Data Flow And Control

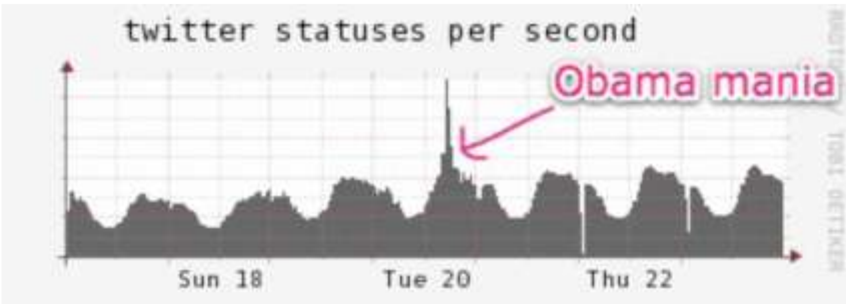
@ 北京航天航空大学软件学院

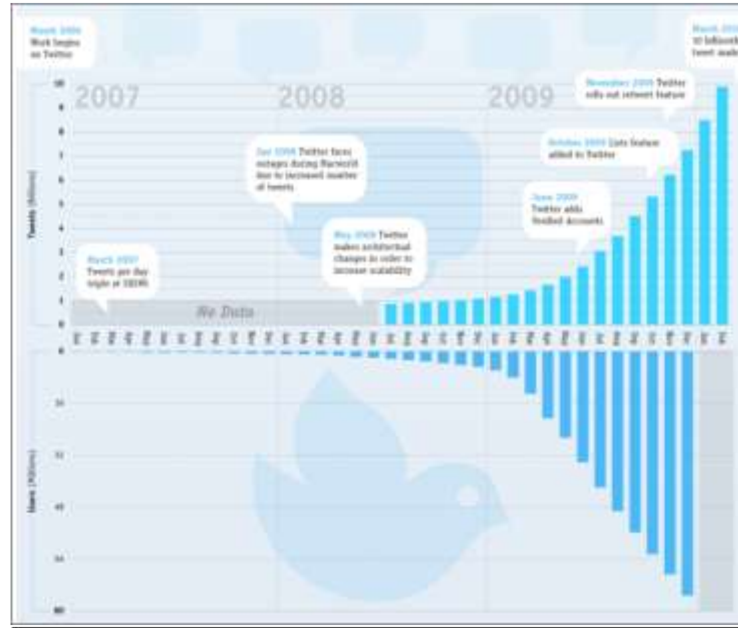
18:00 03/06/2012 北航主校区 主M201

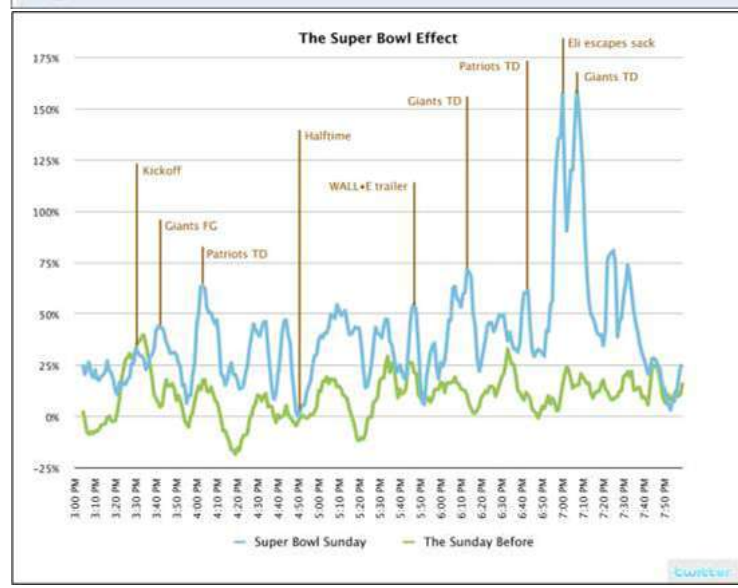
By 邓侃 Ph.D, 罗彦, 焦洋

SmartClouder.com

Anatomy of Twitter







- Big,
  Over 100 million users.
  Over 1 billion tweets.
  Growing every minutes.
- Uneven,
  Different number of followers.
  Different number of tweets at diff time.
  e.g. Inauguration of Obama, 1/20/2009,
  Peak time 350 tweets / second,
  Be forwarded over 40K times / second.


Following
Be Followed
Tweet Category
Tweet ordered by time
Tweet Reference
新浪微博 beta
首页
广场
微群
应用
游戏
每天成千上万的互联网帐号存在被盗的风险，你知道自己帐号的安全状况么，请立即查看»
有什么新鲜事想告诉大家？
983 关注
5064 粉丝
微博
我的首页
@提到我的
我的评论
我的私信
我的收藏
全部微博
我的微群
猜你喜欢
全部
相互关注
悄悄关注
北航云计算
合作伙伴
同事
更多
孙伟：我就喜欢这样的挑战文化！//@邓侃: 如果我们的云计算公开课，被砸了场子，那将是我们的荣幸，因为真正的牛人出现了！上课这事儿，就怕"我说你听"，搞一言堂。讲的人累，听的人烦。有问题随时问，举手发言亦可，递小纸条也行。没有愚蠢的问题，不问才是愚蠢的。如果你不问，那我来问，看你听懂了没。
@李开复：最近很多"中国Siri"、"安卓Siri"创业计划。我的回馈：1）智能手机主界面是手触，语音助手解决了什么真正用户需求和痛处？2）如何克服后台海量数据学习技术门槛？全球只有一个公司有这个技术。3）应用谁开发？自己开难扩张，用别人的应用整合不佳影响体验，4）语音服务器和带宽成本较大，如何克服？

ID	Following ID	Follower ID
748229	481293, 223838, …	193922, …
481293	223838, …	748229, 193922, …

Tweet ID	Time Stamp	Author ID
6793232	2012030618455245	748229
6793231	2012030618455243	481293

Tweet ID	Tweet Content
6793232	我就喜欢这样的挑战文化。
6793231	如果我们的云计算公开课，被砸了场子，那将是我们的荣幸，因为真正的牛人出现了！

Reader ID	Tweet ID in Newsfeed
193922	6793232, 6793231, …
748229	6793231, …

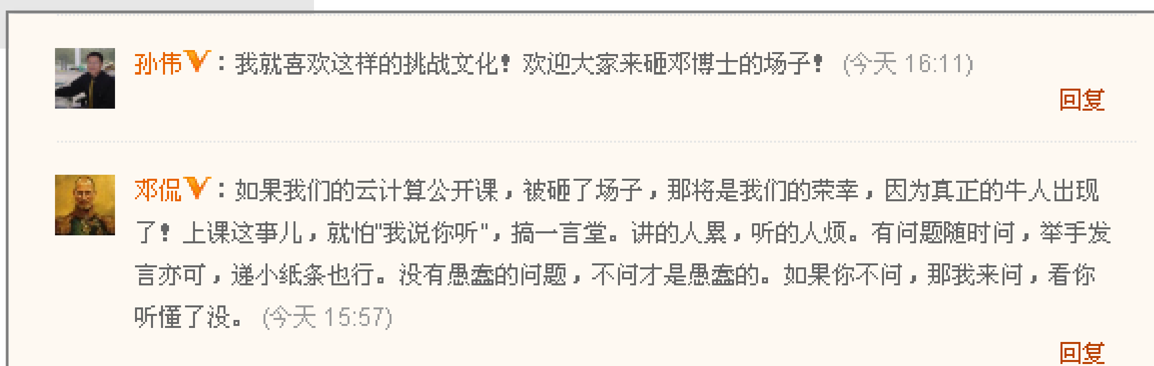
- A simple implementation of Twitter:
  Create tables to store the relationship,
  Create tables to update the tweets.
- When Mr.481293 writes a Tweet 6792321,
  "如果我们的云计算公开课 …",
  Twitter system will push the Tweet,
  into his follower's Newsfeed table.
- e.g. Tweet 6792321 is pushed into
  Mr.481293's follower,
  Mr.193922's Newsfeed.

