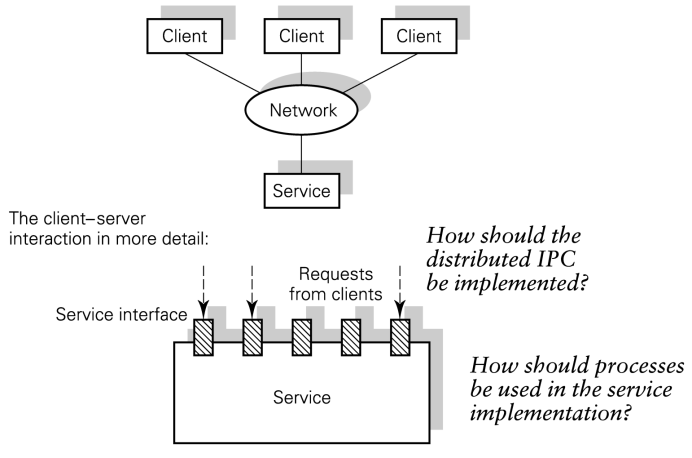
**Chapter 16: Distributed IPC**

16.1 Introduction

How should the service be implemented in terms of concurrent processes?

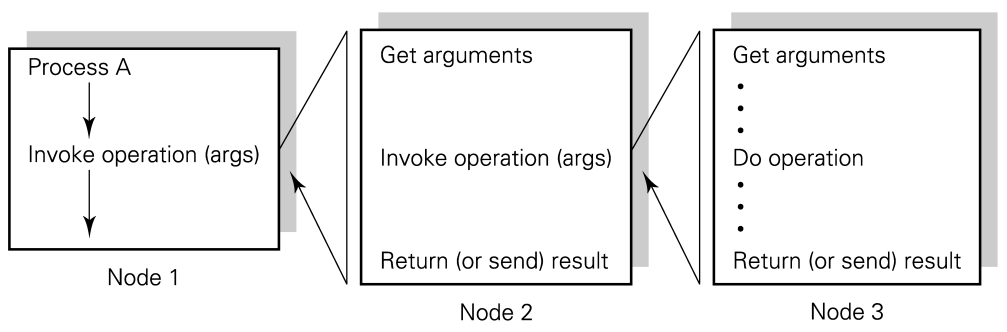
How should the distributed service invocation be implemented?

How is the IPC integrated with the lower levels of communications service provided by the OS?

16.2 Special characteristics of distributed systems

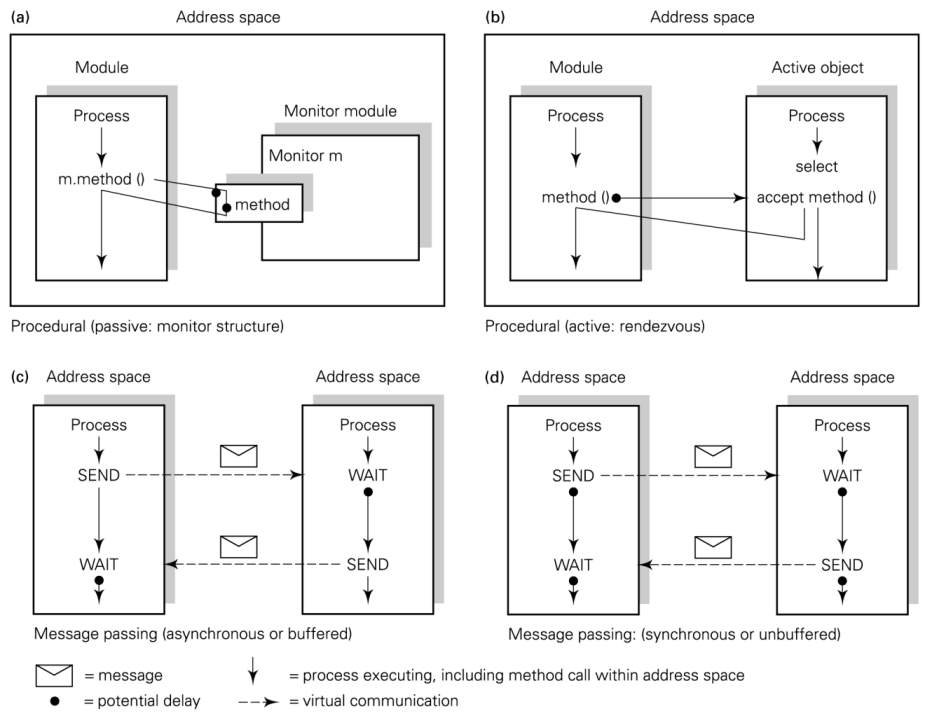
(zie ook chapter 7)

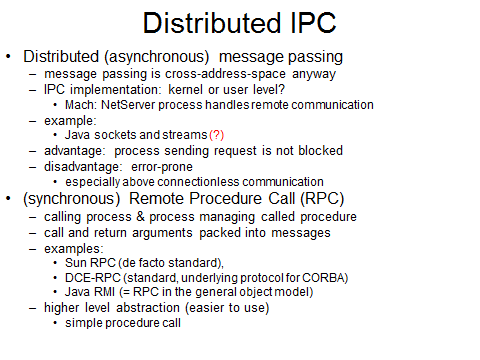
* Concurrente uitvoering van hun componenten
* Onafhankelijke failure modes
* Geen globale tijd
* Inconsistente toestand
* Vertraging in communicatie



