1 Introduction

Enterprise SY architecture:Cs, Web S, APP Ss, MsgQueue, Pub/Sub SY, DBs

Distributed APP developed on sockets. (Fallacies, Interoperability in spite of heterogeneity, Data representation and encoding, Parameter passing convention calling remote procedures, Atomic execution of TX, Enable APP integration, Data persistence)

Middleware as layer between transport & application layer ⇒ Above OS and below APP. comprises services and abstractions(RMI, Messaging, PubSub,TX,Naming) that facilitate the design, development, and deployment of distributed APP in heterogeneous, networked environments. Deals with interoperability 互通性 , SY integration |C request/response ↔ Middleware ↔ S. |All Computers share single distributed APP and Middleware.

Heterogeneity: Computer architecture(big, little endian), OS(Comm. sub-system), Host and network representation of data, Programming language(Representation of characters)

-ilities: Reliable; Fault-tolerant; Highly available; Recoverable; Consistent; Scalable; Predictable pref.; Secure; Heterogeneous; Open

8 Fallacies 1.Network Reliable Assumptions:Power Supply, Hard Disk, Node Failures, Config., Bugs Effect:APP hangs, crashes Countermeasures: Redundancy HW&SW systems,middleware &application; Catch Exceptions; Check Codes, React; Retry After Timeouts; pos. neg. ACK; Identify & ignore duplicates; idempotent operations 2.Latency延迟 Is Zero time For Data Transfer(speed Of Light); Bandwidth:how Much Data transferred (bit/s) |Local Call: Push and a Jump to Subroutine; SY Call: OS, 100s of assembly; Call across LAN: SY calls on caller + callee + network latency; Call across WAN: transmission delay 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

Standardization: Official:ISO, ITU, DIN, Semi-official: IEEE, W3C, OpenGroup

2 Comm. Basics

ISO OSI: Open Systems Interconnection model, Basis for standards development on SY interconnection, Reference model. |Layers: Application(Peer protocol: HTTP, DNS, FTP); Presentation; Session; Transport: TCP/UDP; Network: IP; Data link; Physical

IPv4 v6(no checksum): Relays datagrams accross networks, Routing enables internetworking, Deliver packet from source to host address, Foundation for TCP/UDP

IP routing:routing protocol, BGP used in internet: Border Gateway Protocol, Routing between AS (Autonomous systems, provider or bigger organization), Exchanges routing and reachability information between AS.

TCP: for HTTP, RPC, Slower than UDP, but with reliability using ACK, Connection oriented protocol, session is initiated, Provides ordering, sequencing, Flow control, sender can't overflow receiver(same speed sending, receiving), PortNum in TCP protocol(not in IP protocol)

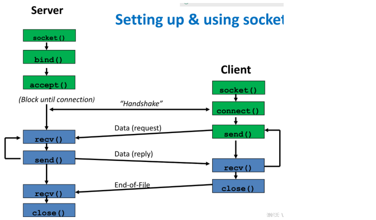
UDP: for DNS, Video, Voice, Faster than TCP, Connectionless(no session); Best-effort; Packet independent; No guaranteed delivery; No ordering guarantees; P2P and P2-multipoint.

Ports: a 16 bit number to local host to identify the connection, to differentiate APPs, if packet arrive; Separate for UDP and TCP; $0-1023$ reserved to root, $1024-65535$ available to regular user; http 80/tcp, ftp 21/tcp, ssh 22/tcp, telnet 23/tcp, finger 79/tcp, snmp 161/udp

APP Layer protocol: Set of rules specifying data transfer between computing end-points (Connection establishment & tear-down Data representation, Comm); DHCP (Dynamic Host Configuration Protocol), HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), Telnet (Telnet Remote Protocol), SSH (Secure Shell Remote Protocol), SIP (Session Initiation Protocol), POP3 (Post Office Protocol 3), SMTP (Simple Mail Transfer Protocol), IMAP (Internet Message Access Protocol), ||browser retreives web page(HTTP 1.1 over TCP, HTTP, FTP, SMTP)