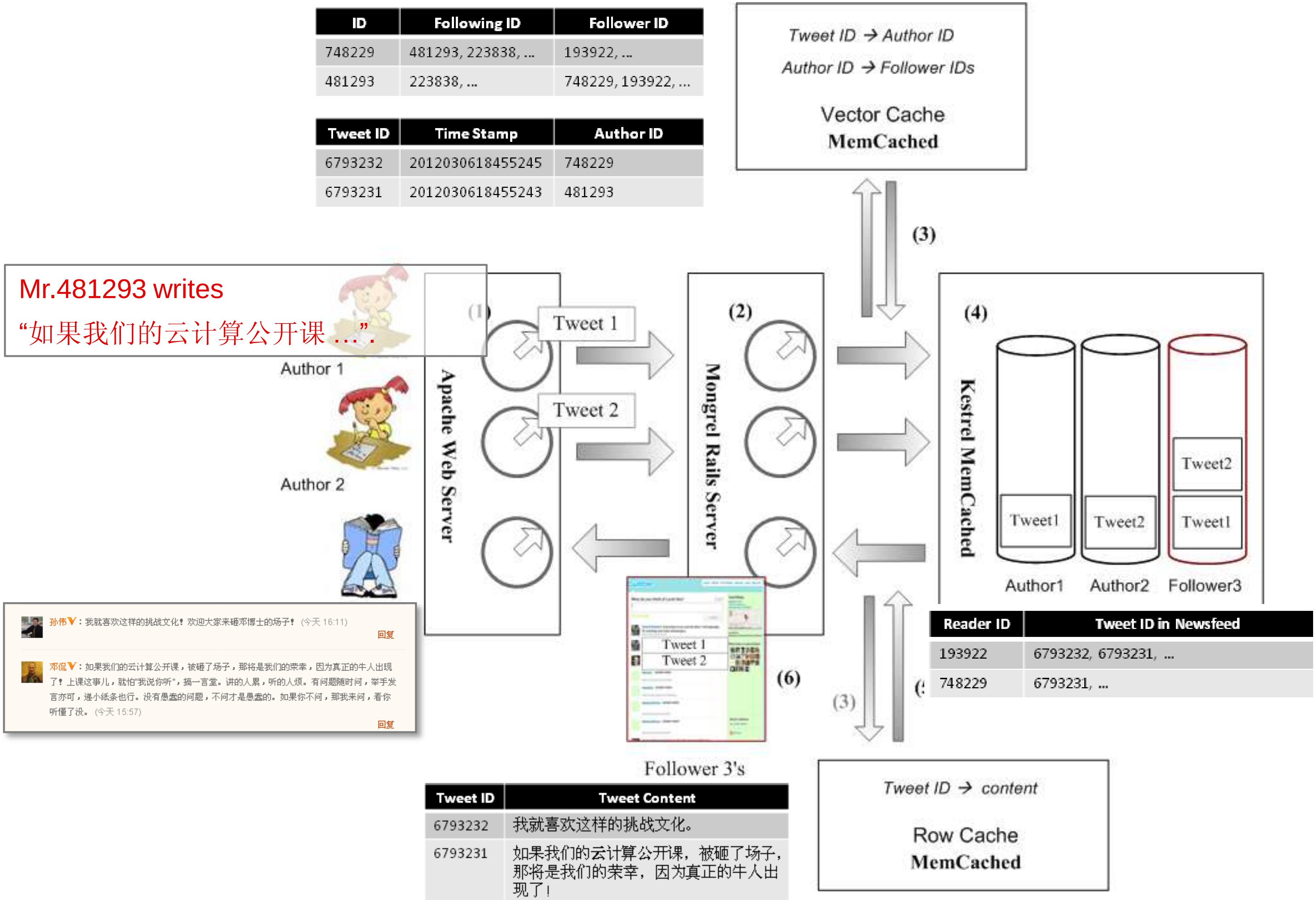
Tweet ID	Time Stamp	Author ID
6793232	2012030618455245	748229
6793231	2012030618455243	481293

Tweet ID	Tweet Content
6793232	我就喜欢这样的挑战文化。
6793231	如果我们的云计算公开课，被砸了场子，那将是我们的荣幸，因为真正的牛人出现了！

ID	Following ID	Follower ID
748229	481293, 223838, ...	193922, ...
481293	223838, ...	748229, 193922, ...

Reader ID	Tweet ID in Newsfeed
193922	6793232, 6793231, ...
748229	6793231, ...



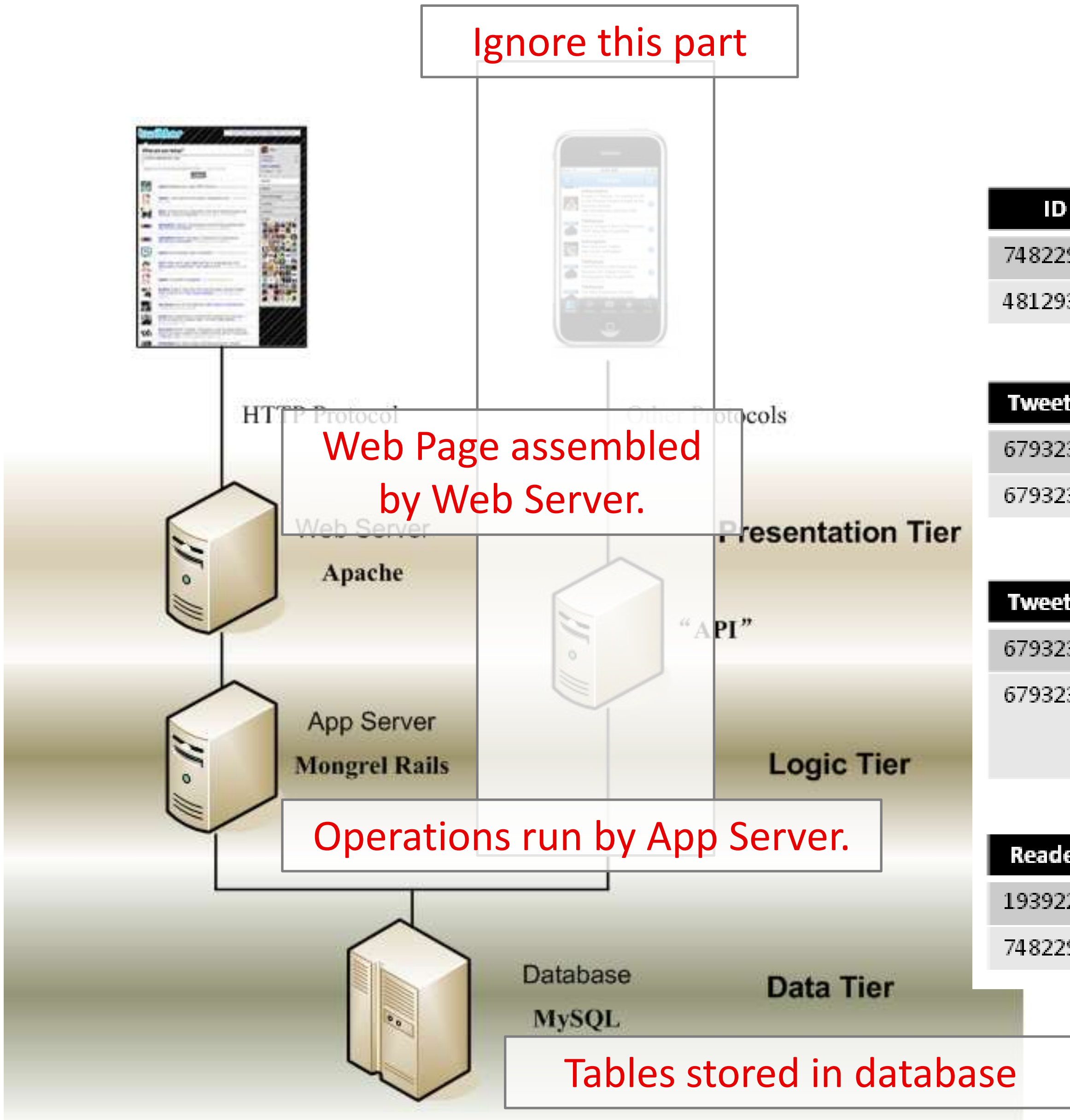


ID	Following ID	Follower ID
748229	481293, 223838, ...	193922, ...
481293	223838, ...	748229, 193922, ...