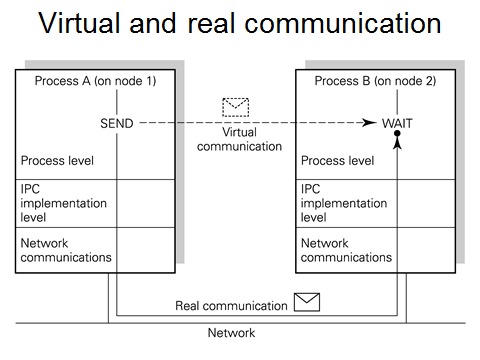
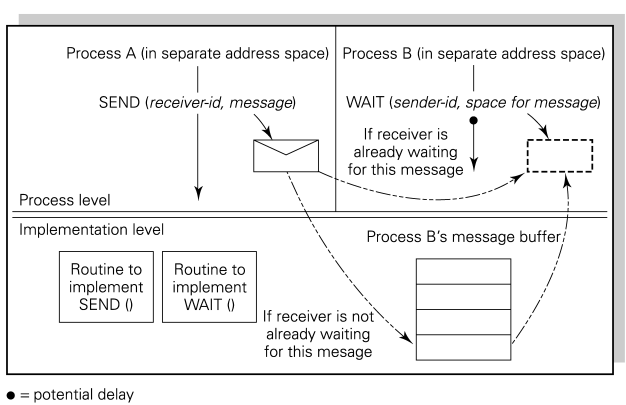
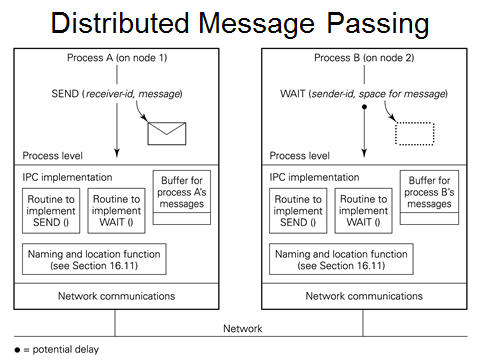
*Figuur:* 3 computers verbonden in 1 netwerk  
 een process op 1 node kan een operatie op een andere node aanroepen, die op zijn beurt weer een operatie op een derde node aanroept. Om het even welke node kan onafhankelijk  
 van de andere crashen op gelijk welk moment in de aanroepen; ook de netwerk connecties  
 tussen de nodes kunnen falen terwijl de processen nog steeds verdergaan.

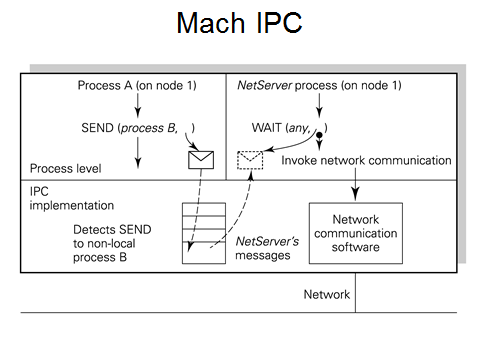
16.3 Distributed IPC: Message passing



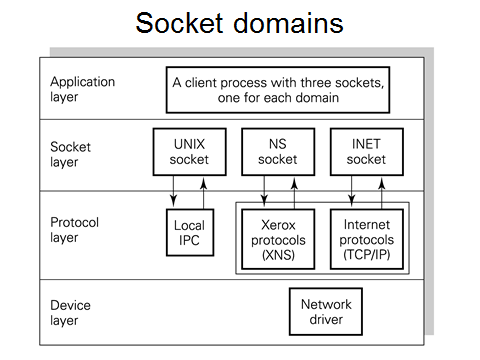
*Figuur*: samenvatting van high-level IPC mechanismes (Ch13)  
vat de stijlen voor inter-proces communicatie samen; (a) en (b): shared address space  
verspreiden van de cross-address-space IPC in (c) en (d): de adresruimtes van de communicatieprocessen kunnen zich op natuurlijke wijze op aparte computers bevinden en de messages die doorgegeven worden moeten dan gestuurd worden via network communications services.  
degree of transparency?  
\* distribution transparency (and location transparency): je moet niet weten waar server zit of dat het op een andere machine zit of niet  
\* location transparency (but no distribution transparency): weten op andere machine of niet, maar niet exacte loactie kennen  
\* no transparency at all: sockets

Typisch async message passing => zender blokkeert niet  
java? Jse heeft geen message passing  
sockets ≠ message passing  
=> via socket stuur je stroom van bytes door, ongestructureerd  
(cfr pipes: UNIX pipe = byte stream => op pipe zijn processen op zelfde systeem aanwezig)

