# 1 Chap0

## 8Fallacies

1.Network Reliable. |PowerSupply, HardDisk, NodeFailures, Configurations, Bugs| **Effect:**applicationHangs, crashes |**Countermeasures:** Redundancy HW&SW systems,middleware &application; CatchExceptions, CheckCodes, React;Retry ConnectingUponTimeouts 2.LatencyZero.|Latency:timeForData Transfer(speedOfLight)& Bandwidth:howMuchData transferred 3.Bandwidth is infinite. 4.The network is secure. 5.Topology doesn't change. 6.There is one administrator. 7.Transport cost is zero. 8.The network is homogeneous

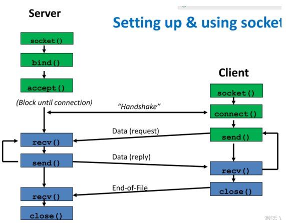
# 2 Chap1-CommunicationBasics

## NETWORKING BASICS

ISO OSI: Open Systems Interconnection model, Basis for standards development on systems interconnection.

## SOCKET

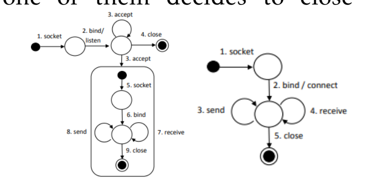
Move data/message/(invoke operation/service and return result/failure) from Application I on Host A to Application K on Host B. **Client**:Issues requests to server(send & receive). **Server**:Starts up and listens for connections, requests, and sends/receives. **Client/Server examples**: telnet/telnetd, ftp/ftpd (sftp/sftpd), Firefox/Apache. **Socket**: network programming abstraction for communicating among processes (applications) based on (Unix) file descriptors. **File descriptor**:an integer representing an open file managed by the OS \In Unix any I/O is done by reading/writing from/to file descriptors. **Socket types**: Stream socket:java.net.ServerSocket, TCP based, Ordering guaranteed, Error-free \Datagram socket:java.net.DatagramSocket, UDP based \IPv4 & IPv6




## NIO(Nonblocking sockets)

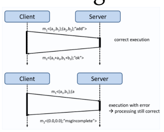
**Synchronous**: Single thread reading data from clients(stream) and blocked until ready(no multiple read) **Asynchronous**: Single thread reading data from clients: Thread → Channel: read data into buffer, Channel → Buffer: fill data into buffer, Thread → Buffer: check data in buffer (main thread not blocked) **Synchronous vs. Asynchronous**: S: A thread enters into action and waits until I/O is completed \Limited scalability, one thread per I/O connection(Overhead:context switching → time between diff. tasks) A: Passes the request immediatly to the OS-kernel and then do other tasks → worker thread `while(true){` only do computation, never blocked, no context swtich **Java NIO Channels**: All IO operations can be done with channels(File, TCP, UDP) \Multiple types of channels(FileChannel (File on disk),DatagramChannel (UDP), SocketChannel (TCP, support concurrent read/write), ServerSocketChannel (TCP)) \Responsibilities(Read, write buffer)

**U1** Finite state machines that describe **a communication session between a client and a server**. The first FSM represents the server and the second FSM represents the client. Both parties (client and server) keep the communication session open and exchange messages until one of them decides to close



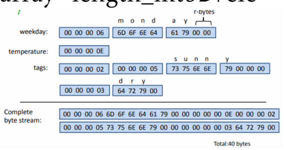

it Figure 1.1: FSM for server. Figure 1.2: FSM for client.