HTTP/1.0 C S: Request: GET <path>/index.html HTTP/1.0 Response: HTTP/1.0 200 OK ErrorResponse: HTTP/1.0 404 Not Found ||HTTP/1.1: HyperText Transfer Protocol, Request/Response protocol for C S comm. on top of TCP (Browser, RESTFUL API's, SOAP over HTTP, NoSQL databases) Methods: GET:cacheable get info by Request-URI HEAD: like GET but MUST NOT return a message-body POST: post a form (bulletin board), uploading data, can be a data-accepting process Responses not cacheable(200 (OK), 204(No Content), 201 (Created), 303 (See Other)) PUT: Enclosed entity to be stored under Request-URI Responses not cacheable DELETE: Delete the resource by Request-URI Response: 200 (OK), 202 (Accepted), 204 (No Content) TRACE: for diagnostics CONNECT: to initialize secure connection, HTTPS on port 443, Proxy is asked to forward the TCP connection |HTTP/1.1 proxy:C GET req. → Proxy, but not in cache. Proxy GET req. → Origin, Origin → Proxy response. C GET req. → Proxy, Proxy → C cached resp. |HTTP/2.0: Better utilization network capacity, Headers compressed, On a single connection req and resp interleaved, Prioritization of requests GET/HTTP/1.1 Host:www.tum.de Connection:keep-alive Accept: text/html,application/xhtml+xml,application/xml;q=0.9, image/webp, */*;q=0.8 User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.120 Safari/537.36 Accept-Encoding: gzip,deflate,sdch |||HTTP/1.1 200 OK Date: Wed, 08 Oct 2014 15:25:53 GMT Server: Apache X-Powered-By: PHP/5.5.17 Content-Encoding: gzip

Socket
Move data/Msg/(invoke operation/service and return result/failure) from APP I on Host A to APP 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 participants (client and server) keep the communication session open and exchange messages until one of them decides to close it

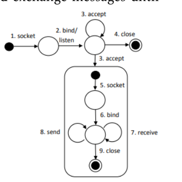


Figure 1.1: FSM for server.

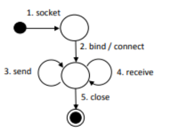
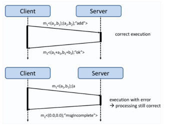


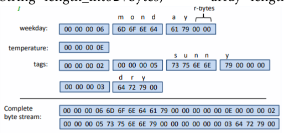
Figure 1.2: FSM for client.

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: $m_2 < C_r; st >$

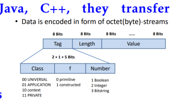


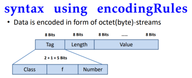
3 C2 EXTERNAL DATA REPRESENTATION, Presentation Layer

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| $\#2*(n-2)$, -2if 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

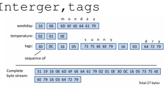


struct forecast:String weekday;`int` temperature; String tags<>; ASN.1:Abstract description of data types,telecommunication,internet protocol |Enables exchange in heterogeneous systems |abstractSyntax $\xrightarrow{compiler}$ concreteSyntax Java, C++, they transfer syntax using encodingRules

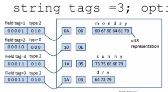




Forecast::==SET{weekday IA5String,temperature Interger,tags

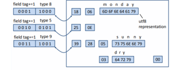


SEQUENCE OF IS5String; |encodes type information, +:receiver not need to know data description,-:additional overhead Java object serialization,JOS Stream-based transmission of serialized objects(Via TCP or UDP sockets) , Receiver of object needs implementation of class , Serialization does not require 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:obj2bitSocket s = `new` Socket (`"localhost"`, 8022);ObjectOutputStream oos = `new` ObjectOutputStream(s. getOutputStream()); oos.writeObject(obj); deserialize:bit2objServerSocket ss = `new` ServerSocket(8022);Socket s = serverSocket .accept();ObjectInputStream ois = `new` ObjectInputStream(s.getInputStream());obj =(Obj)ois.readObject(); XMLDe 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 ProtocolBuffersGoogle: 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:framework, 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 stub 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 Call-ByValue: 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.LostRequest(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 RMIvsRPCRPC(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

SOAPSimple Object Access Protocol Message format: Envelope(Enclosing entity of a message,Defines message format) 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

MessageQueuingPatterndata 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 message product) with generic operations, Lower-level of abstraction than remote invocations, Flexible in that message passing 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, Checksums, 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 |TransactionQueuerecovery 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 atomicRequest 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 |Exponential distribution: $f(x) = \lambda e^{-\lambda x}$, mean$= 1/\lambda$ |Poisson arrival process(arrivalProcess): $P_n(t) = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$, |Traffic intensity (occupancy): $\rho = \lambda/\mu$, n customers: $Prob[n] = \rho^n(1-\rho)$, empty: $Prob[n=0] = 1-\rho$ |Mean # customers: $N = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu-\lambda} = L$ |Total waiting time (including service time): $T = \frac{1}{\mu-\rho}$