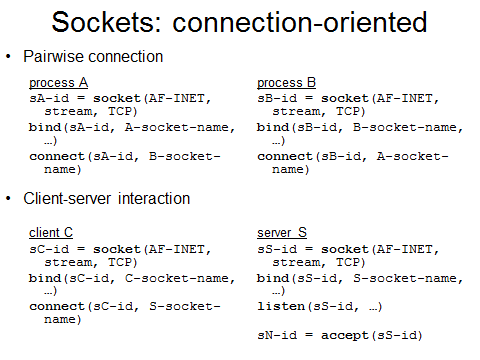
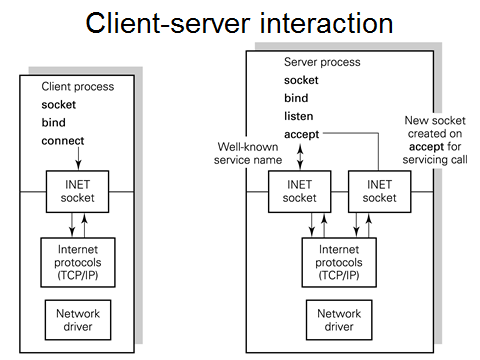
16.3.2 Distributed, async message passing  
  
=> we beschouwen gelijkaardige situatie maar met A en B nu op verschillende computers  
  
  
  
  
  
  
(ik zie processen op andere computer niet, dus kan zijn dat ik niet eens weet hoe ander proces heet  
A heeft buffer omdat deze ook berichten kan ontvangen (symm.) – zie ook Ch3)  
one approach to implementing distributed async message passing. The IPC mgmt function is extended to invoke network communication. A now runs on node 1 and SENDs a message to B on node 2. B WAITs for message from A. We assume that the IPC implementation on node 1 can detect that B is not a local process and either knows or can find out that B resides on node 2. The implementation of a SEND message on node 1 can invoke network communication software to transmit the message to node 2. The network communication software on node 2 can receive the message from the network, and pass it to the IPC mechanism for delivery to the message buffer of process B.  
fig2: higher-level view of same procedure (prev fig)  
virtual communication at the process level is implemented by real communication through the supporting levels and across the network. The basic approach illustrated here may be used to distribute any form of IPC transparently to the app. Here we have assumed that the kernel IPC implementation determines that a communication is to a non-local process and itself takes action to locate the destination.

alternative approach:  
kernel detects a non-local communication and passes the message to a local process which is responsible for all network-communication, the *NetServer* process. The IPC implementation now handles only local communication and the NetServer process acts as an indirection for remote communication. An advantage of this approach is that the kernel is kept simple. Disadvantage: extra context switching and message passing between user-level processes (A and the NetServer) that is needed to send a message to a non-local process. Incoming messages are passed by the communications software to the NetServer process, which can use a local SEND to deliver them to their intended destination.  
(IPC => process B staat niet op zelfde computer, dus message kan niet rechtstreeks naar B gestuurd worden; wordt gestuurd naar NetServer: moet uitzoeken waar B zit.  
Mach: minikernel => doet alleen wat strikt noodzakelijk is, zoveel mogelijk overgelaten aan user processen; maximale grootte message = 4GB => zou hele adresruimte in 1x kunnen doorsturen  
NetServer gedraagt zich als soort externe schijf voor ontvanger => stuurt alleen door wat ontvanger nodig heeft)

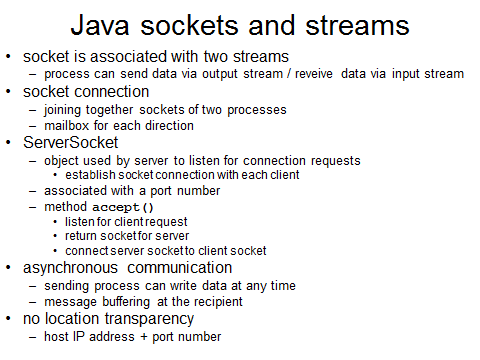
16.4 Integration of IPC with communications