Tweet ID	Time Stamp	Author ID
6793232	2012030618455245	748229
6793231	2012030618455243	481293

Reader ID	Tweet ID in Newsfeed
193922	6793232, 6793231, ...
748229	6793231, ...

Tweet ID	Tweet Content
6793232	我就喜欢这样的挑战文化。
6793231	如果我们的云计算公开课，被砸了场子，那将是我们的荣幸，因为真正的牛人出现了！

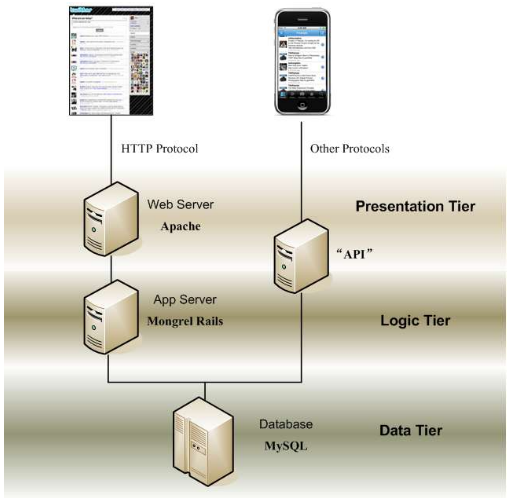


ID	Following ID	Follower ID
748229	481293, 223838, ...	193922, ...
481293	223838, ...	748229, 193922, ...

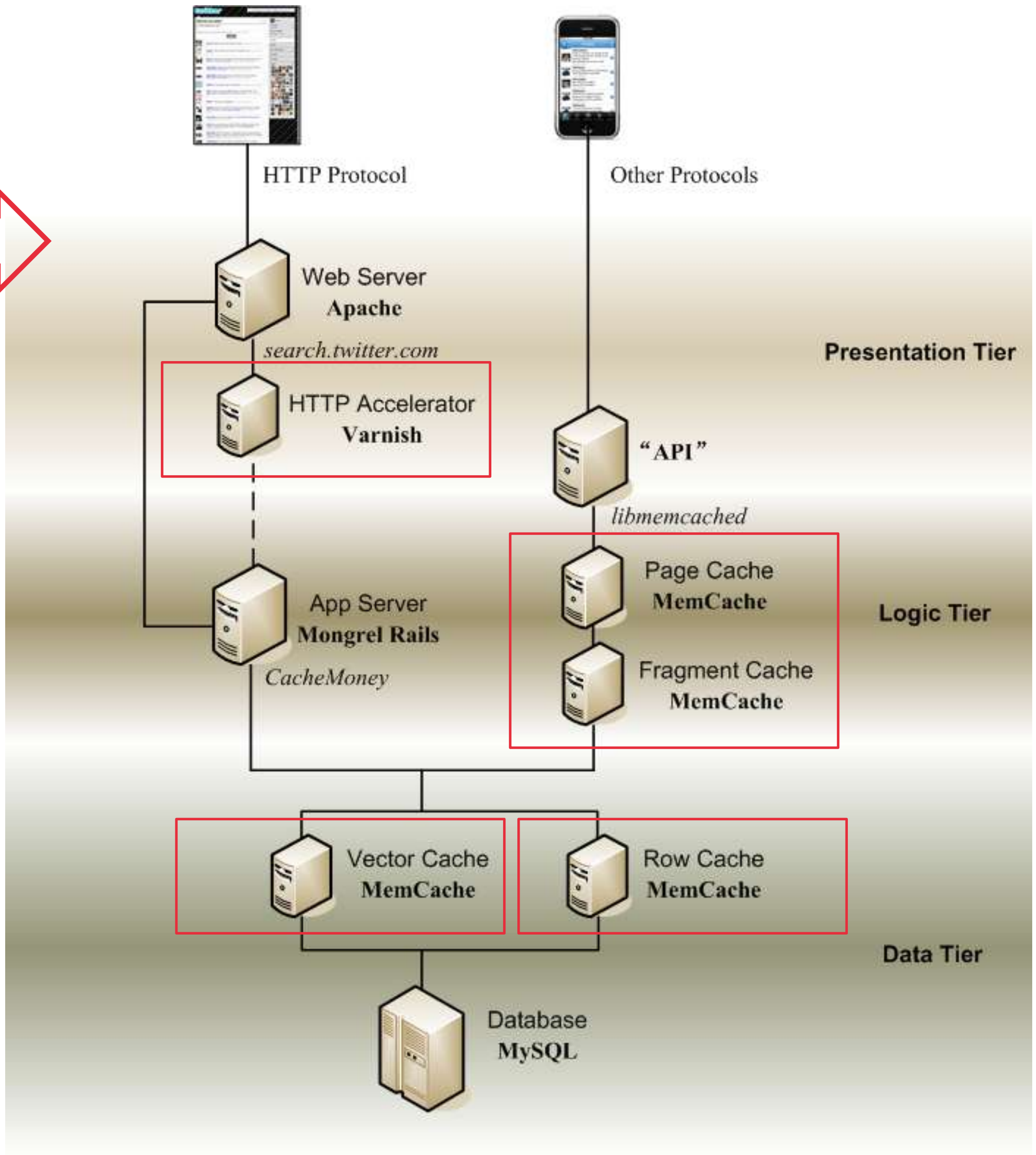
Tweet ID	Time Stamp	Author ID
6793232	2012030618455245	748229
6793231	2012030618455243	481293

Tweet ID	Tweet Content
6793232	我就喜欢这样的挑战文化。
6793231	如果我们的云计算公开课，被砸了场子，那将是我们的荣幸，因为真正的牛人出现了！

Reader ID	Tweet ID in Newsfeed
193922	6793232, 6793231, ...
748229	6793231, ...



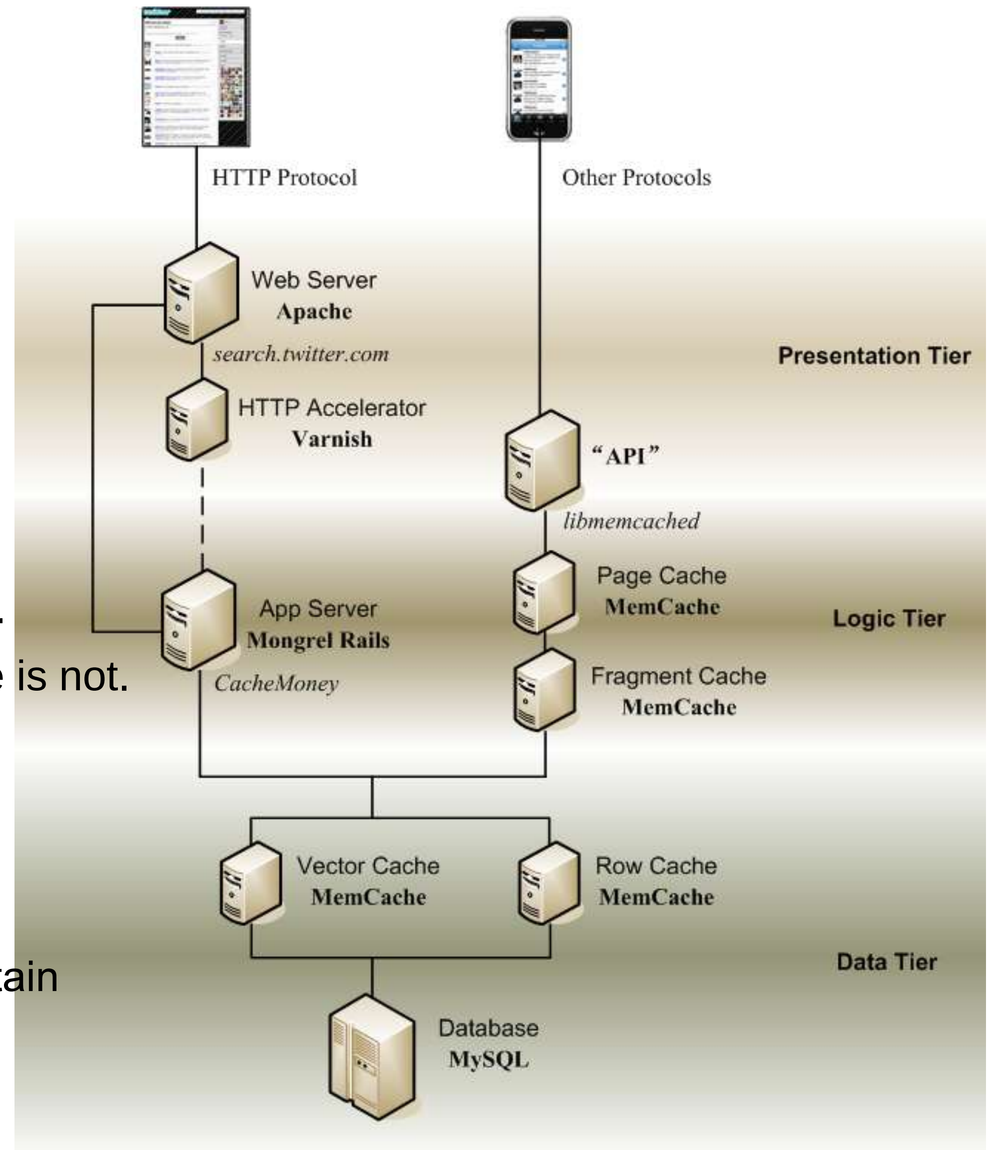
A simple but workable Twitter system.