9 S-Side Architecture

Thread: has own Stack, Instruction pointer,Processor State(Register). multiple included in proc. |Process:running program as set of Threads(Binary image loaded into memory, Instance of virtualized memory, Resources as open files, sockets, Security context as associated user, Thread as Unit of activity of process) |Single-threaded Pro. One instance of virtual memory |Multi-threaded Pro.: One instance + th. share same memory address space(ob) |Synchronisation:sharing same address(obj), Read/Write access must be synchronized(Mutex, Locks, Semaphores) Lock:`Lock l; l.lock(); /*critical section, can be Deadlock*/l.unlock();` Mutal Exclusion:Critical sections of th.s not overlap Freedom from deadlock, Freedom from

starvation |**Th. State** Created → Waiting(Scheduled or unscheduled) ↔ Swapped outWaiting(No enough memory), Running → Terminiated, Blocked(Blocking IO) → Waiting(IO finish) Blocked ↔ Swapped out andBlocked(No enough memory) |**API**: `public class WorkThread extends Thread{ public ConnectionHandleThread(Parameter params) @Overridepublic void run(){Computation(); /*usage*/Thread t = new WorkThread(params) ; th.start(); Thread.sleep(1000);/*current thread */th.getState();/*NEW, RUNNABLE, BLOCKED, WAITING, TERMINATED, TIMED_WAITING */th.setPriority(para); th.yield();/*give current use of processor to other th.*/`
|**Amdahl's law**: maxium theoretical speedup using multiple processors $T(n) = T(1) * (B + \frac{1}{n}(1 - B))$ n:# processor, $B \in [0,1] serial, nonparallizable,$ speedup:$S(n) = \frac{T(1)}{T(n)} = \frac{1}{T(n)}$ |**Overload**: 1.Transient ov.(Short spikes in requests which ov. the S temporarily) 2.Continous ov.(Incomming rate exceeds the capacity of the S) 3.Methods to tackle ov.(Refuse requests after certain queue length,Add capacity) |**Concurrency Models** 1.**Thread-based conc.** (One request is completely handled by a single th. time of request, same th. is responsible until the response is returned If request is blocked (due to a DB access), the th. is blocked 2.**Event-based conc.**not th. based Request in Network/HDD→EventHandler→# FSM = # Steps(th.s)→Response (–: Only non-blocking(async) I/O operations used, Code less modular,nonreusable, Complex control flow(FSM, flags), debugging OS support like async I/O libraries) (A th. processes a request until th. blocked. When th. blocked, the th. continues with the next requests, Once a th. becomes unblocked, it gets enqueued in the list of ready tasks. Scalar code separated in micro tasks. Finite State MachinesFSM(Java NIO)to track state) C socket created → th1 accept connection register 'C socket event' |requests from C socket → th2 lookupDB register 'query done event' |QueryDone → th3 send 1MB to client register 'data sent event' |data sent → th4 Cancel connection |**Single-threaded S = Iterative S**: S only single worker th.(do all also I/O), Thread-based conc. Waits till a new request from Network is in input queue, if empty ,Dequeues the next request ,Computes the result of the request ,Generates response to Network ,When blocked due to I/O, thread also blocks(I/O read disk to memory) ++:Simple programming, cache locality(one request in cache) –:Insufficient use of resources, Limited scalability(reject requests if too many) |**Multi-threaded S**: S with multiple worker th.(same way as single th. wait …block OS can schedule non-blocked threads(get nonblocked th. from pool), more threads than cores since a certain percentage of I/O is assumed ++: Programming abstraction(Dividing up work and assigning to a th.), Parallelism(better throughput), Responsiveness(Doing heavy operations in background, while UI stays responsive), Blocking I/O(When blocking I/O, other th. continue to work), Contex switching(Switching costs from one thread to another within the same process is cheaper than process-to-process context switching), Memory savings(Th. to share memory, yet utilize multiple units of execution) |**One th. per request/Connection**:scheduled by operations, Request in Network→InputQueue→Dispatcher (starts one th. per request,when response sent, th. ends)→Response to Network ++:Simplified programming –: Too many th. → thread-starvation(too little CPU-time) , cost of Starting/Removing th. , Not ideal for highly parallel S `ServerSocket ss = new ServerSocket(); ss.bind(new InetSocketAddress("127.0.0.1", port)); while (true)Socket ss = serverSocket.accept(); Thread t= new ConnectionHandleThread(kv, ss); t.start();` + `public class ConnectionHandleThread extends Thread { public ConnectionHandleThread(KVStore store, Socket clientSocket){... @Override public void run(){BufferedReader in = ... clientSocket.getInputStream(); PrintWriter out = ... clientSocket .getOutputStream(); String firstLine; while ((firstLine = in.readLine())!= null){String res = kv.process(firstLine);out. write(res);out.flush();` |**Abstracting Th.**: deadlock free, starvation free **Abstractions:** Thread pool, Event based concurrency, SEDA, Actor model of concurrency, Reactive programming |**Th. pool**managed collection of available th.s, size as maximum(eg number of CPU cores), Completes tasks in parallel, Reuses th. for multiple tasks **Multi-threading with th. pool** optimal size depends on # cores, # blocking of I/O operations |Request in Network→InputQueue→Scheduler(Thread pool with size)→Response to Network `ExecutorService es= Executors.newFixedThreadPool(4);while (true ){Socket serverSocket= serverSocket.accept ();es.submit(new ConnectionHandlerRunnable( kv, clientSocket));` ++ RUNNABLE above in One th. per request/Connection |**Scheduling Algorithm**:Most cache efficient to assign tasks to th.s in pool, Two main paradigms:Work sharing,Work stealing Work Sharing Algo.When th. created, scheduler moves work of other th. to new th. –:Comm. between cores, Cache misses |Work stealing algo. Under-utilized processors steal work from busy processors(The migration of threads occurs less frequently with work stealing than with work sharing) ++:Maintain th. on same CPU(data locality → better cache usage, Minimize comm. between th |**Futures**:result of asyn. computation in thread pool `/*result in future obj */FutureTask<LinearRegressionResult> future = new FutureTask<Integer>(new Callable <String>(){public Integer call(){List< Doubles> lr = getDailyTurnover(..); return calcLinearRegression(lr);}); threadpool .execute(future);` |**SEDA**Staged event-driven architecture, a network of stages connected by event queues ++: Support massive concurrency, Simplify the construction of services(Provide abstractions), Enable introspection(Adapt behavior to changing load conditions), Support self-tuning resource managment, Server is created out of many SEDA stages |**Actor**:No access to shared memory(No locks,no deadlock), Scheduled on top of threads |**Reactive Programming**: oriented around data flows and propagation of change(Dependency graph) |||**App S**: a component-based product that resides in the middle-tier of a server-centric architecture. It provides middleware services for security and state maintenance, along with data access and persistence. Java:WildFly, Websphere |**EJB**: Standard component architecture, Distributed business applications(Support development, deployment and use of web services), Write once, run in all application containers ++:Developer not to care about Fail-over,Clustering,(Distributed) Transaction handling,Databases,Security,Deployment

