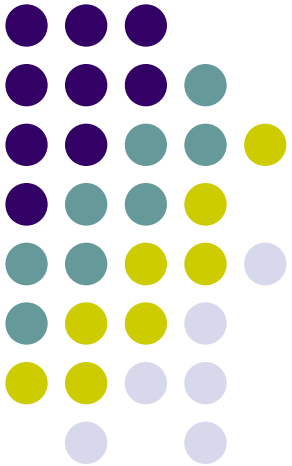
# 网络编程技术

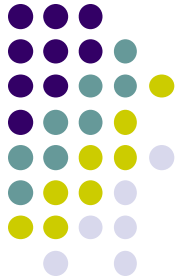
应用层协议设计

清华大学网研院

张千里(zhang@cernet.edu.cn)

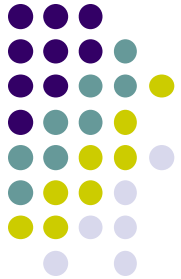


# 协议基本概念（一）
# 定义



- 协议是计算机网络和分布式系统中各种通信实体或进程间相互交换信息时必须遵守的一组规则或约定。
  - 实体是指任何可以发送或接收信息的硬件或软件进程，在大多数情况下，就是一个特定的软件进程。
  - 位于不同系统的同一层次内交互的实体，就构成了对等实体。
- 协议测试
  - 一致性测试：协议实现是否能满足该协议规范所规定的所有规则
  - 性能测试：协议效率、吞吐量、差错率和时延等
  - 互操作性测试：协议实现的不同版本的互操作性
  - 鲁棒性测试：协议是否存在未被考虑的异常路径

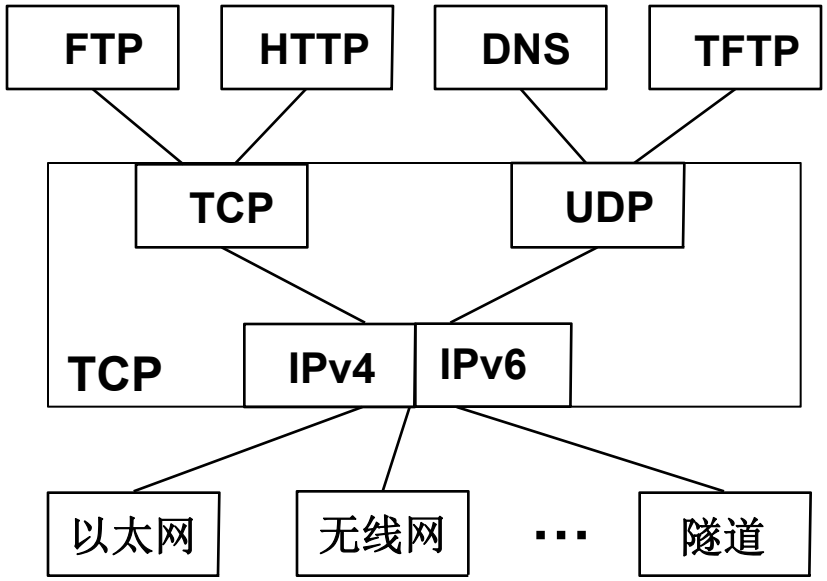
# 协议基本概念（二）
# 设计准则



- 简易
  - 层次上的简易：减少额外的层次
  - 功能设计上的简易：减少协议的状态
- 可扩展
  - 规模上的可扩展：在大规模应用时不会出现问题
  - 功能上的可扩展：在新功能加入时不会出现问题
- 互操作
  - 异构系统上的互操作：异构系统下不会出现问题
  - 不同实现上的互操作：不同实现下不会出现问题

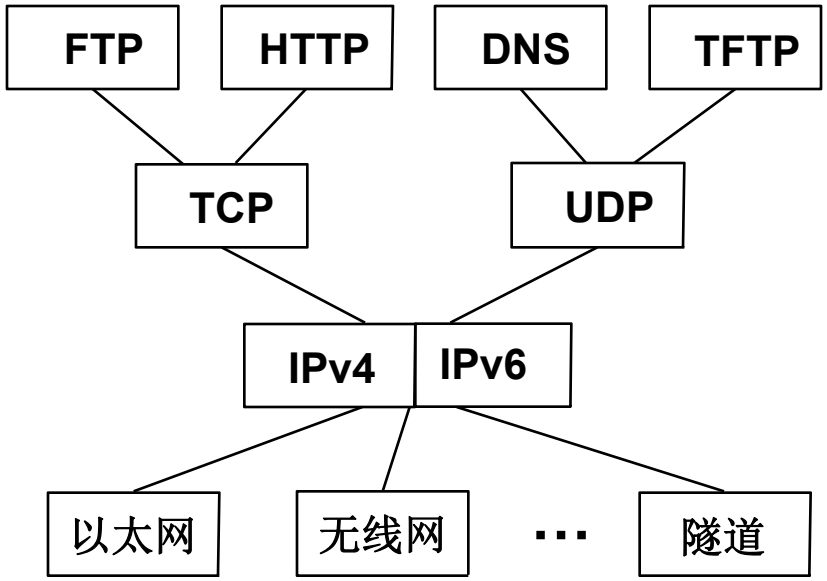
# 协议基本概念（三）
# 简易：避免不必要的复杂

- 简易的设计思想
  - 最重要的需求：远程登陆
  - 这些需求的最基本功能：可靠传输
  - 其他需求的实现：可扩展
- 避免复杂的准则
  - 角色颗粒化：控制和数据分离（名字服务、元数据、配置服务等与数据服务分离）
  - 传输统一化：使用相同的传输层以减少设计复杂度
  - 功能选项化：协议设计之初就需要考虑到安全、性能等相关选项，但是可以先不实现

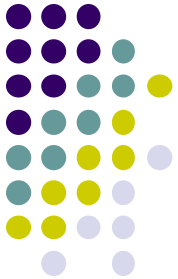


# 协议基本概念（四）
# 可扩展与可演化

- 需求的增加
  - 多媒体通讯的需求导致**UDP**的出现
- 规模的扩张
  - 域名从全局文件到**DNS**协议
- 版本的更迭
  - **IPv4**到**IPv6**