The real Twitter system architecture.

Tweet ID	Tweet Content
6793232	我就喜欢这样的挑战文化。
6793231	如果我们的云计算公开课，被砸了场子，那将是我们的荣幸，因为真正的牛人出现了！

Reader ID	Tweet ID in Newsfeed
193922	6793232, 6793231, ...
748229	6793231, ...



- Why not use so many caches?
  Disk IO is much slower than RAM IO.
  But disk is permanent storage, cache is not.
- Message transportation,
  Why not HTTP? Overhead.
  200ms - 500ms for tweet publishing.
- Why doesn't the newsfeed table contain
  Tweet content directly,
  rather than Tweet IDs?

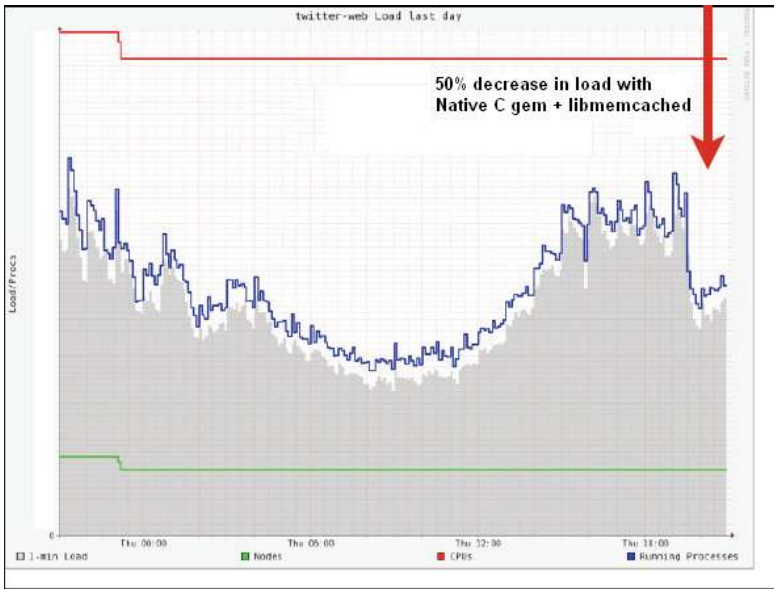
- Define the data structure first,
  Decide the workflow,
  Design the architecture.

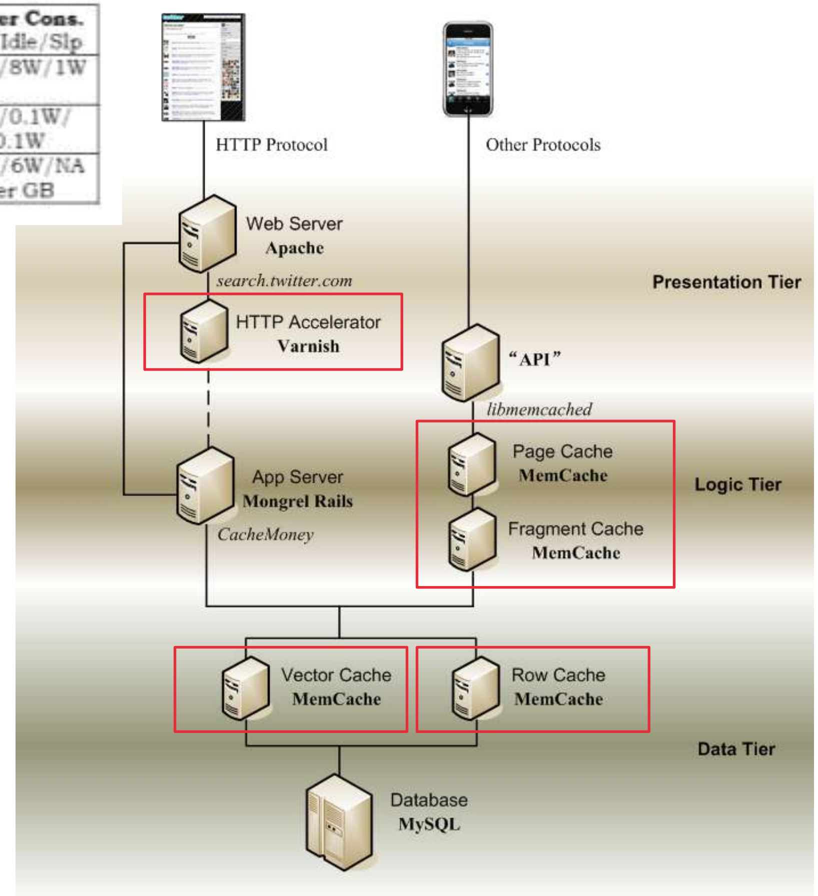
- Use IDs more, move content less.
  So called, “separate signal control from data flow”.

- Use cache more, write into disk less.
  Database is usually bottleneck.

Inside MemCached

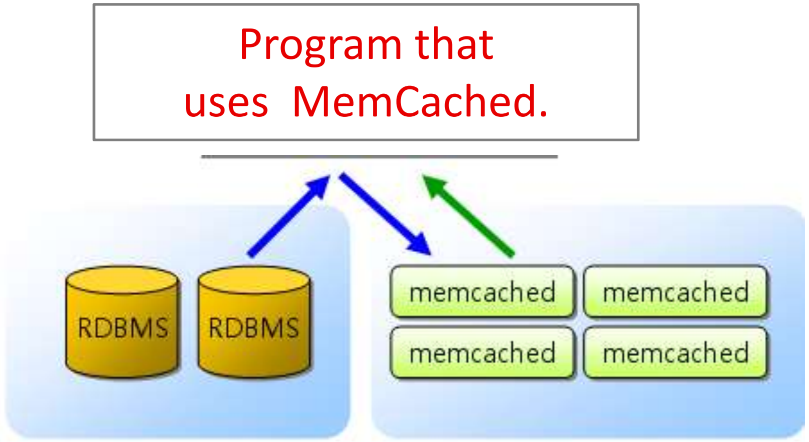
Media	Price	Access time			Transfer bandwidth	Power Cons.
		Read	Write	Erase		Act/Idle/Slp
Magnetic Disk	$0.3/GB	12.7 ms (2 KB)	13.7 ms (2 KB)	N/A	85 MB/s	13W/8W/1W
NAND Flash	$30/GB	80 µs (2 KB)	200 µs (2 KB)	1.5 ms (128 KB)	25 MB/s	1W/0.1W/0.1W
DDR2-533 RAM	$25/GB	22.5 ns	22.5 ns	N/A	4266 MB/s	12W/6W/NA per GB





- 500ms for tweet publishing,
  Disk IO is million times slower than RAM.
- E.g. By using Varnish, to cache search results,
  Twitter load decreased for 50%.