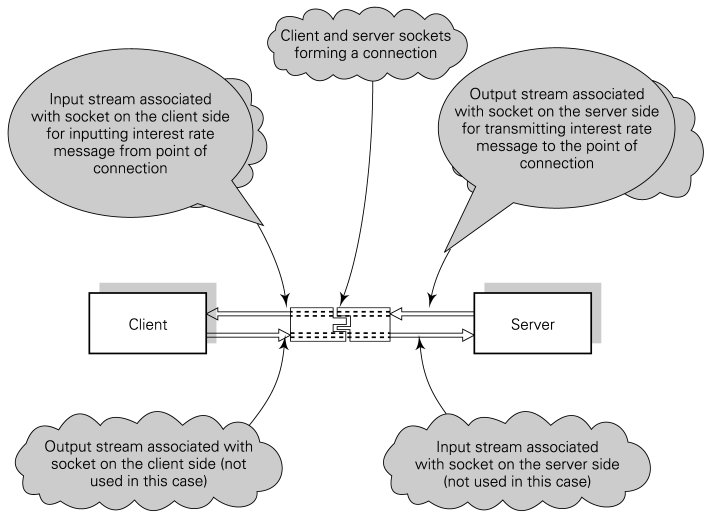
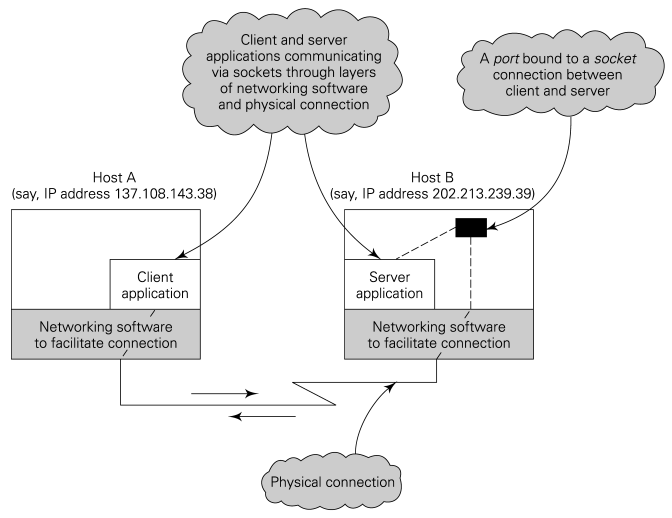
16.4.1 IPC above sockets  
socket = OS provided interface to communications service  
basic concepts at OS interface for integrating IPC with communications software:  
\* socket: an endpoint of communication  
\* domain: within which sockets are named and communication takes place. A socket operates in a single domain and the address format of the socket depends on that domain. Each domain employs a single protocol suite. => local (UNIX): socket name = file system pathname; internet (INET) with TCP/IP protocol stack: socket name = IP address (32 bits) + TCP/UDP port number (16 bits)  
several socket types:  
\* stream socket: provides reliable duplex data stream, a ‘virtual circuit’ => pipe can be implemented as a pair of sockets of this type  
\* datagram socket: will transfer a message of variable size in either direction; unreliable datagram service: no guarantee that order of sending will be preserved on receipt and a datagram might be lost  
\*…  
Operations:  
socket-descriptor = **socket**(domain, socket type, protocol)  
**bind**(socket-descriptor, local socket name, …)  
**connect**(socket-descriptor, remote socket name, …)  
**listen**(socket descriptor, …)  
new-socket-descriptor = **accept**(socket-descriptor)  
(types: onderscheid tss gestructureerde stroom en discrete blokken (enkel stroom beschouwd) reliable => als er iets fout gaat, moet TCP dit opvangen  
duplex => in twee richtingen  
speciaal geval: indien server -> listen, accept => connecteert met meerdere clients)

TCP/IP nu zo dominant dat andere network-technologiën niet meer van belang zijn  
=> we beschouwen dus enkel INET sockets

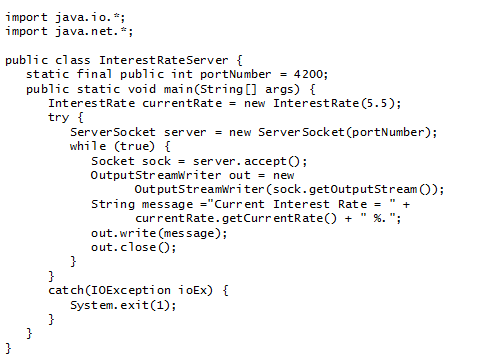
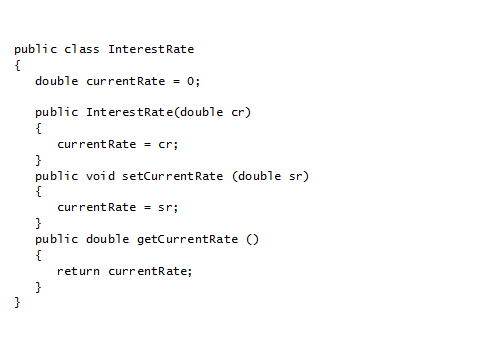
(proces A & B symmetrisch  
client C is zelfde als bij proces A  
server S: listen => ik ben bereid te luisteren  
iedere keer als client connect doet, wordt op server met accept een nieuwe socket geconstrueerd om met client te communiceren)  
pairwise connection: 2 processes which wish to communicate must each create sockets and bind names to them. When both processes have executed **connect**, the communication can proceed.  
client-server interaction: the clients will know the name of the server, but not vice versa. Two additional system calls are provided: **listen**(*socket-descriptor, queue-length*) which the server executes to tell the kernel that it is ready to accept calls for service. It also indicates how many requests the kernel should allow to accumulate. *New-socket-descriptor* = **accept** (*socket-descriptor*): allows server to take a single request for connection from a client. A new socket is created for that client.  
fig: server creates and binds a name to a socket to function as the well-known address of the service. A client connects to that well-known name.