# 协议基本概念（五）
# 互操作



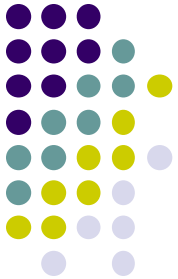
- 不同的实现
  - 字节序问题
- 新旧版本的互操作
  - ***Graceful degradation***
- 协议可选项的不同取舍
  - ***Be liberal in what you require but conservative in what you do***
  - ***Be liberal in what you accept but conservative in what you send***

# 协议举例：HADOOP的RPC

```
+-------------------------------+
|  "hrpc" 4 bytes               |
+-------------------------------+
|  Version (1 byte)             |
+-------------------------------+
|  Service Class (1 byte)       |
+-------------------------------+
|  AuthProtocol (1 byte)        |
+-------------------------------+
```

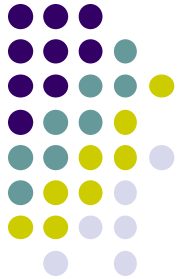
```
//h   r    p    c
0x68, 0x72, 0x70, 0x63,
// version, service class, AuthProtocol
0x09, 0x00, 0x00,
// size of next two size delimited protobuf objets:
// RpcRequestHeader and IpcConnectionContext
0x00, 0x00, 0x00, 0x32, // = 50
// varint encoding of RpcRequestHeader length
0x1e, // = 30
0x08, 0x02, 0x10, 0x00, 0x18, 0xfd, 0xff, 0xff, 0xff, 0x0f,
0x22, 0x10, 0x87, 0xeb, 0x86, 0xd4, 0x9c, 0x95, 0x4c, 0x1
0x8a, 0xb0, 0xd7, 0xbc, 0x2e, 0xca, 0xca, 0x37, 0x28, 0x0
// varint encoding of IpcConnectionContext length
0x12, // = 18
0x12, 0x0a, 0x0a, 0x08, 0x65, 0x6c, 0x65, 0x69, 0x62, 0x6
0x76, 0x69, 0x1a, 0x04, 0x70, 0x69, 0x6e, 0x67,
```

# 应用层协议设计（一）
# 基本内容

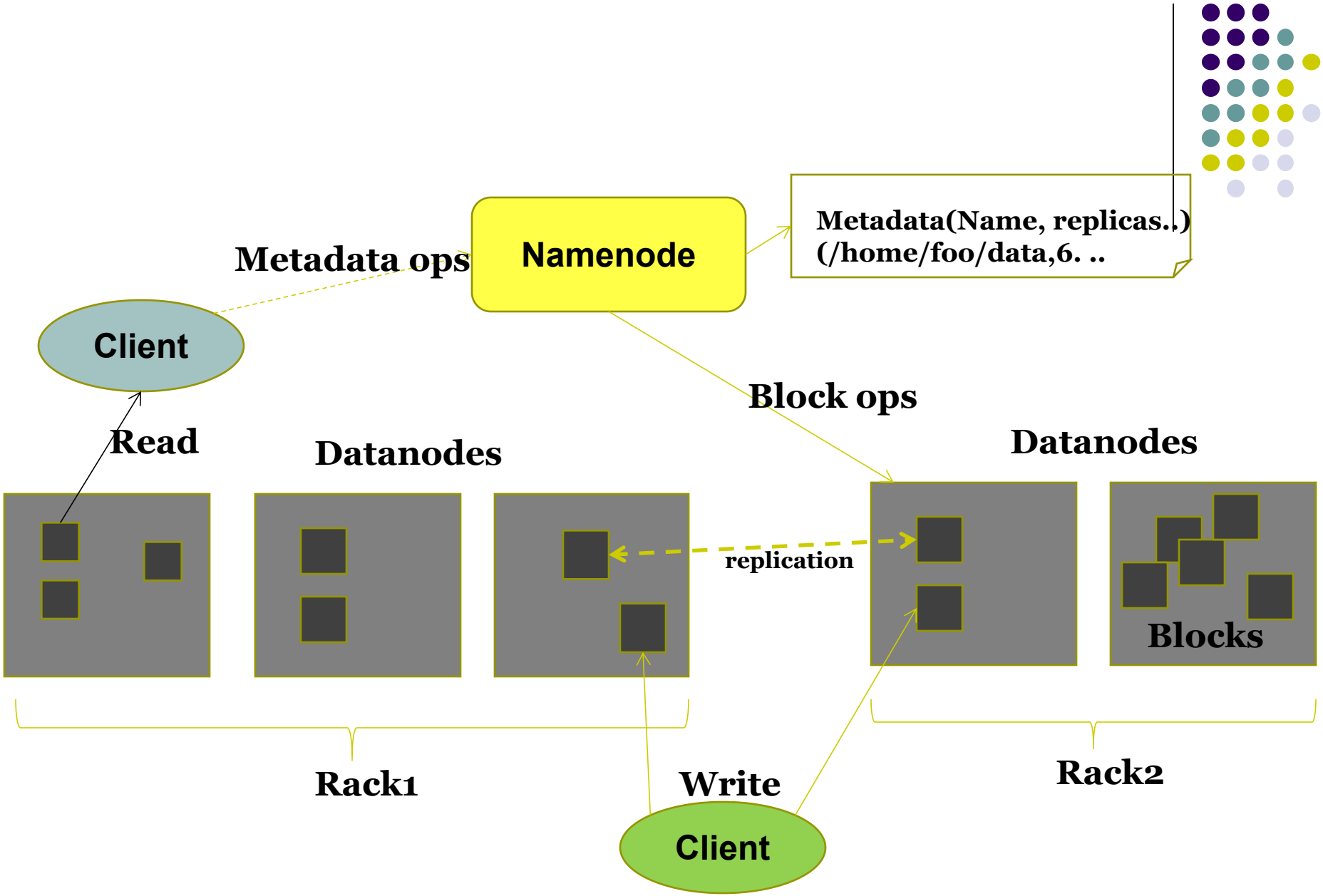


- 角色的区分和定义
- 状态的定义和转移
- 协议数据单元（PDU）定义
- 协议传输编码格式
- 传输层协议

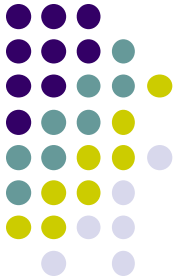
# 应用层协议设计（二）
# 角色



- 角色划分
  - 客户端
  - 名字服务器
  - 时间服务器
  - 数据服务器
  - …
- 如何划分角色
  - 面向的实体不同
  - 使用的数据不同
  - 安全的权限不同
  - 设计的重点不同