*accept* is `while loop`, detail in rectangle: create a new socket (and therad) for commu. with client **Simple protocol design** complex number as string: $c_i = (a, b)$, $op \in \{add, sub, mul, div\}$, C to S message format: $m_1 < c_1; c_2; op >$, Status: $st \in \{OK, msgIncomplete, ...\}$, S to C message format: $m_2 < C_r; st >$




# 3 C2 EXTERNAL DATA REPRESENTATION, Presentation Layer

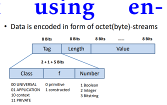
**Heterogeneity** HW: Diff. HW architectures store bytes:Big, Small Endian ProgrammingLanguage:Diff. PL store data types differently:AB, 0AB **Transformation between representations**: Transformation between local and remote representations|Information may lost **Two realizations**: 1.**Pairwise transformation between** $n$ **local representations**(vollständigGraph,$\#n^2 - n$, Either sender or receiver has to transform) 2.**Transformation to and from canonical representation**(a single canonical $C$ as intermediate representation|No local information about communication partner needed| $\#2 * (n-2)$, $-2$if canonical is one of $n$) **XDR** partOf NFS, OSPresentationLayer, encodes only data items, no meta information about their types +:easy, -:Receiver lost data description |exactly 32 bit integer is stored according to **big endian** +:Fixed length reduces computation. -:wasting |Data is encoded into blocks of multiples of 4: n-bytes contain data; r-bytes are used for padding with n + r mod 4 = 0 |int: int32; float=Sign+Exponent+Mantissa, String=length_int32+bytes, array=length_int32+ele



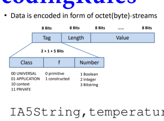

```
struct forecast:String weekday;int temperature ;String tags<>;
```

**ASN.1**:Abstract description of data types,telecommunication,internet protocol |Enables exchange in heterogeneous systems

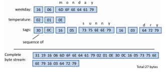
|**abstractSyntax** $\xrightarrow{compiler}$ **concreteSyntax Java, C++, they transfer syntax using en-**



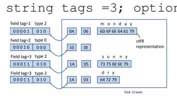

**codingRules**




```
Forecast::==SET{weekday IA5String,temperature Interger,tags SEQUENCE OF IS5String;
```

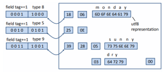


|encodes type information, +:receiver not need to know data description,-:additional overhead **Java object serialization,JOS** Stream-based transmission of serialized objects(Via TCP or UDP sockets) , Receiver of object needs implementation of class , Serialization does not require class specific code(Java reflection) , Class implements java.io.Serializable interface -: locked into Java(No support for heterogeneous systems), No support for versioning(If the serialized class changes, all network nodes have to be updated) **serialize:obj2bit**`Socket s = new Socket ("localhost", 8022);ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream ()); oos.writeObject(obj);` **deserialize:bit2obj**`ServerSocket ss = new ServerSocket(8022);Socket s = serverSocket .accept();ObjectInputStream ois = new ObjectInputStream(s.getInputStream()); obj=(Obj)ois.readObject();` **XML**De facto standard for data exchange |Schema: `<xsd:element name ="forecast"> <xsd:complexType> <xsd:all > <xsd:element name="weekday"type="xsd: string"/> <xsd:element name="temperature "type="xsd:integer"/> <xsd:element name ="tags">`<xsd:complexType> <xsd:sequence> <xsd:element name="tag"type="xsd:string"maxOccurs="unbounded"/> </xsd:...> |`<forecast> <weekday>monday </weekday> <temperature>14</temperature> < tags> <tag>sunny</tag> <tag>dry</tag> </..>` **JSON** human-readable text to transmit data objects |`{"forecast":{"weekday":"monday","temperature":14," tags":["sunny","dry"]}}` |+ **XML/JSON**:readable, defined as standard, JS support JSON(directly loaded Browser and deserialized) |- **XML/JSON**: verbose, badPreformance, longOverhead, slowWriteParse **ProtocolBuffers-Google**: Similar concept like ASN.1, but not standard, efficient binary serialization, heterogeneous systems **data structures defined in .proto file(IDL) generate serialization code(Java, C#), then java, .NET projects request, response each other** |DataModel.proto: `message forecast{required string weekday =1;required int32 temperature =2;repeated string tags =3; optional float price=4;}`



