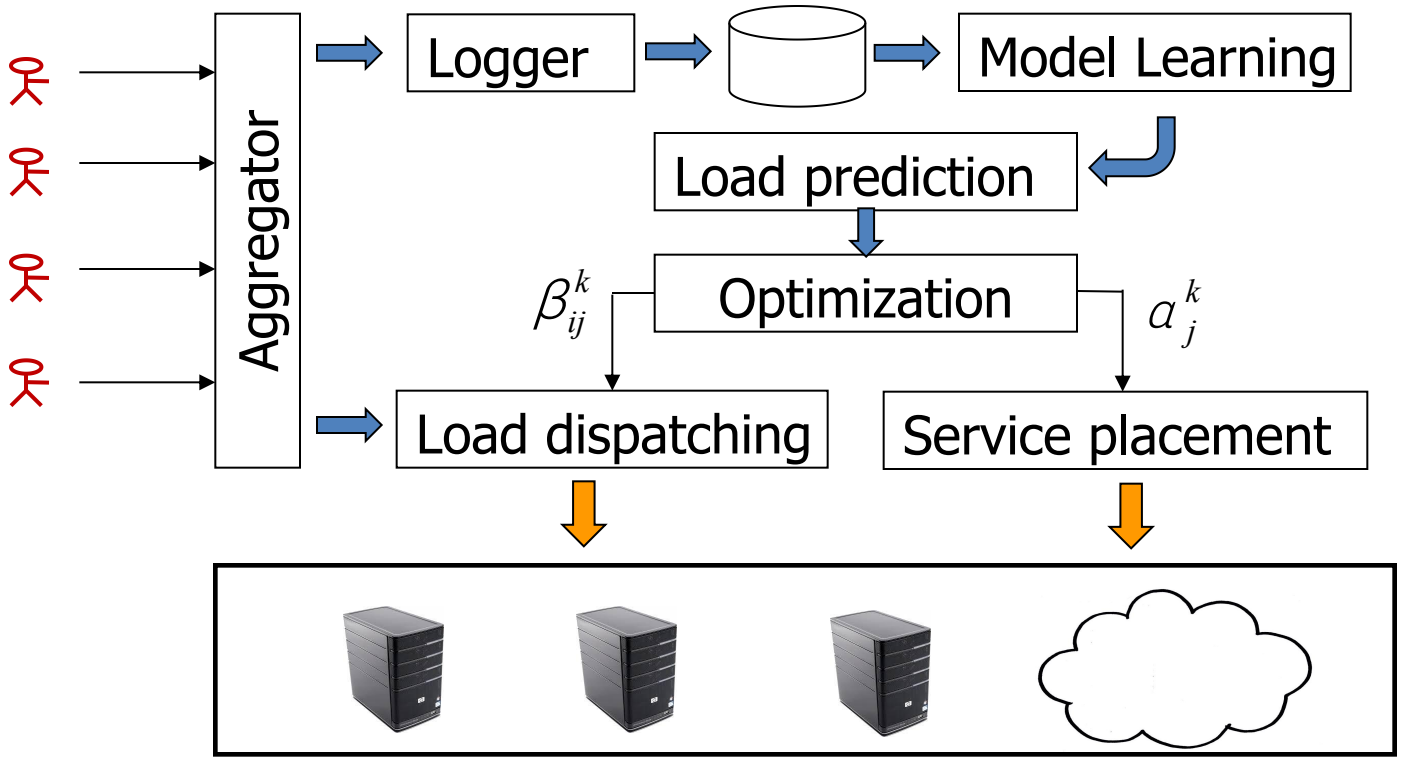
Constraints:

$$\sum_{k=1}^{K} s_k \times a_j^k \le u_j, \forall j \in [1, N]$$

$$\sum_{k=1}^{K}\sum_{i=1}^{N}\beta_{ij}^{k}\times n_{i}^{k} \le c_j, \forall j \in [1, N]$$

$$\beta_{ij}^{k} \le a_j^k, \forall j \in [1, N+1], \forall k \qquad \sum_{j}\beta_{ij}^{k} = 1$$