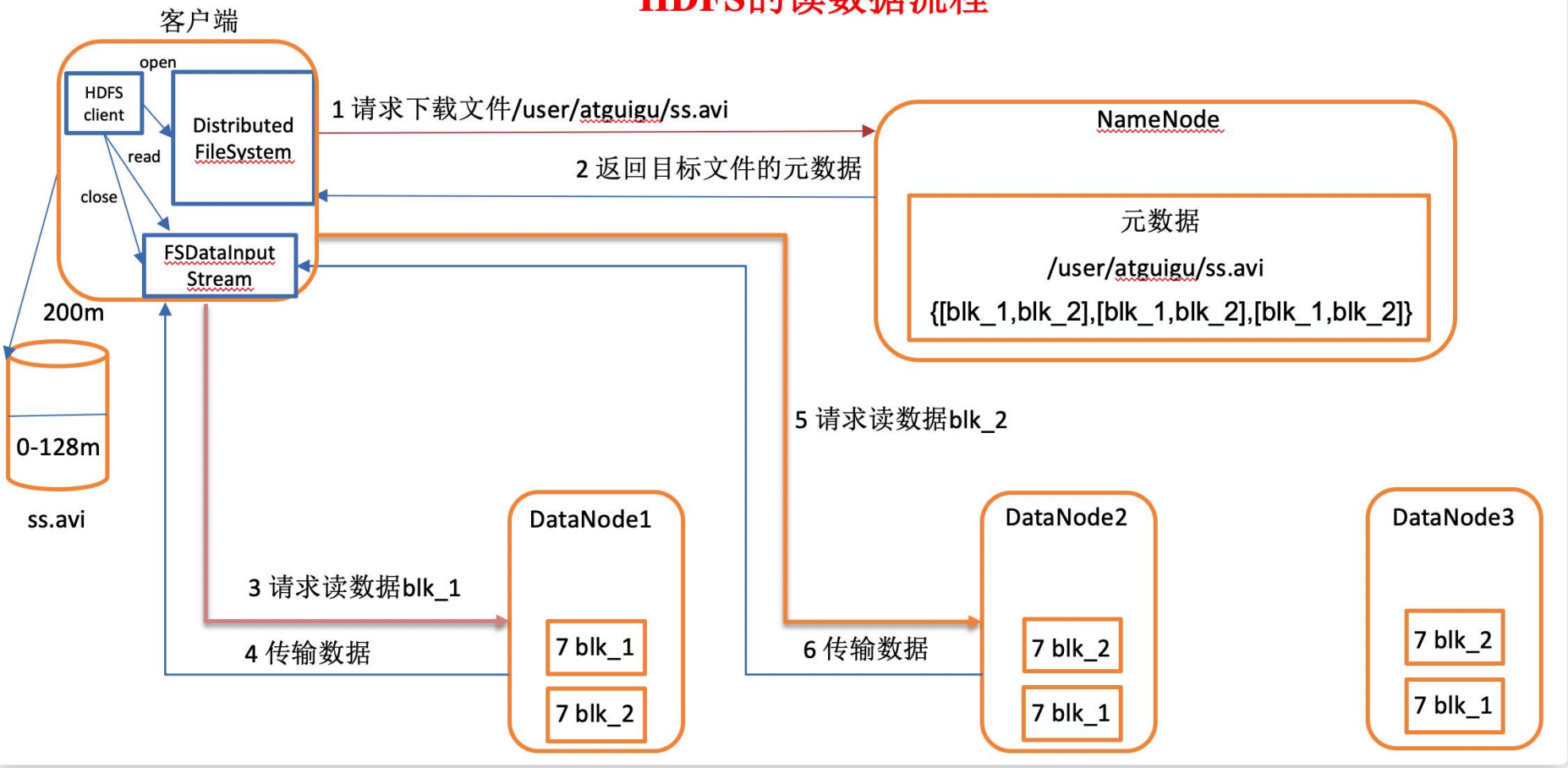

Metadata ops
Namenode
Metadata(Name, replicas..)
(/home/foo/data,6. ..
Client
Block ops
Read
Datanodes
Datanodes
replication
Blocks
Rack1
Write
Client
Rack2

# 应用层协议设计（三）
# 状态

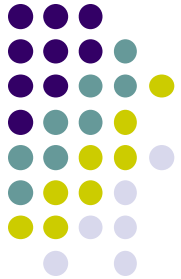


- 一问一答
  - 一次请求得到一次应答
  - 典型如**DNS**、**WWW**
- 有依赖条件的一问一答
  - 在多个依赖条件满足之后进入一问一答状态
    - 依赖条件一：需要认证
    - 依赖条件二：指定目录
  - 典型如**FTP**协议

# HDFS的读数据流程

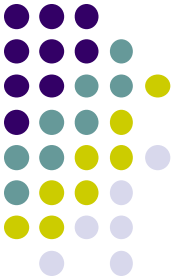

客户端
open
HDFS client
Distributed FileSystem
read
close
FSDataInput Stream
200m
0-128m
ss.avi
1 请求下载文件/user/atguigu/ss.avi
2 返回目标文件的元数据
NameNode
元数据
/user/atguigu/ss.avi
{[blk_1,blk_2],[blk_1,blk_2],[blk_1,blk_2]}
3 请求读数据blk_1
4 传输数据
5 请求读数据blk_2
6 传输数据
DataNode1
7 blk_1
7 blk_2
DataNode2
7 blk_2
7 blk_1
DataNode3
7 blk_2
7 blk_1