|RPC ServiceInterface.proto: `enum Status{OK=0;EXISTING=1;NOT_EX=2;ERROR=3; message SearchRequest{required String attribute =1;required String value =2; message CustomerList{repeated Customer customers =1; service AdministrationService{ rpc CreateCustomer (Customer)return (Status) ; rpc DeleteCustomer(Customer)return (Status ); rpc SearchCustomer(SearchRequest)return ( CustomerList);`

|+:efficient writing/parsing,well documented,Versioning -:No RPC **ApacheThrift:framwork, applied Hadoop and HBase** .thrift: `struct Forecast{1: string weekday 2: i32 temperature 3: list<string> tags}`



|+:Multiple protocols to serve different purposes(binary, JSON),RPC,Open source,widely,Versioning **VariableLength:**

# 4 Chap3-PRC/SessionLayer

**Remote Procedure Call**: Call non-local procedure on remote machine, like local call |Hidden: Definition of message and content types, Marshalling/unmarshalling of parameters, Sending/receiving messages **Sequence, COMU** 1.Client RPC, issues request to a client stub (a proxy object). |2. Client stub encodes parameters(marshalling, Chap2) |3. Given a network address(LAN, Internet), the client stub calls the server stub via RPC. The RPC can be executed using a RPC library,a socket connection or another protocol. |4. The server stub receives the RPC. |5. The server sub decodes the parameters(unmarshalling) and calls the local procedure on the server. |6. Server executes RPC. |7. Local procedure $\xrightarrow{result}$ server stub $\xrightarrow{result}$ client. **Parameter** CallByValue: easy forRPC |CallByRef: notForRPC (client,server different address spaces) → call-by-copy (serialize objects with Thrift, Protocol Buffers) **Binding** StaticB: Hardcoded reference to server |Simple and effective, no additional infrastructure |Client and Server tightly coupled ||DynamicB: Relies on Name and directory services to locate servers |Cost is in additional infrastructure, protocol and registration primitives |in 3. C requests service, executes RPC with S address. NameService provides S address (S registed service at directory) **RPC Sockets**in 3: RPC call, p2p comm, connect S port. S accept connection. **Error** 1.Lost-Request(longer than Timeout of C,resend $\xrightarrow{SequenceNum}$ NoDuplicates) |2.LostReply(S caches result, resend back whenDuplicates. If reply acknowledged,delete cache) |3.ClientCrash(S waits ACK forever, orphan. C restart $\xrightarrow{SessionCounter}$ old reply not interfere with new requests) |4.ServerCrash(C notKnow RPC executed or not. S logs keepingstate of the procedure → noMultiExecutions) **FailureSemantics** 1.Maybe (No repetition,Simple and efficient,No guarantee for success) |2.AtLeastOnce(Repetition after timeout,No identification of duplicates, Acceptable for idempotent operations) |3.AtMostOnce(Identification of duplicates:sequenceNum, No repetition, Acceptable for non-idempotent operations, No result when S crash) |4.ExactlyOnce(State of procedure is recorded, Only possible with transactional processing → Atomic execution, After S crash operation can be recovered and executed exactly once)

**Remote Method Invocation RMI** Object-oriented RPC, Procedure call on remote object, Method parameters can be send two ways (call-by-value →Implement Serializable, call-by-reference → extend Remote interface), Object reference identifies remote object **RMIvsRPC**RPC(Procedures of remote processes are called,Service interface provides set of procedures) |RMI(Objects in different processes communicate with each other, Remote interface specifies methods of an object)

# 5 C4-Naming

**NamingService**

DNS(DomainNameSystem) → url to IP of host serving this domain |A world-wide distributed database of name servers