System Architecture



Simulations for BSPP

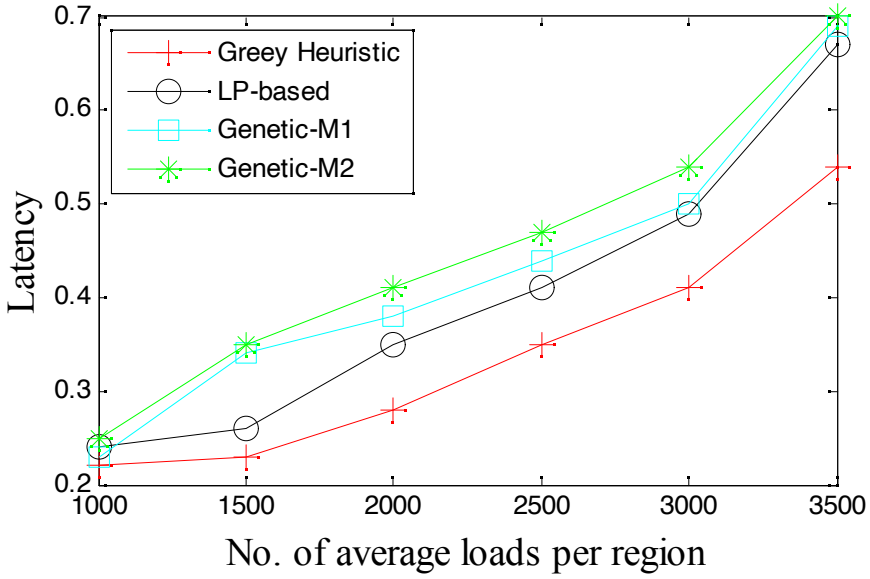
- Setup

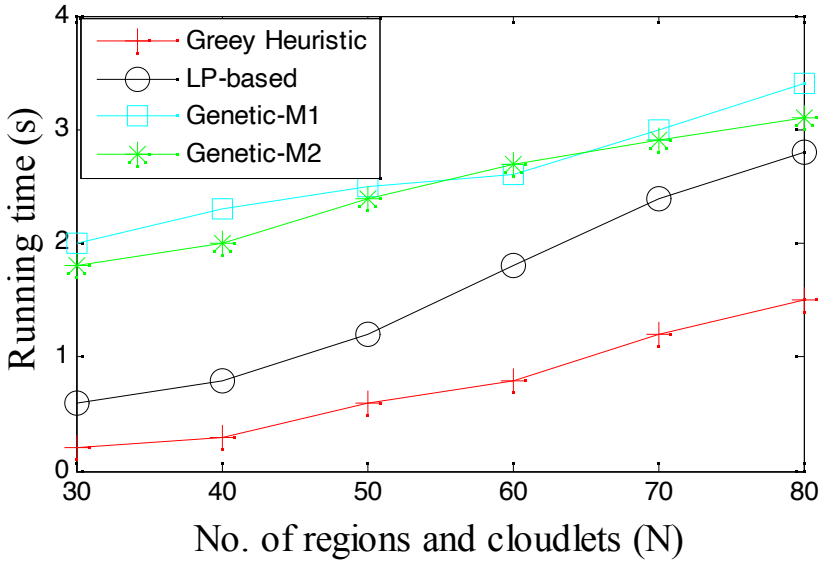
Number of regions (or cloudlets)	$N = 20$
Number of services	$K = 30$
Average data size of the services	$\frac{1}{K}\sum_k \Lambda_k = 50$
Average loads from regions	$\frac{1}{N}\sum_{i,k} n_i^k = 2000$
Average computation capacity of cloudlets	$\frac{1}{N}\sum_j c_j = 1800$
Average storage capacity of cloudlets	$\frac{1}{N}\sum_j u_j = 500$

- Metrics
  - Normalized latency: the average latency divided by the latency without caching on cloudlets
  - Algorithm running time

Simulation Results

- **Comparison with other algorithms**
  - Genetic algorithms with two different chromosome representations
  - LP-based algorithm

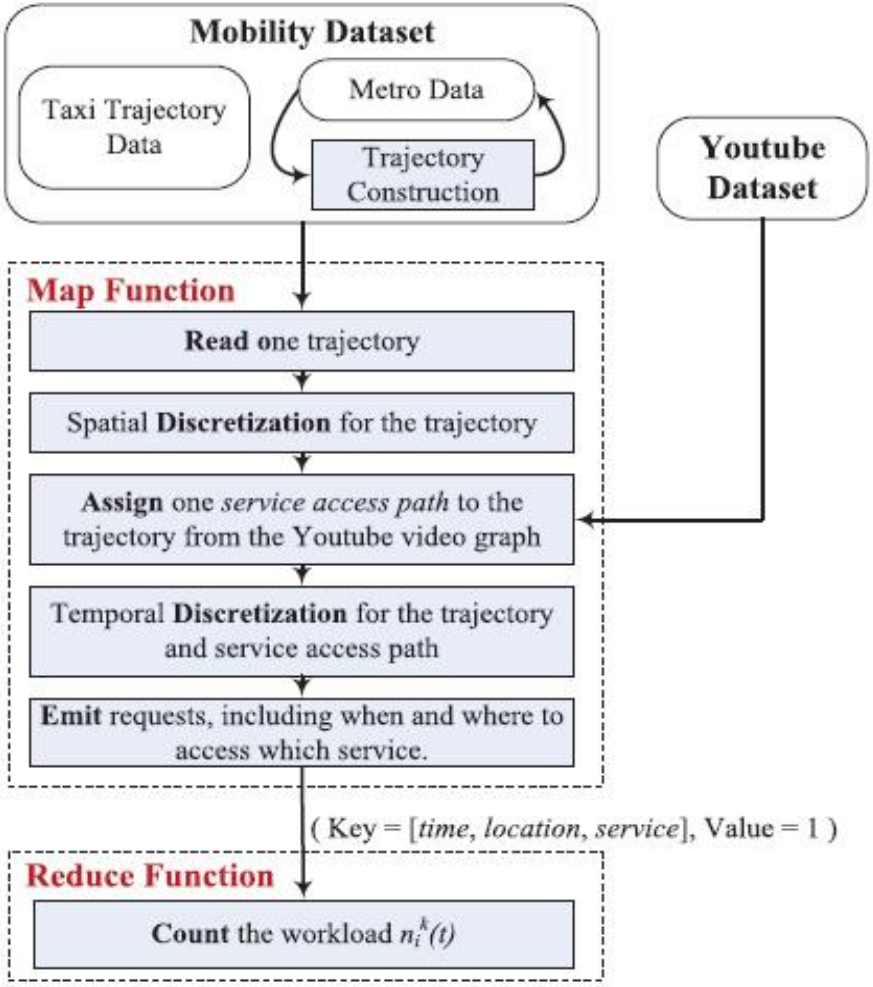




Simulation for CSPP

- Load generation

Parameters	Values
Number of time slots of the generated load trace	96
The duration of one time slot	10 minutes
Average load at each time slot	34,500
Variance of the load trace	8,630
Number of the services	20
Average data size of the services	6,710 MB
Average computation capacity of cloudlets	3,000
Average storage capacity of cloudlets	13,000 MB



Results for CSPP

- **Comparison with offline algorithm**
  - CSPP has longer latency than the offline algorithm due to the inevitable error in the load prediction.