# 应用层协议设计（四）
# 协议数据单元的切割



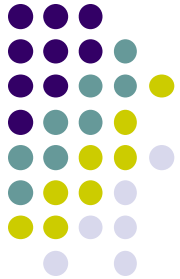
- 识别每个协议单元的开始和结束
  - 将TCP数据流切割为各个协议原语
  - UDP协议中每个报文和协议原语的对应
- 解决方案
  - 固定格式：固定长度的头、分隔字符
  - 指定长度：指定可变部分的长度
  - 传输层指定：一次TCP连接、一个UDP数据报文

# 应用层协议设计（五）
# 协议数据单元的内容



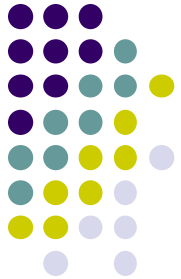
- 基本内容
  - 状态：命令类型和回应状态
  - 并发：命令和回答的匹配
  - 安全：认证、加密
  - 扩展：功能扩展和版本更迭
- 参考资料
  - **RFC 3117: On the Design of Application Protocols**

# 应用层协议设计（五）
# 编码

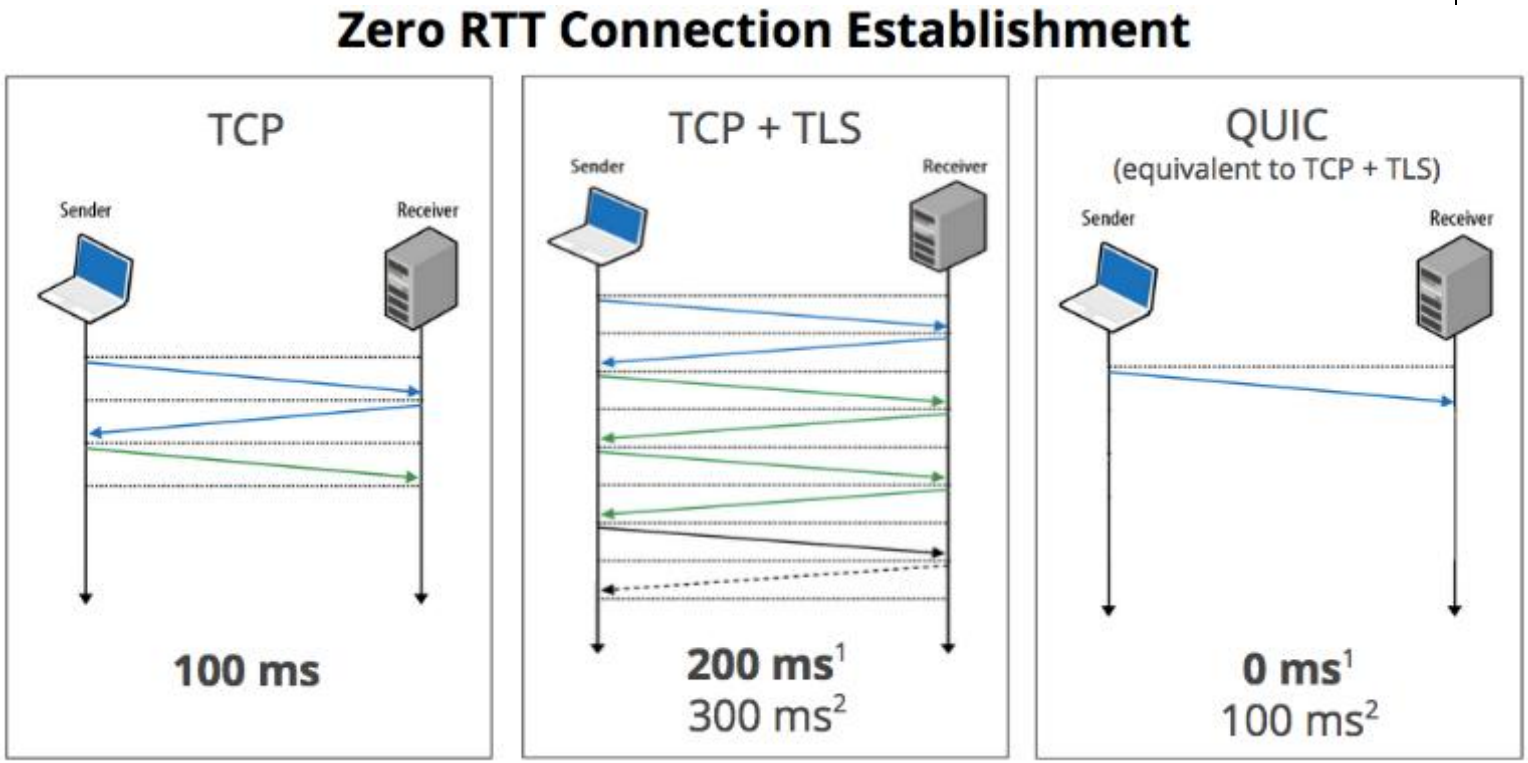


- 主要问题
  - 区分协议的控制信息和内容信息
  - 数据内容的序列化
- 解决方法
  - 文本格式
    - **SMTP、WWW、FTP**等控制协议
    - **MIME**的内容协议
    - 自定义的如**JSON、XML**
  - 二进制格式
    - 固定格式包头如**TCP、IP**等，采用固定字节序如网络字节序
    - 自定义的如**Protocol Buffer、Thrift**等