# 6 WebService

**SOAP**Simple Object Access Protocol Message format: Envelope(Enclosing entity of a message,Defines namespace) Header(Contains metadata for the body,Many WS-* extensions add additional information here) Body(Contains the payload, Further specifications define the body structure) **RPCvsDocumentStyle** SOAP message is constructed in a specific way, call the Web service just like a normal function, Body of message contains parameters and method name as wrapper element,marshalling/unmarshalling is part of the standard |contains no restrictions, Message body is a XML document, C/S handles the marshalling/unmarshalling

## 7 C5-Messaging&Queuing

**MessageQueuingPattern**data shared across applications on different platforms, MQP transfer packets of data (messages) frequently, immediately, reliably, exactly once, and asynchronously, using customizable formats (Enterprise application integration EAI) |MessagePassing: message(receiver+sender+type+payload) queuing without message buffering `send(message); receive(message); callback();` |**MessagingvsRMI** One interface (per messaging product) with generic operations, Lower-level of abstraction than remote invocations, Flexible in that messaging allows arbitrary interaction patterns between sender and receiver, Sender is not blocked after sending, Can emulate request/reply pattern, Asynchronous behavior more difficult to use and debug ⇔ Interface required and known, but differs across apps, Programming model resembles non-remote calls, Realizes request/reply pattern, Sender blocked until reply arrives, unless there is an error, As compared to non-remote call, more potential for failures |**MessagePassingProperties** Reliability(SequenceNum, Ack, Timeout and re-transmission→Message loss, Check-sums, Error correcting codes (redundancy in transmission to reconstruct),Repeated sending → Message Corruption) ordering(message arrive in a specific order: FIFO,Causal,Total), Synchronous vs. asynchronous sending Synchronous vs. asynchronous receiving Buffering of messaging → Queuing |**SynchronousComm** Sender blocked until receive on receiver-side completed, Receiver blocked until sendon sender-side completed, Syn equalizes sender/receiver speeds → Deadlock, **+:** Sender knows that message was received, Only single message needs to be buffered, Synchronizes sender and receiver operating at different speeds **-:** Sender and receiver are coupled (i.e., need to be up at the same time), Sender and receiver are blocked, which reduces potential for parallelism and potential for deadlock |**AsyCom** Send not blocked; MW buffers message on sender-side or on receiver-side, SenderBuffer enables to temporarily send faster than message transmission, Receiver buffer enables to temporarily receive more messages than can be processed, Buffers temporarily mitigate speed variances of sender and receiver, but not permanently! **+:** Sender and receiver are loosely coupled, not same time More potential for parallelism than in synchronous case **-:** Sender does not know if or when a message was received, Buffers full, Dev. Debug. complex |**MessageBuffering** BufferOverflow send buffer full → Sender blocks, A buffered message is over-written, Exception, error returned to sender |receive buffer full → Message dropped, A buffered message is over-written, A NACK is returned to MW at sender. |**Sliding Window ProtocolTCP** prevent receiver buffer to overflow: sending side may only have a set number of unacknowledged messages (the sliding window): Receiver sends out ack, SW size is determined statically or dynamically Window size < buffer size (receiver buffer never overflows), Window size > buffer size( Potential for higher throughput but risk of buffer overflow ,Use flow control (dynamically adjusting the window size)) |**Queuing**:message passing + message buffering |**Decoupling Ser,Rer** → indirect and asy compared(via Callback = deque) Space de.(Ser not know Rer, but know Queue), Time de.(Ser Rer not same time), Flow de.(Ser not blocked sending→non-blocking enqueue) |**MessageQueuingPatterns** OneToOne, OneToMany, ManyToOne, ManyToMany |**QueueManager** Specialized component providing queuing functionality to apps administrative functions(Creation and deletion of queues, Starting and stopping of queues, Altering properties of existing queues,Monitoring of performance, failures, and recoveries) , Often queue managers can be configured to forward messages to other queue managers to form a network |**TransactionQueue**recovery from failures,ACID(atomic,consistent, isolated durable) Enable local messaging transactions , Group a set of consumed & produced messages into an atomic unit of work , A transaction either commits (succeeds) or aborts (fails) , Messages are actually received/sent if transaction commits , Producer side(Produced messages are retained until commit, If transaction aborts, messages are discarded) , Consumer side(All consumed messages are kept until commit, If transaction aborts, messages are re-delivered) Sending of a request and receiving a corresponding reply cannot be part of a single local messaging transaction |**Request/Reply Queue,not atomic**Request with correlationID used to match with reply at