16.5 Java sockets and streams

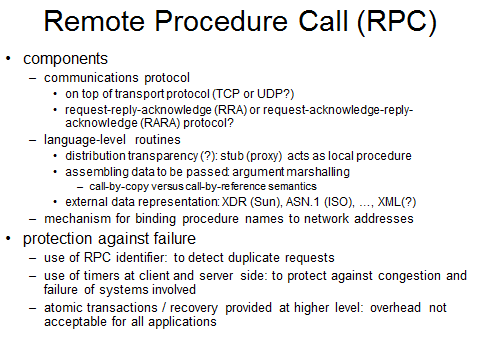


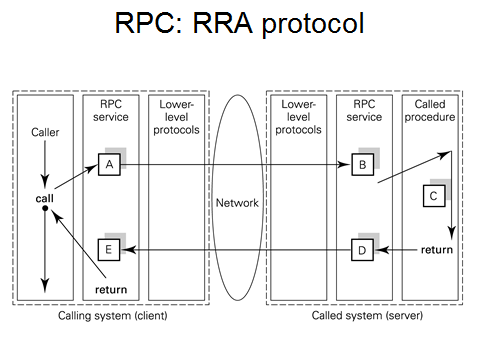
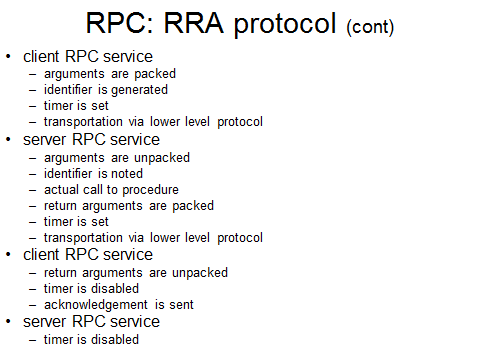


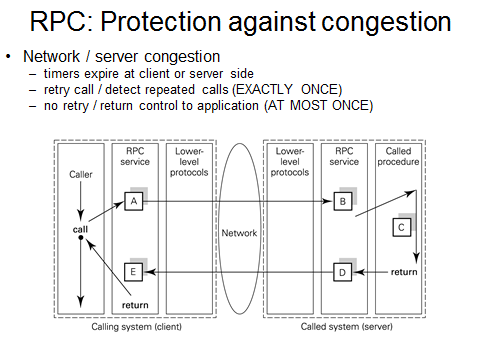
(left) client and server program (app) executing on 2 different host machines, communicating with each other via layers of standard networking software and a physical connection. Note that for each host machine, the other machine will be considered a **remote machine**. The server’s name is a port name plus its host name (either a name under which the machine is registered in the DNS of an IP address that has been assigned to it). This name is bound to the server’s socket and a two-way communication is set up between client and server. The connection provides an input and output stream for both client and server to enable data to be transferred between them. (right) gives a high-level view. The figure shows pictorially how the client and server are connected by ‘plugging’ together their respective sockets. It also indicates the way in which the message is sent from the server to the client via the two shaded streams. The server is shown using its socket’s output stream (shaded) to write the message. The client, in turn, uses its socket’s input stream (shaded) to read the message.

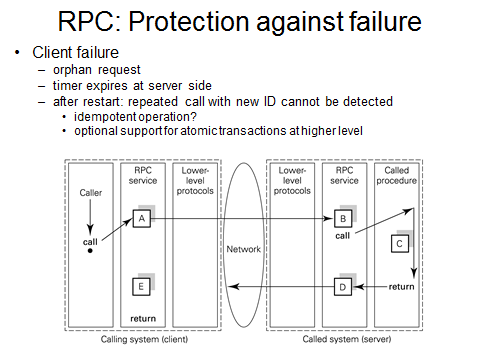
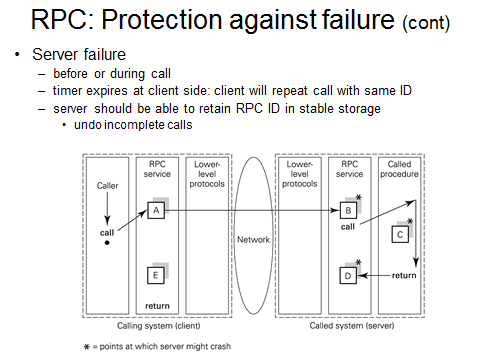
  
  
1) voorbeeld: client kan ook aan server intrestvoet vragen; setcurrentrate en getcurrentrate zijn enige 2 methodes  
2) interestrateserver creëert nieuw object currentrate (-> new interestrate)  
server gaat proberen nieuw object socket te maken opdat hij kan luisteren (moet maar 1x aangemaakt worden), ‘while’ is het leven vd socket. Eens socket er is => stream genereren  
outputstreamwriter: zet bytes om in karakters om te schrijven  
3) poortnummer gedeclareerd door server, dus client moet dit kennen (vraag is, hoe kent die dat?)  
als server draait op zelfde machine => localhost; anders => zie volgende lijn in code  
=> client moet weten waar server zit; eigen poortnummer wordt niet gedefinieerd – typisch java, dynamisch bepaald. Inputstreamreader: zet bytestroom om in leesbare stukken  
client moet weten waar server zit => niet transparant = probleem

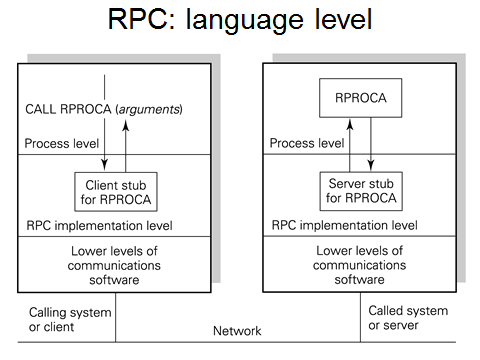
16.7 Remote Procedure Call (RPC)