# 应用层协议设计（六）
# 传输

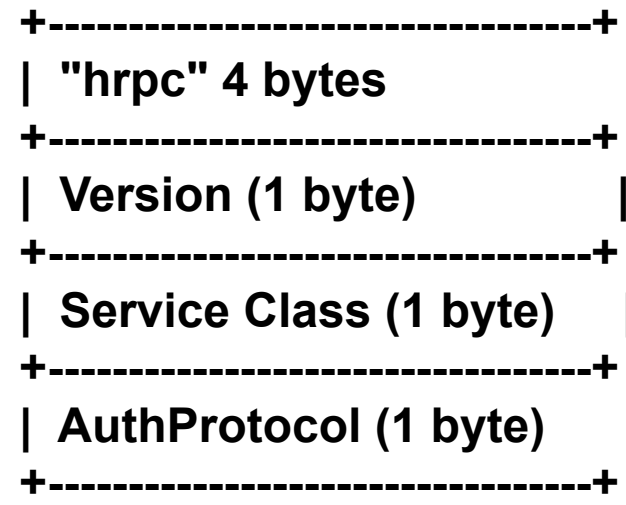


- **UDP**
  - 优点：性能高、开销小、并发好
  - 缺点：代码复杂度较高
- **TCP**
  - 优点：代码复杂度低、可靠性好
  - 缺点：较为固化
- **HTTP/HTTPS**
  - 优点：设计实现较为简单
  - 缺点：性能较差


Zero RTT Connection Establishment
TCP
Sender
Receiver
100 ms
TCP + TLS
Sender
Receiver
200 ms[1]
300 ms[2]
QUIC
(equivalent to TCP + TLS)
Sender
Receiver
0 ms[1]
100 ms[2]

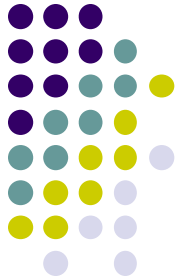
# 协议举例：HADOOP的RPC

思考：为什么头7个字节没有使用proto进行编码？




```
//h   r    p    c
0x68, 0x72, 0x70, 0x63,
// version, service class, AuthProtocol
0x09, 0x00, 0x00,
// size of next two size delimited protobuf objets:
// RpcRequestHeader and IpcConnectionContext
0x00, 0x00, 0x00, 0x32, // = 50
// varint encoding of RpcRequestHeader length
0x1e, // = 30
, 0x15,
0x8a, 0xb0, 0xd0x08, 0x02, 0x10, 0x00, 0x18, 0xfd, 0xff, 0x
0x22, 0x10, 0x87, 0xeb, 0x86, 0xd4, 0x9c, 0x95, 0x4c7, 0x
// varint encoding of IpcConnectionContext length
0x12, // = 18
0x12, 0x0a, 0x0a, 0x08, 0x65, 0x6c, 0x65, 0x69, 0x62, 0x6f
0x76, 0x69, 0x1a, 0x04, 0x70, 0x69, 0x6e, 0x67,
```

# 协议数据单元的内容（一）类型/状态



- 类型
  - 区分不同类型的原语
  - **WWW：请求使用GET/POST/HEAD命令进行区分**
- 状态
  - 指示是否出现错误
  - **WWW：回应中通过数字来指示不同的情况1xx（临时响应）；2xx（成功）；3xx（已重定向）；4xx（请求错误）；5xx（服务器错误）**

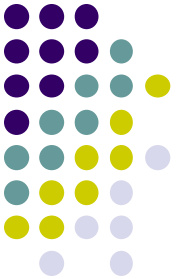
**SMTP**

- 421 <domain> Service not available, closing transmission char
- 450 Requested mail action not taken: mailbox unavailable
- 451 Requested action aborted: local error in processing
- 452 Requested action not taken: insufficient system storage
- 550 Requested action not taken: mailbox unavailable
- 551 User not local; please try <forward-path>
- 552 Requested mail action aborted: exceeded storage allocatio
- 553 Requested action not taken: mailbox name not allowed
- 553 Requested action not taken: Local user only

**FTP**

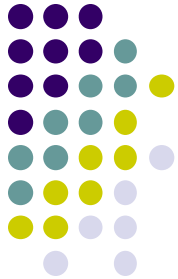
- 331 Username OK, password required
- 125 data connection already open; transfer starting
- 425 Can't open data connection
- 452 Error writing file

# 协议数据单元的内容（二）
# 并发



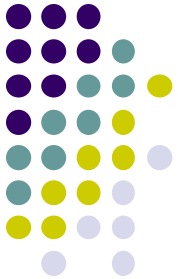
- 不同请求的并发设计
  - 区分多个/一个用户的多次同时服务（**UDP**）
  - 原语中含有“会话标识号”这一项
    - **DNS**协议中的**ID**
- 相同请求的继续
  - 区分一个用户的两次服务，其中后一次是前一次的继续（如断点续传）
  - 原语中可以指定本次会话开始的位置
    - **WWW**协议中的**Range: bytes=0-XXX**

# 协议数据单元的内容（三）
# 安全



- 认证
  - 协议应当有一种方式使得能够识别不属于本协议的数据包
  - 对于**UDP**协议，应当定义协议规范以免对所有数据包都合规
    - 有一种方式判断请求是否合规
    - **UDP**应用应当丢弃不合规的请求
  - 保持协议的可扩展性以便未来加入认证功能
    - 一个域用于表示认证信息
- 加密
  - 保持协议的可扩展性以便可能进行的压缩/加密
    - 一个域用于表示内容是否进行了加密

# 协议数据单元的内容（四）
# 扩展性



- 功能的增加
  - 报文类型：通过扩展报文类型来实现功能扩展
    - 举例：**WWW**、**FTP**、**SMTP**中的命令
    - 举例：**IP**报文中的协议字段
    - 举例：**DHCP**协议中的报文类型
  - 扩展项：通过在报文中定义扩展项
    - **TCP**、**IP**中的选项
- 协议的升级
  - 版本号：通过版本号来实现升级