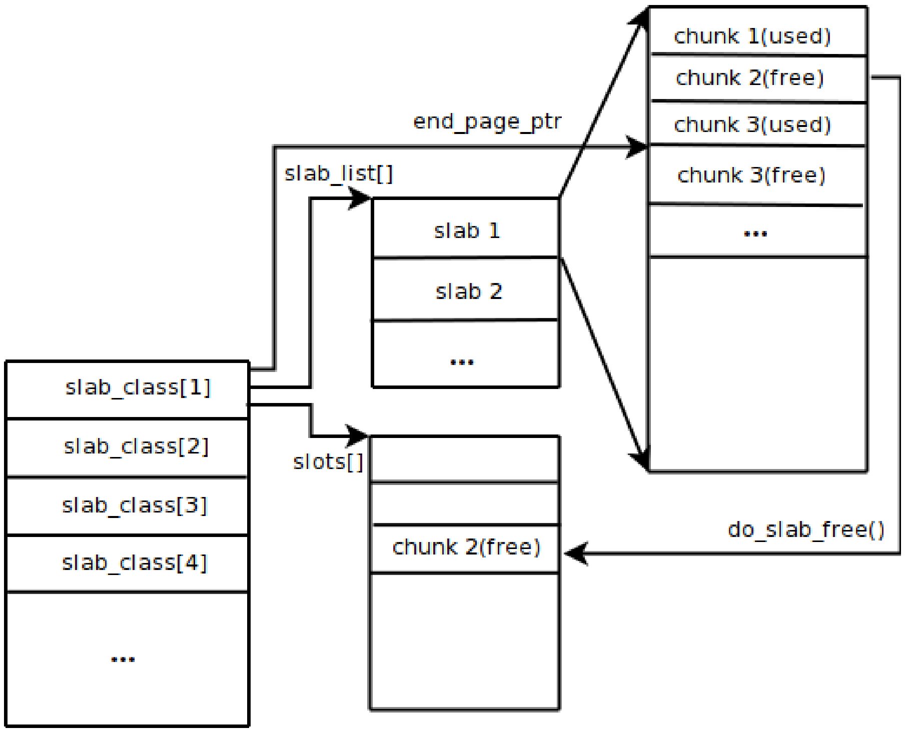
Usage of MemCached.

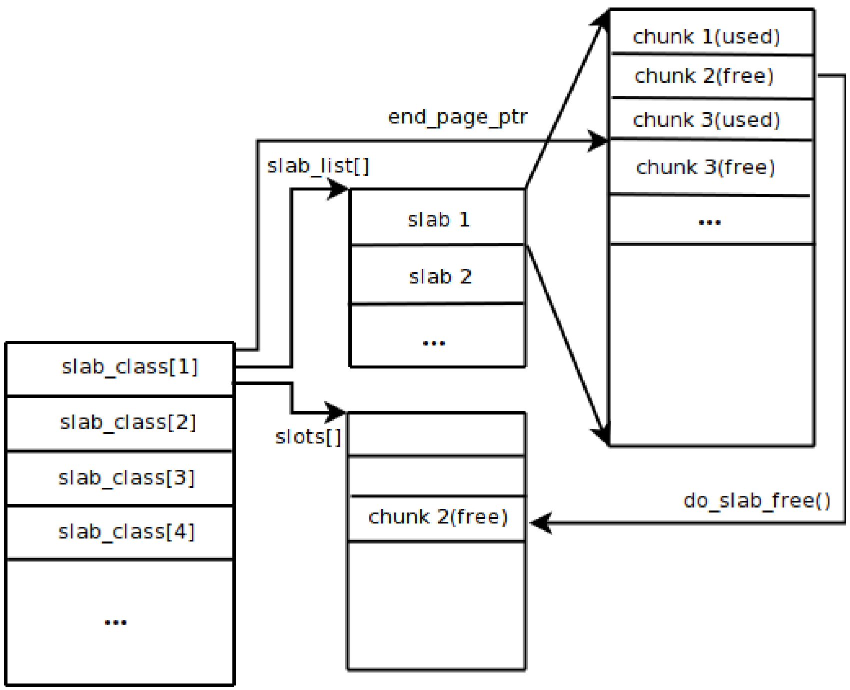


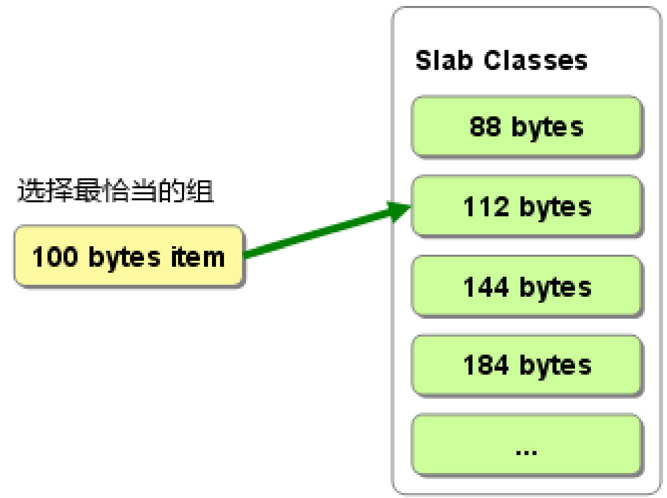

```
function get_foo(int userid) {
    /* first try the cache */
    data = memcached_fetch("userrow:" + userid);
    if (!data) {
       /* not found : request database */
       data = db_select("SELECT * FROM users WHERE userid = ?", userid);
       /* then store in cache until next get */
       memcached_add("userrow:" + userid, data);
    }
    return data;
}
```

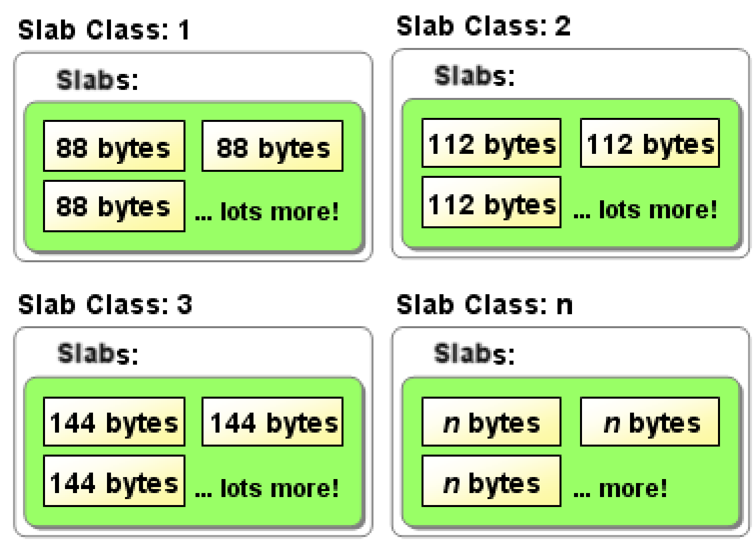
```
function update_foo(int userid, string dbUpdateString) {
  /* first update database */
   result = db_execute(dbUpdateString);
   if (result) {
      /* database update successful : fetch data to be stored in cache */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);
      /* last line could also look like data = createDataFromDBString(dbUpdateString); */
      /* then store in cache until next get */
      memcached_set("userrow:" + userid, data);
   }
}
```

- Slabs are RAM spaces of fixed-size.
- The slabs of the same size, are grouped into SlabClass.
- Each slab consists of many chunks. Usually in the same slab, the chunks are of the same size.
- Each chunk usually contain one item. An item is a pair of (Key, Value).
- Slots is an address list pointing to the re-usable chunks.
- Why split the RAM into fixed-size slabs? Easy to re-use, but may waste from space.