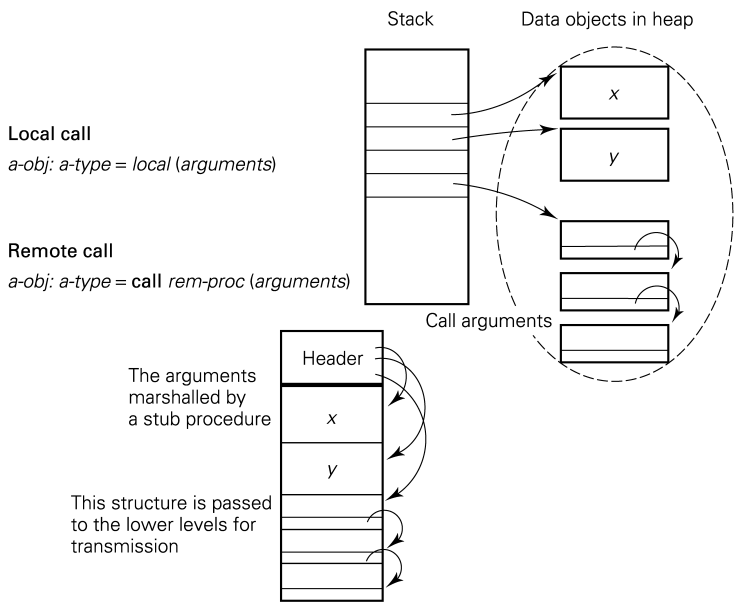
acknowledge niet getoond op figuur (alleen request & reply)  
probleem: procedure die ik wil aanroepen zit op andere machine (server) => call moet in blokje A opgevangen worden  
lower-level-protocols: typisch communicatieprotocols in kernel (bv TCP/IP – maar typisch UDP/IP)  
eigenlijke call gebeurt bij C  
E geeft dit door aan eigenlijke caller  
ideaal: caller merkt niet dat procedure ergens anders zit => distribution transparency -> eiglk moet van E naar D ook pijl teruggaan om te bevestigen dat antwoord is toegekomen. Blokjes (A, B,…) zijn proxy’s: ingebakken in RPC, worden gegenereerd, moet je niet zelf nog programmeren.  
typisch het werk dat door de proxy’s gedaan wordt, er wordt met een timer gewerkt: niet eeuwig wachten op een antwoord  
2 proxy’s: client-side & server-side  
figuur ook onvolledig in die zin dat proxy mechanisme heeft om procedures te lokaliseren

congestion: niets gecrasht, maar client weet dit niet => client stelt vraag, maar er komt maar geen antwoord  
congestion => ‘te veel traffic op netwerk’ => client kan niet zien waaraan het ligt dat er geen antwoord komt  
kan ook zijn dat server antwoord terug stuurt, maar geen bevestiging krijgt – heeft client antwoord gekregen?? Server weet dit niet…  
(timers at A or D expire)  
EXACTLY ONCE RPC: in absence of failures, a request is carries out once and only once. It is the nature of distributed systems that failures can occur and sometimes it is impossible to make remote calls. In this case an exception return is made  
AT MOST ONCE: user is given a choice of RPC semantics; it means that as soon as the timeout at A expires, control is returned to application level. The protocol does not retry, although the app level is likely to do so. If the app repeats the call, several copies of the same message may arrive at the called system, but with different identifiers and therefore not detectable by the RPC system as the same. The app has caused this situation and must deal with any repeated replies and any effects at the server.

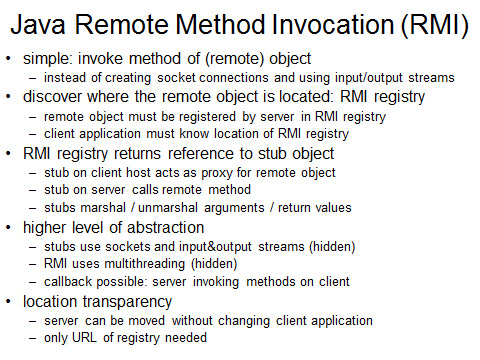
  
client crasht dus kan antwoord dat gestuurd wordt niet opvangen => server heeft werk voor niets gedaan – extern zichtbare effecten  
2 mogelijkheden: rollback via log OF operatie = idempotent, dan kan het geen kwaad  
  
  
  
  
  
  
crash bij B: geen probleem, nog niets gebeurd  
crash bij C: rollback of idempotent  
crash bij D: crash voordat antwoord kan verstuurd worden => voor client zelfde als congestion (client kan verschil niet zien), hier ook: rollback, tenzij idempotent

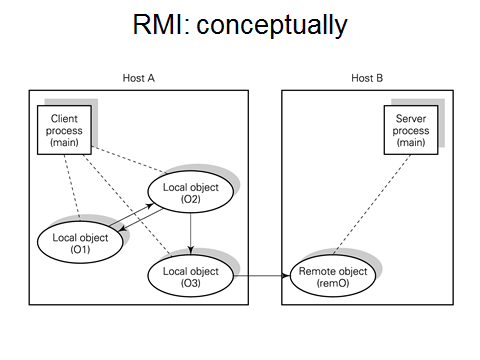
16.8 RPC-language integration  
16.8.1 Distribution transparency