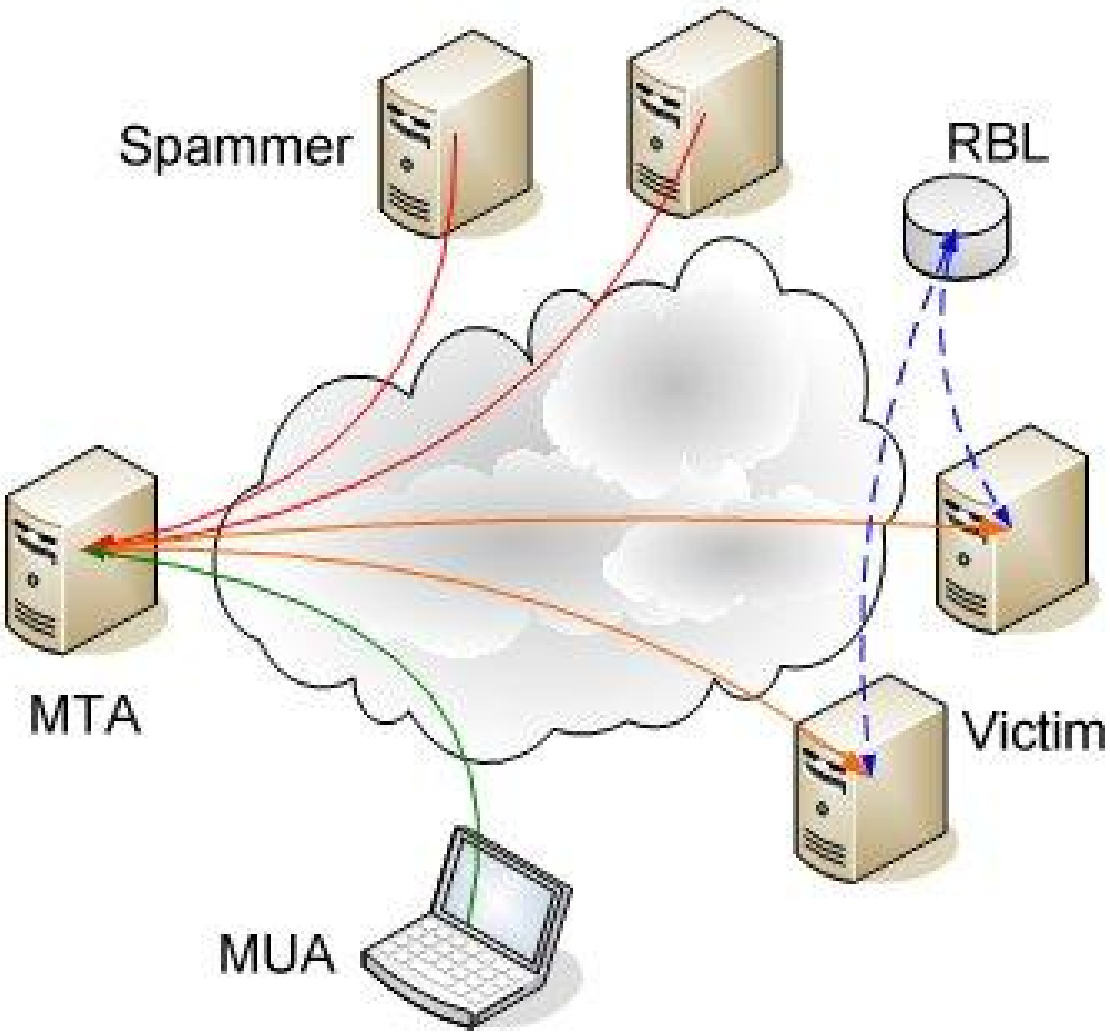
# 协议举例：HADOOP的RPC

```
+-------------------------------+
|  "hrpc" 4 bytes               |
+-------------------------------+
|  Version (1 byte)             |
+-------------------------------+
|  Service Class (1 byte)       |
+-------------------------------+
|  AuthProtocol (1 byte)        |
+-------------------------------+
```

```
//h   r    p    c
0x68, 0x72, 0x70, 0x63,
// version, service class, AuthProtocol
0x09, 0x00, 0x00,
// size of next two size delimited protobuf objets:
// RpcRequestHeader and IpcConnectionContext
0x00, 0x00, 0x00, 0x32, // = 50
// varint encoding of RpcRequestHeader length
0x1e, // = 30
0x08, 0x02, 0x10, 0x00, 0x18, 0xfd, 0xff, 0xff, 0xff, 0x0f,
0x22, 0x10, 0x87, 0xeb, 0x86, 0xd4, 0x9c, 0x95, 0x4c, 0x1
0x8a, 0xb0, 0xd7, 0xbc, 0x2e, 0xca, 0xca, 0x37, 0x28, 0x0
// varint encoding of IpcConnectionContext length
0x12, // = 18
0x12, 0x0a, 0x0a, 0x08, 0x65, 0x6c, 0x65, 0x69, 0x62, 0x6
0x76, 0x69, 0x1a, 0x04, 0x70, 0x69, 0x6e, 0x67,
```

# 举例：SMTP发信认证

- 传统的**SMTP**协议并不能进行发信认证
- 随着互联网的发展，出现了越来越多的垃圾邮件发送者，利用**openrelay**来掩盖来源

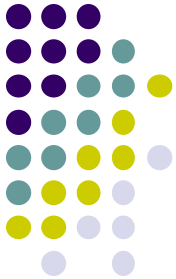


# SMTP发信认证

- 客户端发送**EHLO**之后，服务器提示有哪些认证手段

```
EHLO
250-SMTP.Mydomain.com Hello [192.168.0.1]
250-8bitmime
250-BINARYMIME
250-VRFY
250-AUTH LOGIN PLAIN CRAM-MD5
250-AUTH=LOGIN
250 OK
```

# 通过命令的扩展实现认证



C：auth login ---------- 进行用户身份认证

S：334 VXNIcm5hbWU6 ---------- BASE64编码“Username:”

C：Y29zdGFAYW1heGl0Lm5ldA== ---------- 用户名，使用BASE64编码

S：334 UGFzc3dvcmQ6 ----------BASE64编码"Password:"