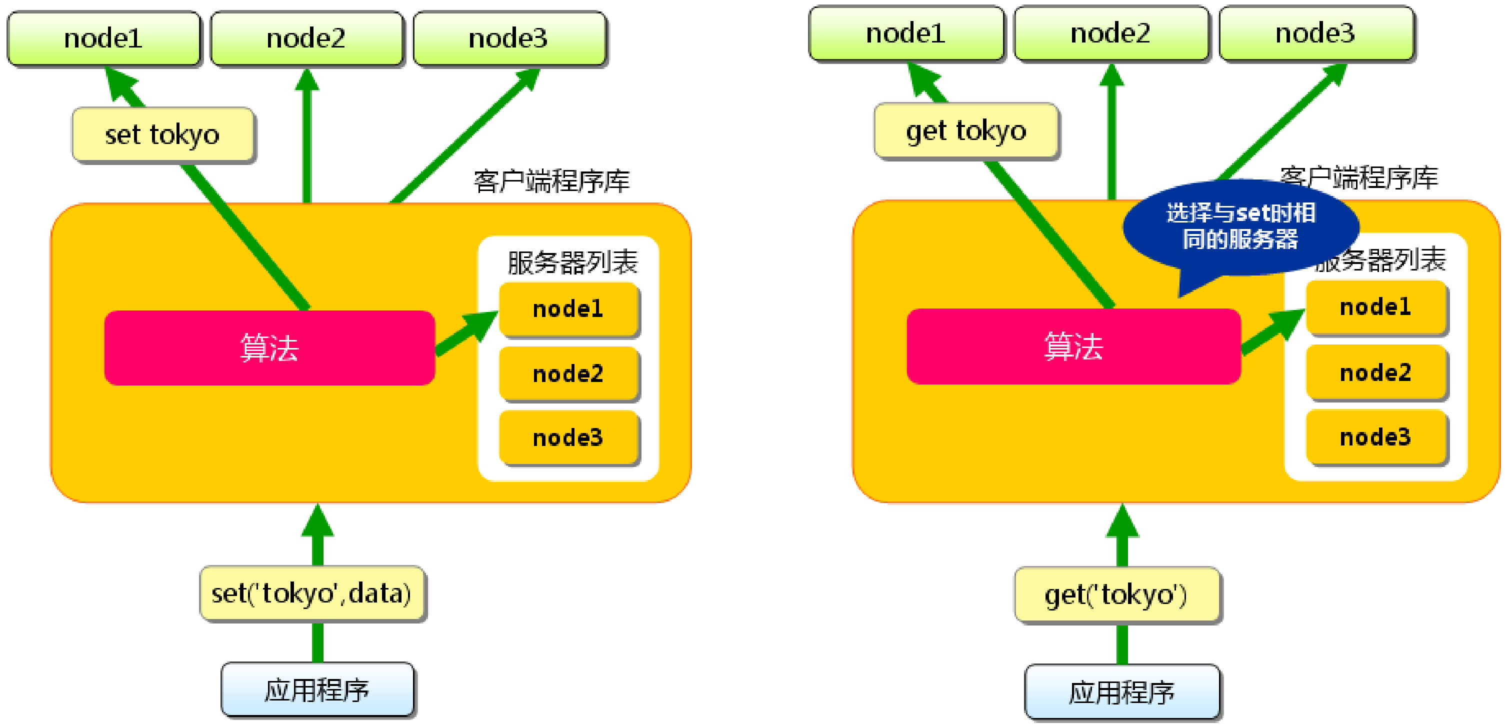




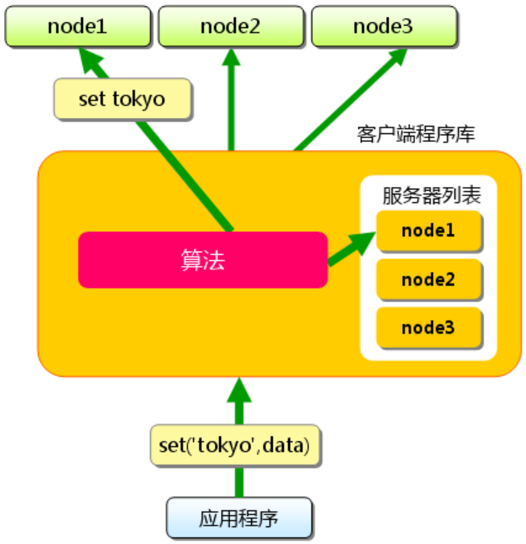


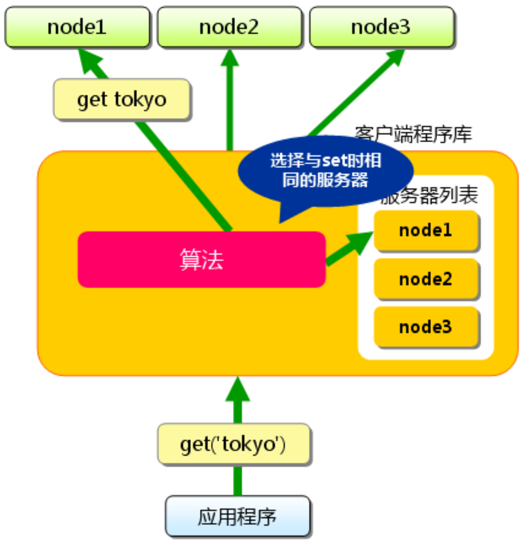


- Before caching an item,
  find the slab with the appropriate size,
  equal or a little bigger than item.



- So far, we learned how to use one single MemCached.
- In case one cache is not sufficient for a large amount of data, then we need more cache instances (nodes).
- The cache instances doesn't communicate with each other. Also, there is no shared information.
- When getting a cached data, how to know where it is cached previously?





Key	Tokyo	Beijing	New York	Hong Kong	Sydney	Paris	London	Moscow
Hash(Key)	1	2	3	4	5	6	7	8
Node = hash % (# nodes)	1=1%3	2	0	1	2	0	1	2

- When getting a cached data, how to know where it is cached previously?
- Solution 1, maintain a lookup table, {node, (key1, key2, …)}.
  Lookup table may consume too much memory space.
- Solution 2, use Hash algorithm, node = Hash(key) % (# of nodes).
  Works fine if all nodes run reliably, and the number of nodes does not change.

3 cache nodes.

Key	Tokyo	Beijing	New York	Hong Kong	Sydney	Paris	London	Moscow
Hash(Key)	1	2	3	4	5	6	7	8
Node = hash % (# nodes)	1=1%3	2	0	1	2	0	1	2

4 cache nodes.

Key	Tokyo	Beijing	New York	Hong Kong	Sydney	Paris	London	Moscow
Hash(Key)	1	2	3	4	5	6	7	8
Node = hash % (# nodes)	1=1%4	2	3	0	1	2	3	0

- Suppose when items are cached, there are 3 cache nodes, but when items are fetched, there are 4 cache nodes.
- There will be many items, cached previously but miss hit.
- Not a lethal damage, but increase database's load.

3 cache nodes.

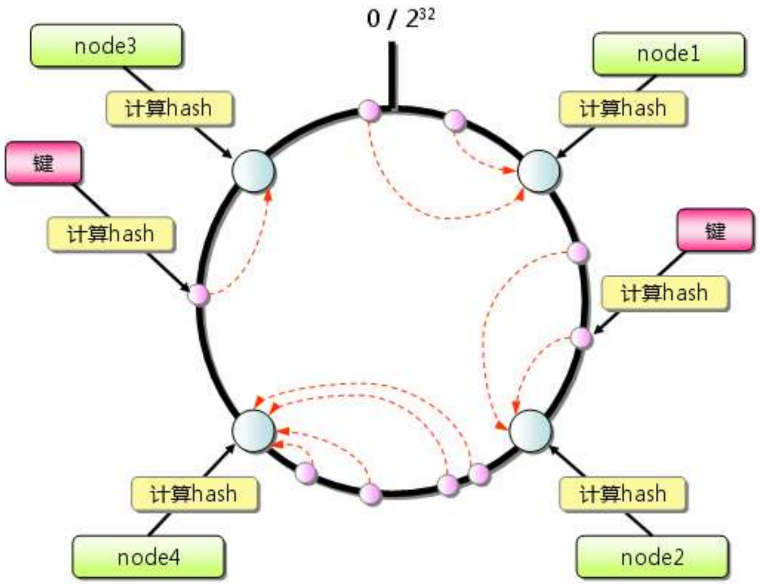
Key	Tokyo	Beijing	New York	Hong Kong	Sydney	Paris	。。。	Moscow
Hash(Key)	1	2	3	4	5	6	。。。	2^32
Node	1				2		3	

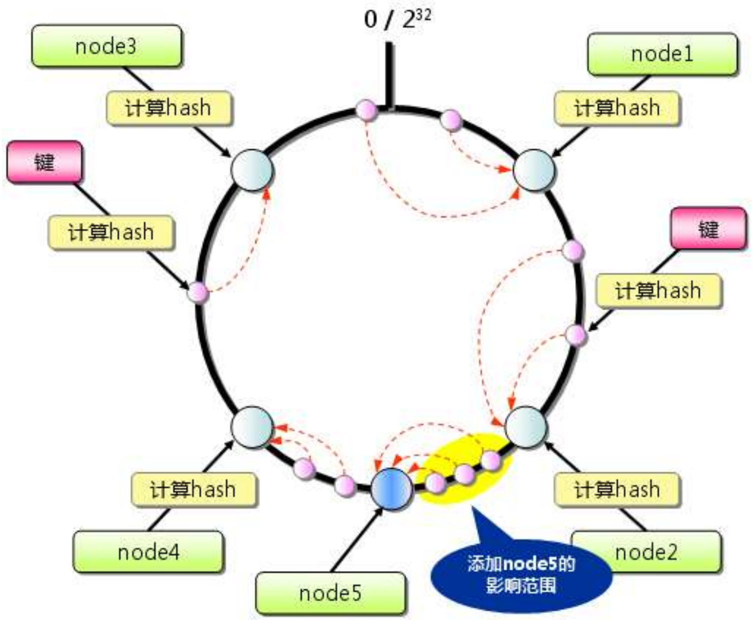
4 cache nodes.