10 Publish/Subscribe

Many-to-many: For communication, For coordination, M: data sources (publishing C) n: data sinks (subscribing C) C decoupled and do not know each other ↔ tightly coupled Observer Design Pattern: Removes explicit dependencies, Reduces coordination synchronization, Increases scalability of dis. SY, Creates highly dynamic(frequent add/remove) decentralized SY, Decoupling in three dimensions(Space Dec. No need for C (publishers & subscribers) to hold references or know each other, C physically distributed, Time Dec.C not to be available same time(Event in Buffer) Synchronization Dec.Control flow not blocked by the interaction)

Comparison: Decoupling in Different Interaction Schemes

Abstraction	Space de-coupling	Time de-coupling	Synchronization decoupling
Message Passing point to point	No	No	Producer-side
RPC/RMI Client Server	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Notifications (Observer D. Pattern)	No	No	Yes
Tuple Spaces like event data base	Yes	Yes	Producer-side
Message Queuing (Pull)	Yes	Yes	Producer-side
Publish/Subscribe	Yes	Yes	Yes

Pub/Sub Models: Content-based, Channel-based(non-hierarchical), Topic-based(hierarchical, sport new USA) **Matching&Filtering in Content-based**: event(Publication) e, set of subscriptions S, find all subscription $s \in S$ matching e. Subscription: Boolean function over predicates Publication(event): Sets of attribute-value pairs **Two-phased Matching Algo.** 1.Match all predicates (Predicate Matching Phase) 2.Match subscriptions from results of Phase 1 (Subscriptions Matching Phase) **1.Predicate Mat.P.**: set P of predicates and event e, identify all satisfied predicates p of P → Predicate bit vector, Hash key = attribute name General Purpose Data Structure: for single attribute: 4 ordered linked list(b tree) for $=, <, >, ! =$ operators, $O(n)$ all events & lists Finite Predicate Value Domain Types: huge matrix eg $1000 * 4\ forPrice \in [1, 1000]$, 1000 for price, 4 for operators, entry lookup $m[i][j] = O(1)$ **2.Subscription Mat.P.: Counting Algo** |**Content-based Routing** 1.Advertisements (schema or types,data sources) Boradcast, 2.Publications & events (data sources)) 根据上一步Broker's Subscription走 , 3.Subscriptions (query, data sinks)