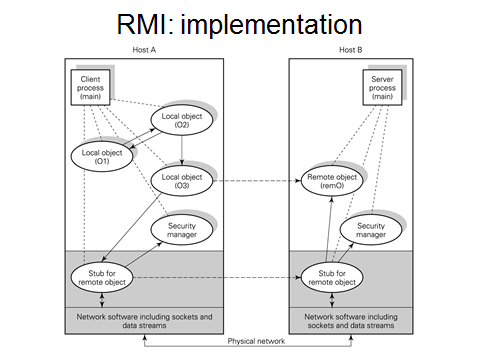
(in vorige figuren stonden lagen naast elkaar, nu onder elkaar)   
als ik een call doe, krijg ik een nieuw stack frame  
transparent approach: compiler must detect any call to a non-local procedure. For each remote procedure, a **stub** (sometimes called **proxy**) is generated so that a local call can be made to it, as shown in the figure. Thus the RPC support level is called transparently at runtime. The functions carried out by the stub are required by both transparent and non-transparent RPC systems. The only difference is that when transparency is needed, the stub is called as a local procedure with the same name as the remote procedure. The stub controls the assembling of arguments and invokes the RPC protocol.

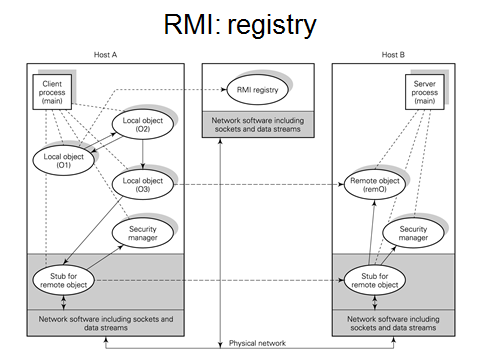
16.8.2 Argument marshaling  
als ik grote objecten wil doorsturen => zitten in heap; in stack zit verwijzing naar object in heap. Als procedure op zelfde machine zit: geen probleem, want heeft toegang tot zelfde adresruimte, maar indien op een andere machine: ‘platstrijken’ om door te geven naar andere kant van TCP/IP => taak van stub – stub aan andere kant doet dan omgekeerde: marshalling en unmarshalling  
(aantal objecten in Java niet serialiseerbaar en kunnen dus niet doogestuurd worden naar remote procedure)  
  
general approach to packing, flattening or **marshalling** arguments into a form suitable for transmission via a network buffer. Objects in heap must be copied into a data structure that the transmission system handles as a flat byte sequence. => call by copy semantics. Pointers (main memory addresses) within the objects must be translated into pointers that are meaningful within the buffer. Any data object that is pointed to more than once need only be copied once into the buffer.

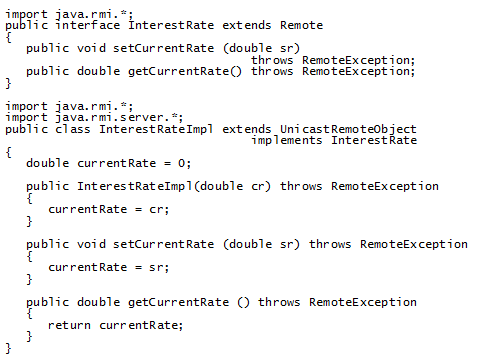
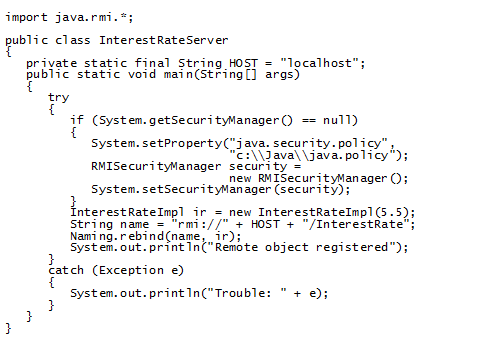
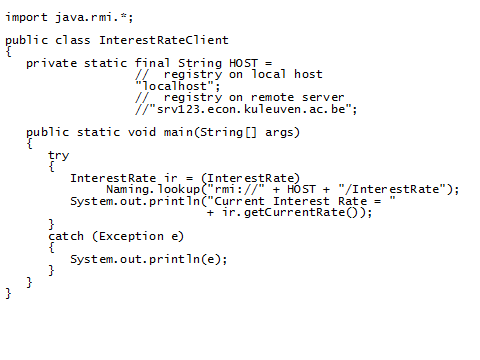
16.9 Java’s RMI: RPC in the general object model

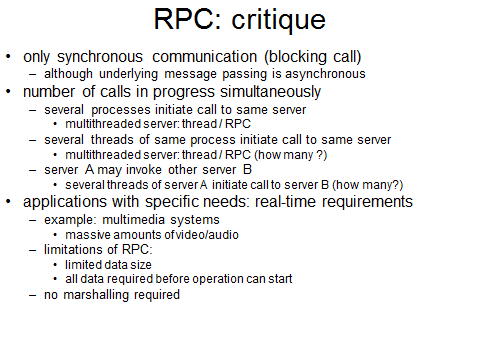
  
server is multithreaded, ook al heb ik dat niet zo gemaakt => RMI doet dit; gevaarlijk: moet gezorgd worden voor synchronisatie of wederzijdse uitsluiting (monitor of semaforen nodig)