Key	Tokyo	Beijing	New York	Hong Kong	Sydney	Paris	。。。	Moscow
Hash(Key)	1	2	3	4	5	6	。。。	2^32
Node = hash % (# nodes)	1		4		2		3	

- Suppose when items are cached, there are 3 cache nodes, but when items are fetched, there are 4 cache nodes.
- Map the hash values into the various cache node. Why 2^32? Because each cache node as an IP address, which is 4 bytes, 32 bits.
- There will be only a few items, cached previously but miss hit.

- In academia, the algorithm is called Consistent-Hash.
- Its mission is to reduce miss-hit, when adding or deleting cache nodes.
- Also applicable to many other uses, including No-SQL database.
- Map each node's IP onto a ring of size 2^32.
- Use the same mapping algorithm, map the key to the same ring.
- Clock-wisely find the node on the ring, which is nearest to the key.
- When adding or deleting a node, only a few keys will be affected.





- Cache is for read only.

  Whenever update occurs, cache must be updated according.

- Internal data structure, fixed-size for easy-reuse.

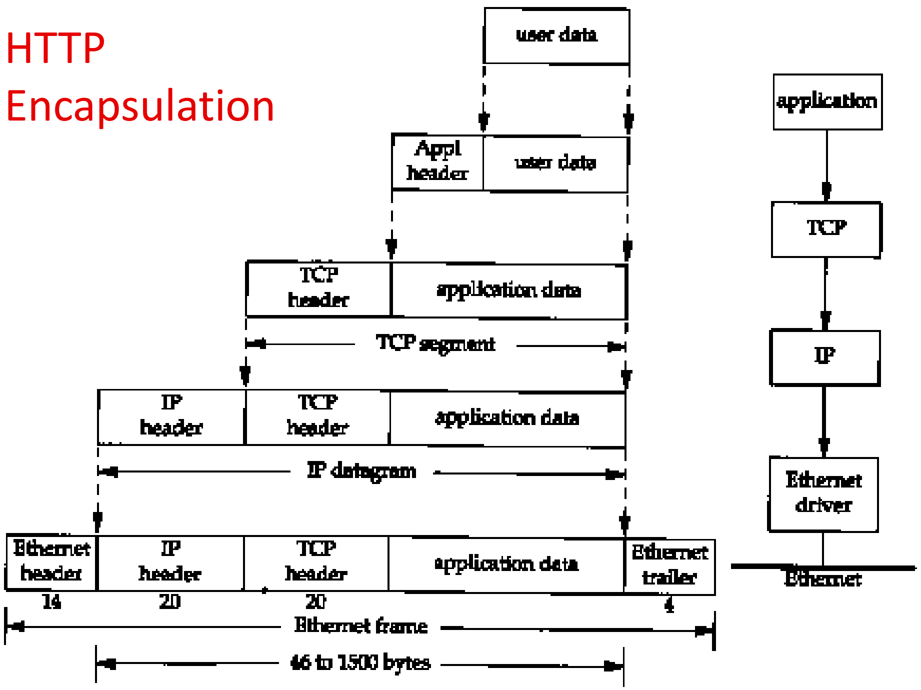
  Reuse the same space to store different data from time to time.

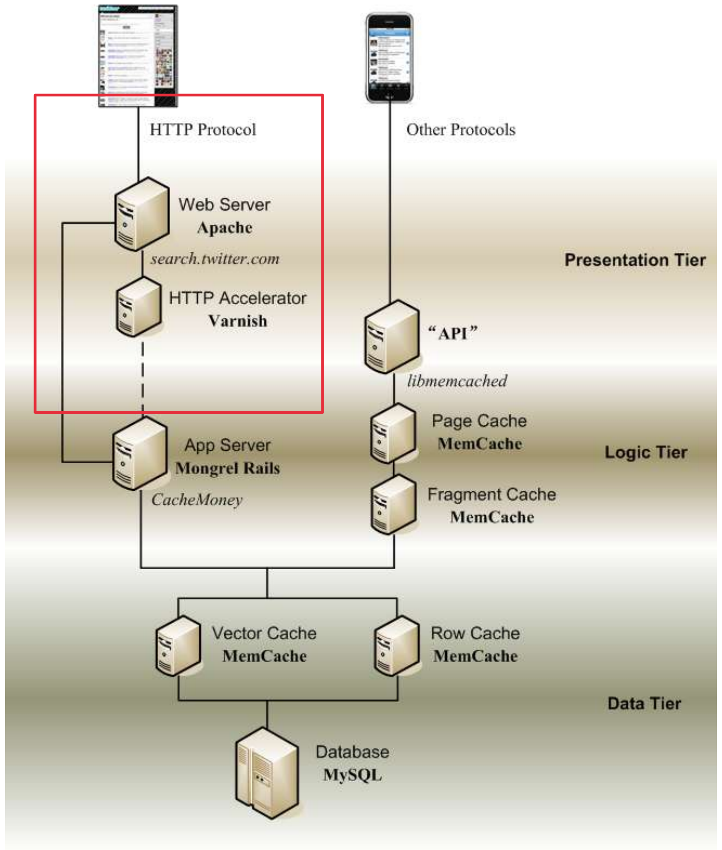
- When using multiple cache nodes,

  Consistent Hash reduces mis-hits, when adding or deleting nodes.

