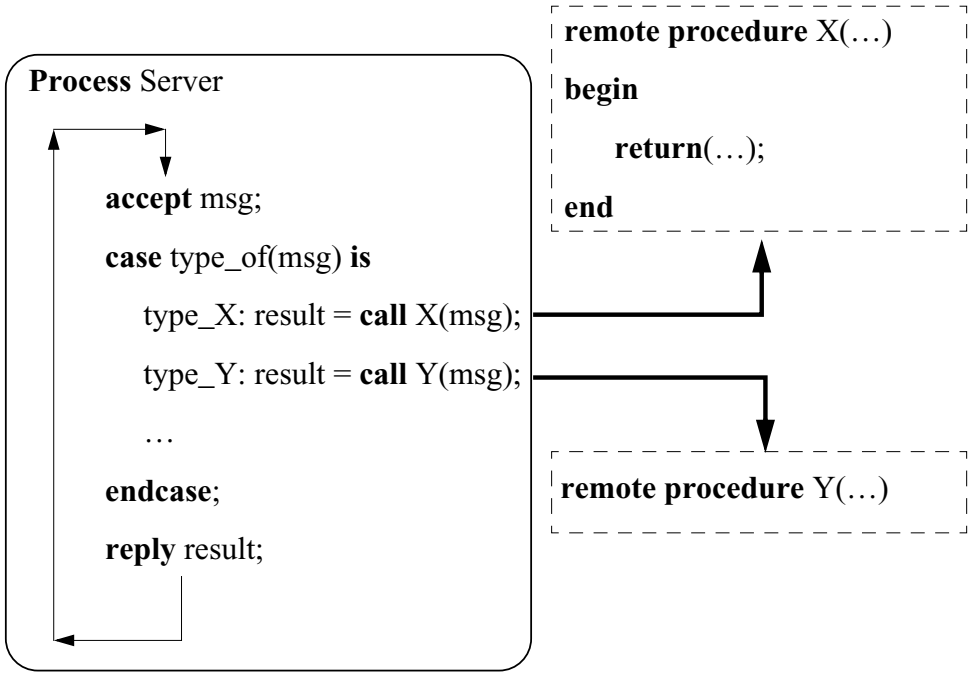
RPC: Server-Kontrollzyklus

- Warten auf Request, Verzweigen zur richtigen Prozedur:



"Dispatcher"

RPC-Stubs

- *Stub* = Stummel, Stumpf

Ersetzt durch ein längeres Programmstück (*Client-Stub*), welches u.a.

Client:

```
xxx ;
call S.X(out: a ; in: b)
xxx ;
```

- Parameter in eine Nachricht packt
- Nachricht an Server S versendet
- Timeout für die Antwort setzt
- Antwort entgegennimmt (oder ggf. exception bei timeout auslöst)
- Ergebnisparameter mit den Werten der Antwortnachricht setzt

- *Lokale Stellvertreter* ("proxy") des entfernten Gegenübers
  - Client-Stub / Server-Stub
  - simulieren einen lokalen Aufruf
  - sorgen für Packen und Entpacken von Nachrichten
  - konvertieren Datenrepräsentationen bei heterogenen Umgebungen
  - steuern das Übertragungsprotokoll (z.B. zur fehlerfreien Übertragung)
  - bestimmen ggf. Zuordnung zwischen Client und Server („Binding")

- Können oft weitgehend *automatisch generiert* werden
  - z.B. mit einem "RPC-Compiler" aus dem Client- oder Server-Code und ggf. einer "sprachneutralen" Schnittstellenbeschreibung (z.B. IDL)
  - Compiler kennt (bei streng getypten Sprachen) Datenformate
  - Schnittstelle zum verfügbaren Transportsystem sind auch bekannt
  - Nutzung fertiger Bibliotheken für Formatkonversion, Multithreading usw.
  - Nutzung von Routinen der *RPC-Laufzeitumgebung* (z.B. zur Kommunikationssteuerung, Fehlerbehandlung etc.)

wird nicht generiert, sondern dazugebunden

- Stubs sorgen also für *Transparenz*

RPC: Marshalling

- Zusammenstellen der Nachricht aus den aktuellen Prozedurparametern
  - ggf. dabei geeignete Codierung (komplexer) Datenstrukturen
  - Glätten ("flattening") komplexer (ggf. verzeigerter) Datenstrukturen zu einer Sequenz von Basistypen (mit Strukturinformation)
  - umgekehrte Tranformation oft als "unmarshalling" bezeichnet