11 Service Orchestration

Business Process: set of linked activities which realise business goal Bus. Pro. vs Programs Granularity(Based on activities, Programming in the large), Control flow(Explicitly defined, Easy understand), Flexibility(change), Execution(call third-party webservices, Scheduled by process engine) VS Granularity(Based on instructions, Programming in the small) Control flow(Implicitly defined, Not easy understand) No Flexibility(Hard-coded) Execution(Invoke instructions locally, Scheduled by operating system) **BPEL, WS-BPEL**:Web Services Business Process Execution Language(XML based), to orchestrate loosely coupled services(to model SOA's Business processe), using web services running at bus. proc. execution engine (ODE), Lacks possibility for formal verification compared to FSP Components: Activities, State handling, Control structures, Exception handling, Partner links, Parallism, Dead path elimination Related Standards SOAP Simple Object Access Protocol, Comm. platform in XML WSDL Web Service Definition Language, set of endpoints operating on messages, Operations and messages described abstractly and bound to concrete network protocol UDDI Universal Description, Discovery and Integration, Services registered, published and reused by other organizations, App store for webservices

Service-Oriented ArchitectureSOA technique that involves the interaction between loosely coupled services that function independent Principles: Loose coupling, Service contract → WSDL, Abstraction of underlying logic, Autonomy, Reusability, Composability → WS-BPEL, Interoperability, Discoverability → UDDI |**BPMN vs BPEL** Business friendly, intuitive, process oriented, Swimlanes that represent organizational units, User friendly data manipulations, include human tasks, expose web service UI VS technical, manipulated with XPath expressions, xpress orchestrations, error handling, compensating actions |**Service composition** conceptual model: Servers exchanged Msg via out-port, in-port. design methodology: 1Bottom-up: Service providers develop and publish services Service consumers discover and select services Service consumers compose selected services 2Top-down: Service consumer develop a global process Service consumers decompose the global process into subprocesses Service consumers select/develop services to implement subprocesses |**Service orchestration** Local perspective, Describe control from one party's behavior(perspective), WS-BPEL as Standard, Executable processes which interact(message level) with web services (Service internal / external), Business processes(business logic + task execution order, span multiple apps and organizations, Define a long-lived transactional multistep process model) |**Service choreography** Global perspective, Describe the global interactions among all the parties, WS-CDL as Standard Tracks the message sequences among multiple source and sinks, Each involved party describes its part of the interaction |**Orchest. vs Choreo.** Two Orchest.(Web Service) sending each other(Choreo.) Request, ACK, Accept, ACK |**Formal Methods:** Modeling approaches, State machine verification, (Finite state processes (FSP), BPEL → LTS → FSP), to represent and reason about Complexity of Concurrency Sy, Performance optimization, Deadlock detection, dead path elimination |**FSP** Labeled Transition System(Abstract machine to study computation Contains a set of states and transitions between states) |**BPEL → FSP** Activitiy: <invoke partner='p1' operation='o1' /> → INVOKE = (invoke_p1_o1 -> END). <receive partner='p2' operation='o2' /> → RECEIVE = (receive_p2_o2 -> END). <reply partner='p1' operation='o1' /> → REPLY = (reply_p1_o1-> END). Sequence: <sequence> 见上面BPEL三个 </sequence> → 见上面FSP三个 SEQUENCE = INVOKE; RECEIVE; REPLY; END. $0 \to 1 \to 2 \to E$ Flow: Parallel Activity <flow> 见上面BPEL三个 </flow> → 见上面FSP三个 || FLOW = (INVOKE || RECEIVE || REPLY). $0 \to 1 \to 2 \to E \leftarrow 4 \leftarrow 5 \leftarrow 6 \leftarrow 7$ 而且每点出3种Activity |**Deadlock Detection**: when two or more competing activities are each waiting for the other to finish, in FSP: a non-final-state with no outgoing arcs, like A sends to B, B sends to A Assumption: Sync Comm $!m_1$: Send a Msg of type m, $?m_1$: Receive a Msg of type m