C C $\xrightarrow{1.Req.enq}$ ReqQue $\xrightarrow{2.Req.deq}$ S $\xrightarrow{3.Req.process}$ RepQue $\xrightarrow{4.Rep.deq}$ C

|**Request/Reply TXQueue** local transactions:1.C RequestTX: C enqueues request.2. S TX: Server dequeues request, processes request, and enqueues reply 3. C Reply TX: C dequeues reply |C RequestTX once committed: Request processed exactly-once, Reply processed at-least-once |C ReqTX{Start enqueueRequest Commit} → Req.Que → S TX{Start dequeueRequest enqueueReply Commit} → Rep.Que → C RepTX{Start dequeueReply Commit}

C checks queues(Request is either in request queue or Request is currently processed by the server or Reply is in the reply queue) , Actions taken in case of abort(If Client Request TX aborts, enqueuing of request is undone, If Server TX aborts, dequeuing of request and enqueuing of reply is undone, If Client Reply TX aborts, dequeuing of reply is undone) |**C crash** 1. C ReqTX not commit $R \neq R_e$ → No message in any queue, old $R_e$ in QM 2. C ReqTX committed but S TX not → Req in ReqQue or being processed by S, 3. S TX committed but C RepTX not → Reply in RepQue(nonempty), 4. C RepTX committed $R \neq R_d$ → No message in any queue.successful Persistent storage at C and queue manager required: C marks each req with ID, QM stores IDs of the last enqueued request $R_e$ and of the last dequeued reply $R_d$, updated after commits. R:ID of the most recent request by the C (stored by the client in local persistent storage): C processes reply, if reply queue is non-empty (R = $R_e$ but QM's and C's Reply IDs do not match) , C checks reply queue for reply and waits (if necessary) **Distributed TXQueuing** Two-Phase Commit(2PC):Only atomicity