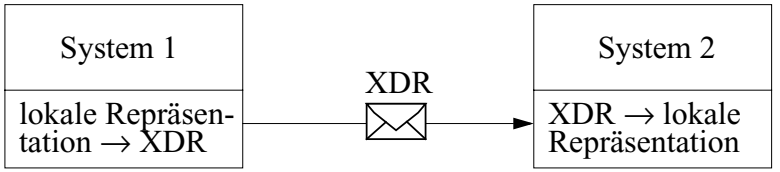
- Problem: RPCs werden oft in *heterogenen* Umgebungen eingesetzt mit unterschiedlicher Repräsentation z.B. von
  - Strings (Längenfeld ↔ '\0')
  - Character (ASCII ↔ Unicode)
  - Arrays (zeilen- ↔ spaltenweise)
  - niedrigstes Bit einer Zahl vorne oder hinten

- Allerdings vorteilhaft: Client und Server kennen Typ der Parameter, falls das Programm in Quellform vorliegt oder vom Compiler generierte Typtabellen existieren
  (Problematisch ggf. bei un- / schwach-getypten Sprachen!)

RPC: Marshalling

1) Umwandlung in eine gemeinsame Standardrepräsentation
   - z.B. "XDR" (eXternal Data Representation)

- Prinzip der Datenkonversion:



- Beachte: Jeweils zwei Konvertierungen erforderlich; für jeden Datentyp jeweils Kodierungs- und Dekodierungsroutinen vorsehen

2) Oder lokale Datenrepräsentation verwenden und dies in der Nachricht vermerken
   - "receiver makes it right"
   - Vorteil: bei gleichen Systemumgebungen / Maschinentypen ist keine (doppelte) Umwandlung nötig
   - Empfänger muss aber mit der Senderrepräsentation umgehen können

RPC: Transparenzproblematik

- RPCs sollten so weit wie möglich lokalen Prozeduraufrufen gleichen, es gibt aber einige subtile Unterschiede

  bekanntes Programmier-paradigma!

  - Client- / Serverprozesse haben ggf. unterschiedliche Lebenszyklen: Server könnte noch nicht oder nicht mehr oder in einer "falschen" Version existieren

- Leistungstransparenz
  - RPC i.a. wesentlich langsamer
  - Bandbreite bei umfangreichen Parametern beachten
  - ungewisse, variable Verzögerungen

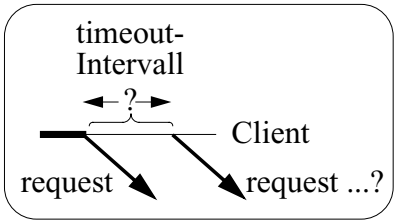
- Ortstransparenz
  - evtl. muss Server ("Zielort") bei Adressierung explizit genannt werden
  - erkennbare Trennung der Adressräume von Client und Server
  - i.a. keine Pointer/Referenzparameter als Parameter möglich
  - keine Kommunikation über globale Variablen möglich

- Fehlertransparenz
  - es gibt mehr Fehlerfälle (beim klassischen Prozeduraufruf gilt: Client = Server → "fail-stop"-Verhalten, also alles oder nix)
  - partielle ("einseitige") Systemausfälle: Server-Absturz, Client-Absturz
  - Nachrichtenverlust (ununterscheidbar von zu langsamer Nachricht!)
  - Anomalien durch Nachrichtenverdopplung (z.B. nach Timeout)
  - Crash kann zu "ungünstigen Momenten" erfolgen (kurz vor / nach Senden / Empfangen einer Nachricht etc.)
  - Client / Server haben zumindest zwischenzeitlich eine unterschiedliche Sicht des Zustandes einer "RPC-Transaktion"

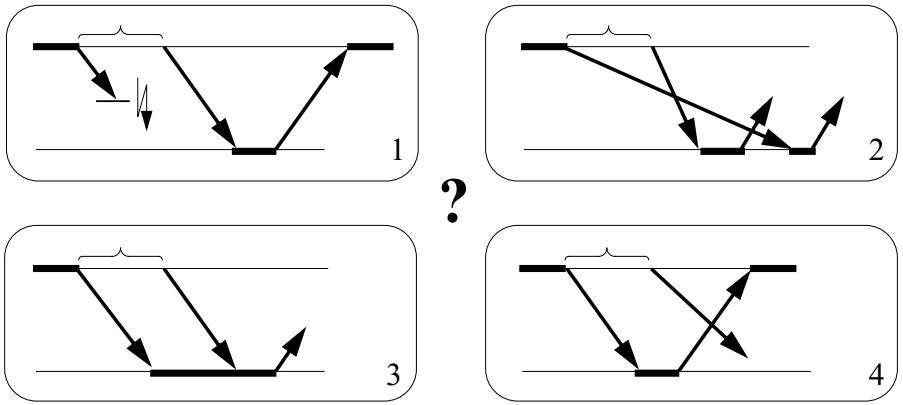
==> Fehlerproblematik ist also "kompliziert"!