C：MTk4MjIxNA== ---------- 密码，使用BASE64编码

S：235 auth successfully ---------- 身份认证成功

S: * OK IMAP4 Server

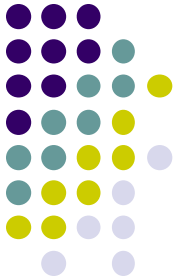
C: A0001 AUTHENTICATE CRAM-MD5

S: + PDE4OTYuNjk3MTcwOTUyQHBvc3RvZmZpY2UucmVzdG9uLm1jaS5uZXQ+ -------- Server发送BASE64编码的Timestamp、Hostname等给Client

C: dGltIGI5MTNhNjAyYzdlZGE3YTQ5NWI0ZTZlNzMzNGQzODkw ------- Client将收到的信息加上用户名和口令，编码为BASE64发送给Server

S: A0001 OK CRAM authentication successful ---------- Server使用该用户的口令进行MD5运算，如果得到相同的输出则认证成功

# 课堂讨论：一个文件下载协议

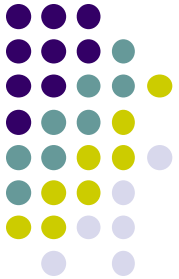


问题：该协议设计有哪些问题？

```
message request {
    required uint32 session_id = 1;
    required uint32 checksum = 2;
    required string user = 3
    required string verification_code = 4;
    required uint32 version = 5;
    required string file_name = 6;
}
```

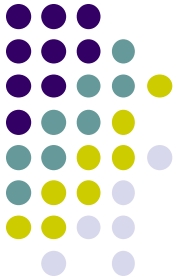
```
message response {
    enum StatusCode {
        SUCCESS = 0;
        FORMAT_ERROR = 1;
        VERIFCATION_ERROR = 2;
        FILE_NOT_FOUND_ERROR = 3;
    }
    required StatusCode statuscode = 1;
    required uint32 session_id = 2;
    required uint32 checksum = 3;
    required string verification_code = 4;
    required uint32 version = 5;
    optional uint32 sequence_num = 6;
    optional uint32 sequence_total = 7;
    optional uint32 begin_loc = 8;
    optional uint32 end_loc = 9;
    optional bytes file_data = 10;
}
```

# Request



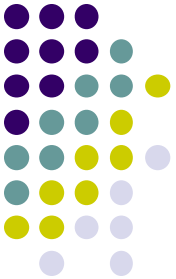
- **uint32 session_id:** 会话号，用来支持并发功能。利于**response**端来规划每次 通信是面对那个**request**
- **uint32 checksum:** 保证信息传输的可靠性，是简单的校验值
- **uint32 verification:** 验证码,用来加一层安全在文件下载上
- **uint32 version:** 版本号，用来区分两个端是否在用同一个版本
- **string file_name:** 这个是想下载的文件名，**response** 端用来查找是否有以及别的信息

# Response

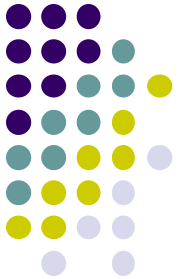


- **StatusCode statuscode:StatusCode是个 enum 类，包括了简单出错情况**
- **uint32 session_id、checksum、verification、version: 同 request**
- **uint32 sequence_num: 在成功访问文件的前提下，用来对于分块下载时的数据编号， 利于在request 端能可靠的判断数据是否全部收到。**
- **uint32 sequence_total: 同于 sequence_num, 让 request 端知道总共有多少个包，用来验 证数据是否全部收到**
- **uint32 begin_loc、 end_loc ：用来支持分块下载，标注本块是覆盖原文件的什么范围**
- **bytes file_data:具体要下载文件的内容**

# Protocol buffer（一）
# 实现目的

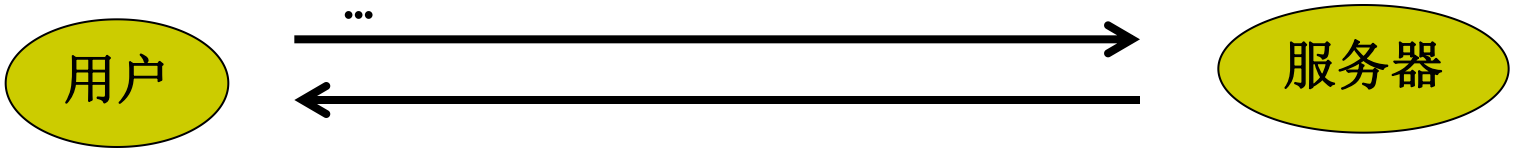


- 协议中存在大量二进制信息
  - 如**RPC**
- 协议对效率的处理效率很敏感
  - **XML**非常繁重，需要较多的处理开销
  - **JSON**虽然好一些，但是也需要二进制到文本转换
- 协议便于跨语言、跨架构的实现
  - **Protocol buffer**能够为**C/C++**、**Java**、**Python**等生成代码



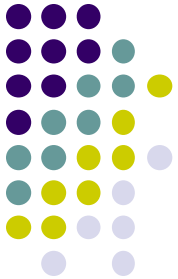
```
message HelloRequest {
    required string name = 1;
    optional int32 age = 2;
    optional int32 salary = 3;
    optional string message = 4;
}
```

```
req.set_name("用户");
req.set_age(26);
…
```




```
res.set_message("成功");
```

```
message HelloResponse {
    required string message = 1;
}
```

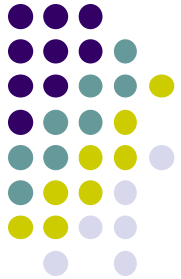


```
message [message name] {
    [field rule] [field type] [field name] = [tag];
}

field rule := required | optional | repeated
field type := int32 | int64 | uint32 | uint64 | double
            | bool | string | bytes
```

- **tag**不能重复，一旦定了也不能随意修改
- **field type**可以是已定义过的**message**
- 可以通过**import xx.proto**来引用其他文件