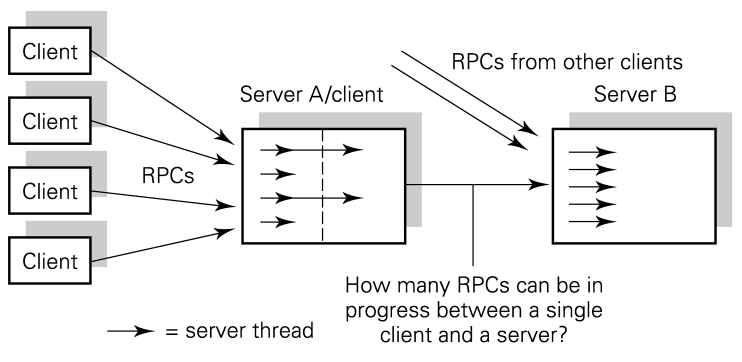
conceptually, what we want to achieve through RMI is represented in the figure. This shows a client process made up of many communicating local objects. The local object O3 is shown invoking the remote object remO. O1, O2 and O3 have been created as part of the client process running on Host A whereas remO has been created as part of the server process running on the remote host, B. the arrows between objects indicate the direction in which one or more method calls are begin made. Eg: O2 is shown as calling a method/methods of O3. Note that, although the method call is shown as flowing in one direction, the data flow between the objects is likely to be bidirectional, with input data flowing from O2 to O3, and return data resulting from the call flowing from O3 to O2; as is always the case in (sync) method invocation.

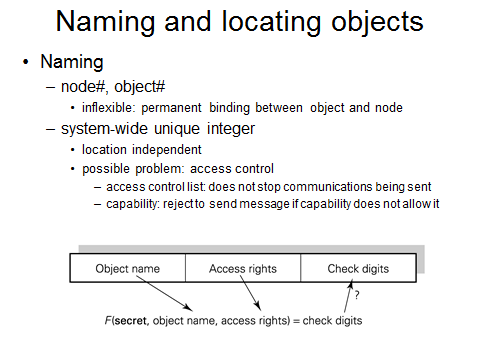
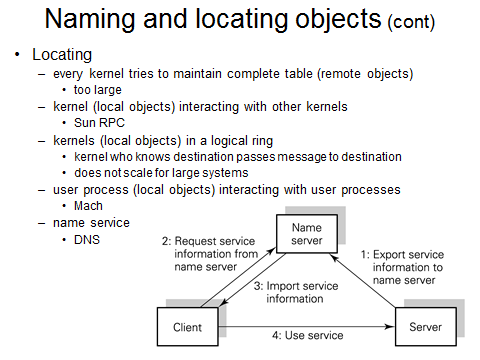
16.9.2 RMI mechanisms  
je wil call van O3 naar remO: gaat niet rechtstreeks => stub nodig  
(grijze zone onzichtbaar)  
stub voor remote object doet dan eigenlijke method call  
all the new objects needed for implementing the previous figure are shown unshaded, and the mechanisms not visible at the app level are shaded. Virtual communication between components are dashed; they indicate that, as far as the app is concerned, invocations are made directly on the objects at the other end of the link. However, the implementation of these virtual communication links by means of real communication links will be via layers that are not visible to the app.

alle objecten die van op afstand opgeroepen kunnen worden, moeten geregistreerd worden in registry  
bij oproep van remote method: registry contacteren => die zorgt voor stub => doet oproep  
sockets gebruiken eiglk sockets om te communiceren (byte stream, asynchroon ⬄ procedyre call = synchroon)  
mechanism by which a client can discover where a remote object is located: via RMI registry  
the registry is a java program that keeps a record of where remote objects are kept so that clients can access this information. One of the app objects, O1, of the client process is shown as calling the RMI registry in order to obtain a reference to the remote object remO.

   
  
  
location transparency: client moet niet weten waar object zit, maar wel waar registry zit

16.10 Critique of synchronous invocation  
  


Wat als een server nu oproep doet naar een andere server?  
4 calls die van 1 server komen, worden op andere server via 1 thread afgehandeld (sequentieel) => probleem…  
inkomende calls van andere servers nog krijgen wel weer meerdere threads  
marshalling is irrelevant voor multi-media toepassingen   
An implementation issue is the number of calls (RPCs or RMIs) that can be in progress at any time from different threads of a given process. It is important that a number of processes on a machine should be able to initiate calls to the same destination.  
Server A is employing several threads to service requests from different clients. Server A may itself need to invoke the service of another server, B. it must be possible for one thread on A to initiate a call to B and, while that call is in progress, another thread on A should be able to initiate another call to B.

16.11 Naming, location and binding  
an alternative to using access control lists stored with objects is to use a special kind of name that includes access rights, called a **capability** or protected name. figure gives an example of a capability and checking procedure. Possession of a capability is taken to be proof that the possessor has the rights indicated in the capability to the object named in it. When an object is created, a secret (random nimber) is generated and stored with the object. An encryption function is available to the object storage service. When a capability is issued, the object name, rights and the secret are put through the encryption function and the resulting number is stored in the capability as check digits. When the capability is presented, with a request to use the object, the object name and rights from the capability and the stored secret are put through the encryption function. The resulting number is checked against that in the capability. If the capability has been changed in any way, the check fails.  
  
fig: a **name service** is designed for the system. When a service is loaded at a given node, it sends it name, location and any other information necessary for using it to the name service. When a client wishes to invoke the service, the information necessary for communication with the service may be looked up in the name service. The location of the name service is assumed to be well known. It can be made available to systems on initial load.