Typische Fehlerursachen bei RPC: I. Verlorene Request-Nachricht

- *Gegenmassnahme*:
  - Nach Ablauf eines Timers ohne Reply die Request-Nachricht erneut senden

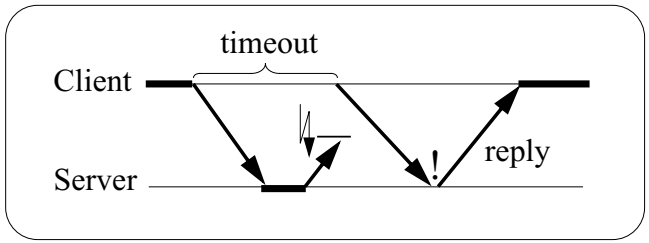


- *Probleme*:
  - Wieviele Wiederholungsversuche maximal?
  - Wie gross soll der Timeout sein?
  - Falls die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam:
    - Doppelte Request-Nachricht! (Gefährlich bei nicht-idempotenten Operationen!)
    - Server sollte solche Duplikate erkennen. (Wie? Benötigt er dafür einen Zustand? Genügt es, wenn der Client Duplikate als solche kennzeichnet? Genügen Sequenznummern? Zeitmarken?)
    - Würde das Quittieren der Request-Nachricht etwas bringen?



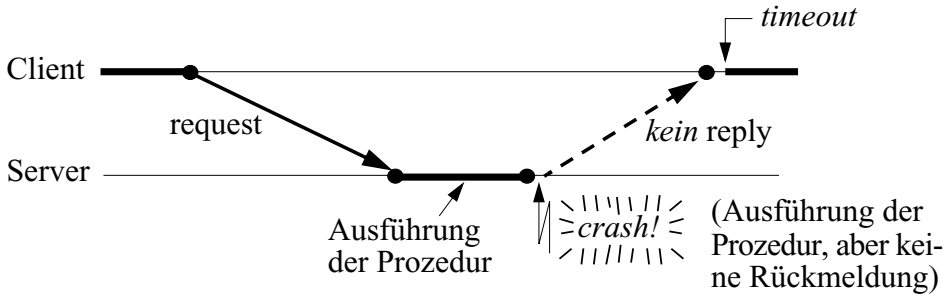
II. Verlorene Reply-Nachricht

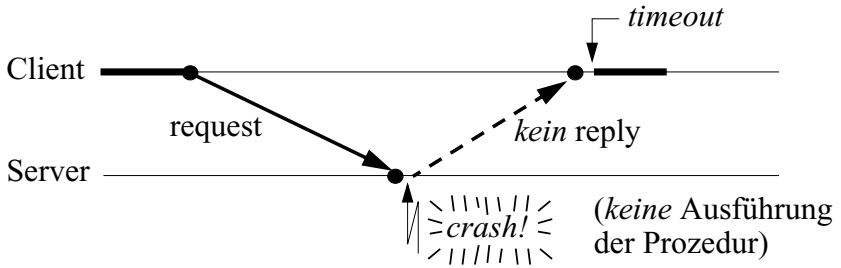
- *Gegenmassnahme 1:* analog zu verlorener Request-Nachricht
  - Also: Anfrage nach Ablauf des Timeouts wiederholen
- *Probleme:*
  - Vielleicht ging aber tatsächlich der Request verloren?
  - Oder der Server war nur langsam und arbeitet noch?
  - Ist aus Sicht des Clients nicht unterscheidbar!



- *Gegenmassnahme 2:*
  - Server könnte eine "Historie" der versendeten Replies halten
    - Falls Server Request-Duplikate erkennt und den Auftrag bereits ausgeführt hat: letztes Reply erneut senden, ohne die Prozedur erneut auszuführen!
    - Pro Client muss nur das neueste Reply gespeichert werden.
    - Bei vielen Clients u.U. dennoch Speicherprobleme:
      → Historie nach "einiger" Zeit löschen.
      (Ist in diesem Zusammenhang ein ack eines Reply sinnvoll?)
      Und wenn man ein gelöschtes Reply später dennoch braucht?