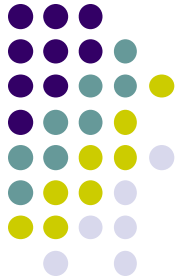
# Protocol buffer（二）描述



- Modifier（修饰符）
  - **required:** 必须存在于消息中，如果不存在，则在后续处理中总是被认为是未初始化并失败
  - **optional:** 可选项，如果不存在则设置为缺省值
  - **repeated:** 可重复项，可以认为是变长数组
- 数据声明（**declaration**）
- **Tag**：用于在编码中标识这个域

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2
[default = HOME];
  }
  repeated PhoneNumber phone =
4;
}
message AddressBook {
  repeated Person person = 1;
}
```

# Protocol buffer（三）
# 字节序



- 常见处理方法
  - 采用网络字节序
    - 如TCP、IP等各个协议
  - 采用主机字节序，但是在最初表明字节序
  - 采用little-endian字节序
    - 各个系统在实现时，定义数据转换函数，只有在主机序是big-endian时进行转换
- Protocol buffers采用的方法
  - 采用little-endian字节序
  - 每7比特编码的varint

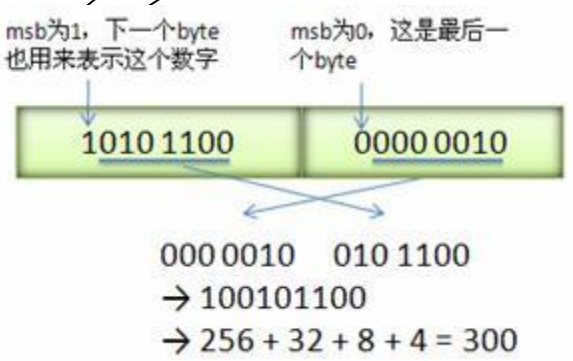
# Protocol buffer（四）
# 编码

| Type | Meaning | Used For |
|---|---|---|
| 0 | Varint | int32, int64, uint32, uint64, sint32, sint64, bool, enum |
| 1 | 64-bit | fixed64, sfixed64, double |
| 2 | Length-delimited | string, bytes, embedded messages, packed repeated fields |
| 3 | Start group | groups (deprecated) |
| 4 | End group | groups (deprecated) |
| 5 | 32-bit | fixed32, sfixed32, float |

第1项，类型为varint，则字节1:
(1<<3) | 0 =0x08

```
message helloworld {
      required int32 id = 1; // ID
      required string str = 2; // str
}
```

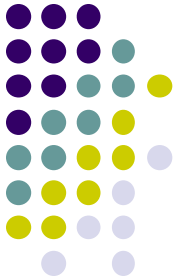
300
“hell
o”




第2项，类型为length-delimited，
则字节1:
(2<<3) | 2 =0x12
字节2为长度05
后面为”hello”

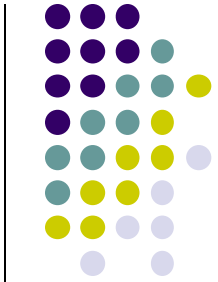
08 ac 02 12 05 68 65 6C 6C 6F

# Protocol buffer（四）使用



- 写出.proto文件
- **protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/addressbook.proto**
- 对于每个消息对应的类，都有如下接口：
  - **bool SerializeToString(string* output) const;**: 将消息序列化，这里的**string**是二进制的
  - **bool ParseFromString(const string& data);**: 从**data**中解析出消息来
  - **bool SerializeToOstream(ostream* output) const;**: 将消息写到相应的输出
  - **bool ParseFromIstream(istream* input);**: 从输入源中解析出消息

https://developers.google.com/protocol-buffers/docs/cpptutorial

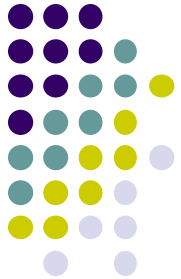


```
int main(int argc, char* argv[]) {
  GOOGLE_PROTOBUF_VERIFY_VERSION;
  if (argc != 2) {
    cerr << "Usage:  " << argv[0] << " ADDRESS_BOOK_FILE" << endl;
    return -1;
  }
  tutorial::AddressBook address_book;
    fstream input(argv[1], ios::in | ios::binary);
    if (!input) {
      cout << argv[1] << ": File not found.  Creating a new file." << endl;
    } else if (!address_book.ParseFromIstream(&input)) {
      cerr << "Failed to parse address book." << endl;
      return -1;
    }
    fstream output(argv[1], ios::out | ios::trunc | ios::binary);
    if (!address_book.SerializeToOstream(&output)) {
      cerr << "Failed to write address book." << endl;
      return -1;
    }
  google::protobuf::ShutdownProtobufLibrary();
  return 0;
}
```

# 协议举例：HADOOP的RPC

```
+-------------------------------+
|  "hrpc" 4 bytes               |
+-------------------------------+
|  Version (1 byte)             |
+-------------------------------+
|  Service Class (1 byte)       |
+-------------------------------+
|  AuthProtocol (1 byte)        |
+-------------------------------+
```

```
//h   r    p    c
0x68, 0x72, 0x70, 0x63,
// version, service class, AuthProtocol
0x09, 0x00, 0x00,
// size of next two size delimited protobuf objets:
// RpcRequestHeader and IpcConnectionContext
0x00, 0x00, 0x00, 0x32, // = 50
// varint encoding of RpcRequestHeader length
0x1e, // = 30
0x08, 0x02, 0x10, 0x00, 0x18, 0xfd, 0xff, 0xff, 0xff, 0x0f,
0x22, 0x10, 0x87, 0xeb, 0x86, 0xd4, 0x9c, 0x95, 0x4c, 0x1
0x8a, 0xb0, 0xd7, 0xbc, 0x2e, 0xca, 0xca, 0x37, 0x28, 0x0
// varint encoding of IpcConnectionContext length
0x12, // = 18
0x12, 0x0a, 0x0a, 0x08, 0x65, 0x6c, 0x65, 0x69, 0x62, 0x6
0x76, 0x69, 0x1a, 0x04, 0x70, 0x69, 0x6e, 0x67,
```



# Q&A