1.Prepare (to commit, voting phase) (Each participant votes to commit or to abort the TX, Once a participant has voted to commit, it can no longer abort the TX unilaterally) 2.Commit(completion phase) (Participants actually commit, after consensus has been reached that all participants are prepared. Otherwise, all participants abort) **Code** `ChannelFactory cf = new ChannelFactory();Producer p;Consumer c;` init(): `cf.setHost("broker.tum.de");cf.queueDeclare("req\_q", Persistent); cf.queueDeclare("res\_q", Persistent); p = cf.newProducer("req\_q"); c = cf.newConsumer("res\_q");//for server change p and c, C S different init()` **RRQueue** Client: send(byte[] msg): `p.enqueue(msg);` byte[] receive(): `return c.dequeue();` |Server: run() `while(true)byte[] req = c.dequeue(); try Database.store(req); p.enqueue('ACK'.toBytes())catch(e)p.enqueue('ERROR'.toBytes());` **Correlation** Client: `Hashmap<Guid, byte[]> intermediate = new Hashmap<Guid, byte[]>(); Hashmap<Guid, Correlation> correlations = new Hashmap<Guid, Correlation>();//two also for Broker` run(): `Thread.start(new Thread()void run()while(True)Delivery d = c.dequeue();/*at broker responseConsumer.dequeue();*/if(delivery != null)Guid g = d.getGuid(); if(intermediate.containsKey(g)Correlation cor = new Correlation(g, intermediate.get(g),d.getMessage()); intermediate.remove(g); cor.add(g, correlation);/*at Broker clientQueues.get(guid.getClientName()).enqueue(g, cor.toBytes());*/` send(byte[] msg): `Guid g = Guid.generate(); p.enqueue(g,msg); intermediate.add(g, msg);//not in Broker C` |Server: run() `while(true)Delivery req = c.dequeue(); try Database.store(req); p.enqueue(req.getGuid(), 'ACK'.toBytes())catch(e)p.enqueue(req.getGuid(),'ERROR'.toBytes());` **Broker** Client: `String clientName;cf.queueDeclare('request_queue_broker');cf.queueDeclare('client_queue_'+clientName);` void receive(): `Delivery d = c.dequeue(); if(d != null)Guid g = d.getGuid(); Correlation cor = new Correlation(); cor.fromBytes(d.getMessage());` |Broker: `List<String> clients = new ArrayList<String>(); List<String> servers = new ArrayList<String>(); Map<String, Producer> serverQueues = new HashMap<String, Producer>(); Map<String, Producer> clientQueues = new HashMap<String, Producer>();` init(): `clients.add("c1"); servers.add("s1");/*multi add*/cf.queueDeclare("request_queue_broker", Persistent); requestConsumer = cf.newConsumer("response_queue_broker"); cf.queueDeclare("response_queue_broker", Persistent); responseConsumer = cf.newConsumer("response_queue_broker"); for(String server : servers)/*also for client*/String queueName = "server_queue_"+server; /*clinet_q*/cf.queueDeclare(queueName, Persistent); serverQueues.put(client, cf.newProducer(queueName));/*clineQueues*/` run(): `Thread.start(new Thread()void run()int count = 0; while(True)Delivery d = requestConsumer.dequeue(); if(d != null) intermediate.add(d.getGuid(), d.getMessage()); serverQueues.get(count\%servers.size()).enqueue(d.getGuid(),d.getMessage()); count++; //with Func Cor Client run` |Server `String sName;` run(): `while(True)Delivery d = c.dequeue(try Database.store(request); p.enqueue(d.getGuid(), 'ACK'.toBytes()); catch(e)p.enqueue(d.getGuid(), 'ERROR'.toBytes());` |**TXQueue** Client: `Queue<byte[]>messageBuffer = new Queue<byte[]>;Queue<Delivery> resultBuffer = new Queue<Delivery>;` request(): `guid = Guid.generateRandom();file = newFile('path/requestFile');file.write(guid); startTX message = messageBuffer.get(); producer.enqueue(guid, message); commitTX` reply(): `startTX Delivery delivery = consumer.dequeue(); resultBuffer.put(delivery); commitTX;` recovery(): `file = newFile("path/requestFile"); Guid R = file.readLastEntry(); Guid Re = cf.getLastEnqueued("request_queue"); Guid Rd = cf.getLastDequeued("reply_queue"); if(R != Re)requestTX()else if(R != Rd)while(cf.getQueueSize('reply_queue')== 0)Thread.sleep(500); replyTX()else /*nothing*/spawn(while (true))` |Server: `processTX startTX Delivery delivery = consumer.dequeue(); producer.enqueue(delivery.getCorrId(), 'ACK'); commitTX;`

## 8 7QueuingTheory

ArrivalRate$\lambda = \frac{1}{meaninter-arrivaltime}$ |ServiceRate $\mu = \frac{1}{meanservicetime}$ |Q.Formation $\equiv \lambda > \mu$ |Stability $\equiv \lambda < \mu$: busy,idle alternating |Throughput:average # completed jobs per unit of time → 0(stable) to maximumThr.(unstableQ.,best performance) |ServerUtilization $= \lambda * \mu$: S busy time percent |A/S/n: arrival process/ service process/# S |A,S:M (Markov,Exponential probability density), D (Deterministic, All customers have the same value), G (General, Any arbitrary probability distribution) |M/M/1: Infinite population of customers, Infinite q.capacity, FIFO