Inside Thrift
A Message Pipe Framework

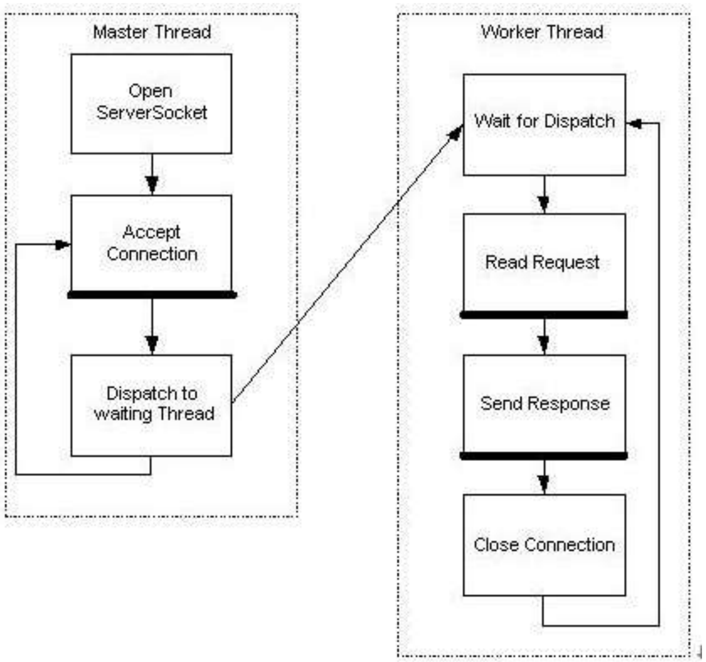
HTTP Encapsulation

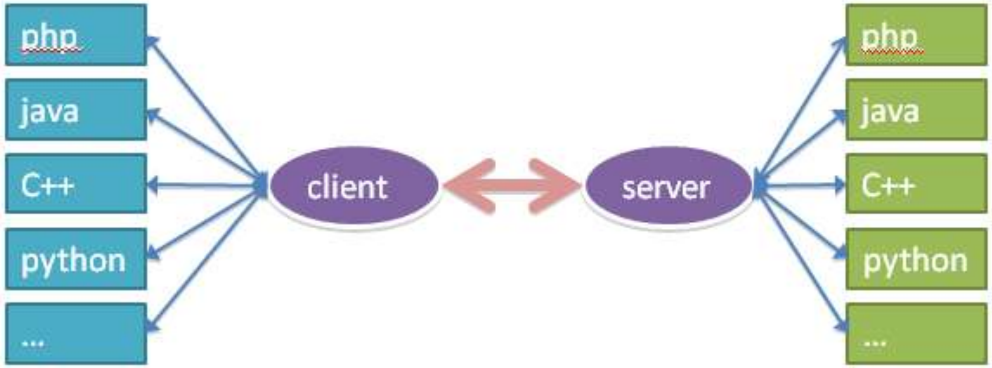




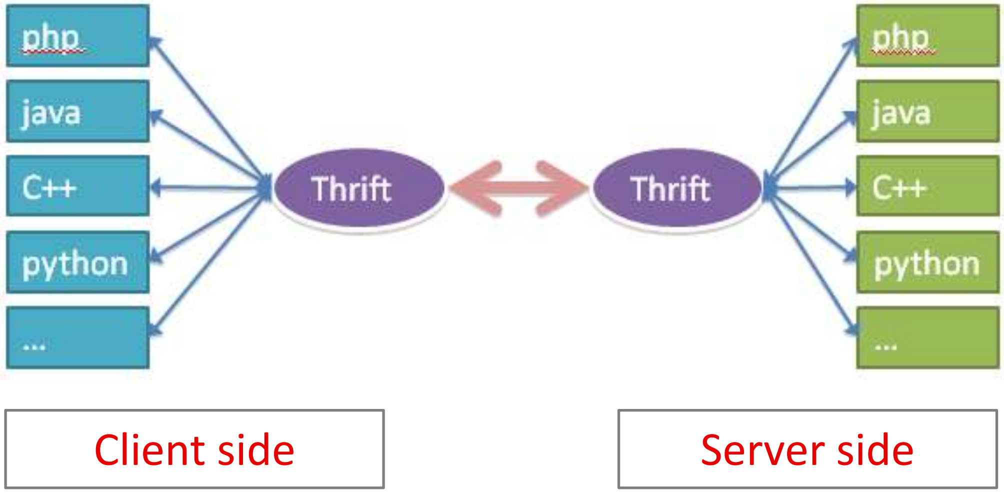
- Message transportation,
  Why not HTTP? Reduce overhead.
- More layers of protocols, more processing cost.
- Especially for frequent, but small-sized control signal messages, as fewer protocol layers as possible.

- TCP connection, Server-side:
  1. Open ServerSocket
  2. Accept Connection
  3. Read Request
  4. Send Response
  5. Close Connection.
- Stream oriented vs. Buffer oriented
  Block IO vs. Non blocking IO
- Encoding/decoding objects.
  IDL, XML, JSON
- Remote Procedure Call (RPC)
  CORBA, DCOM, SOAP, RMI
- Cross languages.
- Any tool to make it easier?





- THRIFT is
  A cross-language framework,
  to generate skeleton programs,
  to setup TCP/IP connections,
  of different types.
- THRIFT supports
  the encoding/decoding
  of popular data-types,
  also supporting RPC.
- THRIFT is
  an open framework.
  User can plug-in
  self-developed
  transport, protocol,
  and data-types.



Types	User defines data structs and service APIs, Write into a script file “xxx.thrift”. In command line, compile the script file, generate several THRIFT skeleton programs, of different language.
Processor	User implements the skeleton program, with the service business logics.
Protocol	User implements the skeleton programs, specifying the encoding/decoding mechanism, by calling THRIFT protocol APIs.
Transport	User implements the skeleton programs, specify the blocking vs non-block, streaming vs buffering connection type, by calling THRIFT transport APIs.

Usage of Thrift.

1. Write Thrift script.
2. Compile the script, generate a few skeleton programs.
3. Fill the details into the skeleton.
4. Compile the skeleton programs, and deploy, then run!

```
创建脚本文件 testJava.thrift，脚本文件内容如下：
namespace java com.javabloger.gen.code   # 注释1  定义生成代码的命名空间，与你需要定义的packag

struct Blog {   #  注释2.1  定义实体名称和数据结构，类似你业务逻辑中的pojo get/set
  1: string topic    #  注释2.2  参数类型可以参见 Thrift wiki
  2: binary content
  3: i64   createdTime
  4: string id
  5: string ipAddress
  6: map<string,string> props
 }
service ThriftCase {  #  注释3   代码生成的类名，你的业务逻辑代码需要实现代码生成的ThriftCase.Iface
  i32 testCase1(1:i32 num1, 2:i32 num2, 3:string  num3) #注释4.1 方法名称和方法中的入参，入参
  list<string> testCase2(1:map<string,string>  num1)
  void testCase3()
  void testCase4(1:list<Blog> blog)  #  注释4.2  list 是thrift中基本数据类型中的一种，list中包含的Bl
struct中定义的
 }
```

```
int main(int argc, char** argv) {
  shared_ptr<TTransport> socket(new TSocket("localhost", 9090));
  shared_ptr<TTransport> transport(new TBufferedTransport(socket));
  shared_ptr<TProtocol> protocol(new TBinaryProtocol(transport));
  example::BookServletClient client(protocol);
try {
  transport->open();
  vector<example::Book_Info> books;
  ….
  client.Sender(books);//RPC函数，调用serve端的该函数
  transport->close();
} catch (TException &tx) {
  printf("ERROR: %s\n", tx.what());
}
}
```

thrift –r –gen cpp service.thrift

service_constants.h   service_constants.cpp
service_types.h   service_types.cpp
SharedService .h,
SharedService .cpp,   SharedService_server.skeleton.cpp

Example:

http://www.javabloger.com/article/thrift-java-code-example.html?source=rss

http://dongxicheng.org/search-engine/thrift-rpc/

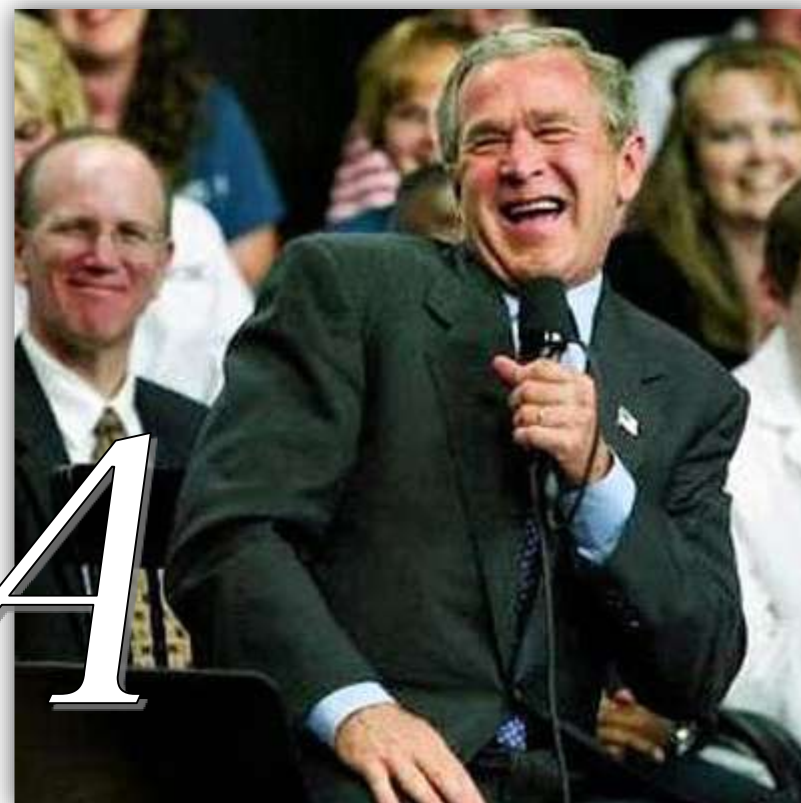
- Network connection programming is different.

  Transport, Protocol, Processor ...

- Framework generates skeleton,

  You write a script, Thrift generates skeletons, then you fill in the details.

Q&A



- No stupid questions, but it is stupid if not ask!
- Ask a good question, and impress your professor and classmates!