III. Server-Crash

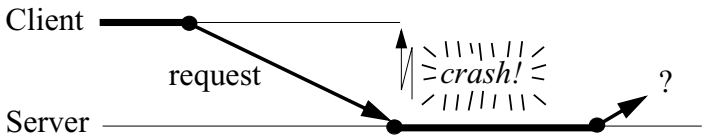




Probleme:

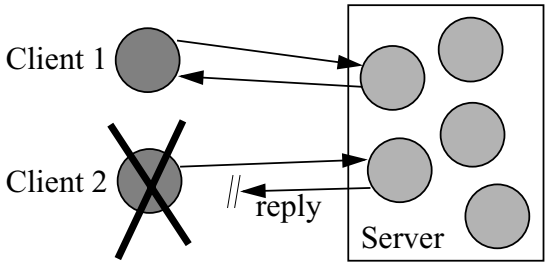
- Wie soll der Client dies unterscheiden?
  - ebenso: Unterschied zu verlorenem request bzw. reply?
  - Sinn und Erfolg konkreter Gegenmassnahmen hängt ggf. davon ab
  - Client *meint* u.U. zu Unrecht, dass ein Auftrag nicht ausgeführt wurde (→ falsche Sicht des Zustandes!)
- Evtl. Probleme nach einem Server-Restart
  - z.B. "Locks", die noch bestehen (Gegenmassnahmen?) bzw. allgemein: "verschmutzer" Zustand durch frühere Inkarnation
  - typischerweise ungenügend Information ("Server Amnesie"), um in alte Kommunikationszustände problemlos wieder einzusteigen

IV. Client-Crash



- Reply des Servers wird nicht abgenommen
  - Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
  - blockiert i.a. Ressourcen beim Server!

- "Orphans" (Waisenkinder) beim Server
  - Prozesse, deren Auftraggeber nicht mehr existiert



Beispiel: Pro RPC-Auftrag eines Client gründet der Server einen eigenen Prozess, der erst terminiert, wenn reply abgenommen wurde

- Nach Neustart des Client dürfen alte Replies nicht stören
  - "Antworten aus dem Nichts" (Gegenmassnahme: Epochen-Zähler)
- Nach Restart könnte ein Client versuchen, Orphans zu terminieren (z.B. durch Benachrichtigung der Server)
  - dadurch bleiben aber u.U. locks etc. bestehen
  - Orphans könnten bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben...
- Pessimistischer Ansatz: Server fragt bei laufenden Aufträgen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)

RPC-Fehlersemantik

Operationale Sichtweise:

- Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Requests oder Replies sowie auf wiederholte Requests reagiert?
- Und wie auf gecrashte Server / Clients?

---

1) Maybe-Semantik:

- Keine Wiederholung von Requests
- *Einfach und effizient*
- Keinerlei Erfolgsgarantien → oft nicht anwendbar
  Mögliche Anwendungsklasse: Auskunftsdienste (noch einmal probieren, wenn keine Antwort kommt)

wird etwas euphemistisch oft als "best effort" bezeichnet

2) At-least-once-Semantik:

- Hartnäckige Wiederholung von Requests
- Keine Duplikatserkennung (*zustandsloses Protokoll* auf Serverseite)
- Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

RPC-Fehlersemantik (2)

3) At-most-once-Semantik:

- Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
- Keine wiederholte Ausführung der Prozedur, sondern ggf. erneutes Senden des Reply
- Geeignet auch für *nicht-idempotente* Operationen

4) Exactly-once-Semantik:

- Wunschtraum?
- Oder geht es zumindest unter der *Voraussetzung*, dass der Server nicht crasht und ein reply letztlich auch durchkommt? (Z.B. durch hartnäckige Wiederholung von Requests?)
- Was ist mit verteilten Transaktionen? (→ Datenbanken! Stichworte: Checkpoint; persistente Datenspeicherung; Recovery...)

- Nochmals: Fehlertransparenz bei RPC?
  - Problem: Client / Server haben u.U. (temporär?) eine inkonsistente Sicht
  - Einige Fehler sind bei gewöhnlichen Prozeduraufrufen nicht möglich
  - Timeout beim Client kann *verschiedene* Ursachen haben (verlorener Request, verlorenes Reply, langsamer Request bzw. Reply, langsamer Server, abgestürzter Server...) → Fehlermaskierung schwierig
  - Vollständige Transparenz ist kaum erreichbar
  - Hohe Fehlertransparenz = hoher Aufwand

Wirkung der RPC-Fehlersemantik

	Fehlerfreier Ablauf	**Nachrichten-verluste**	**Ausfall des Servers**
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0
At-least-once	Ausführung: 1 Ergebnis: 1	Ausführung: ≥1 Ergebnis: ≥1	Ausführung: ≥0 Ergebnis: ≥0 (*)
At-most-once	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0/1 (+)	Ausführung: 0/1 Ergebnis: 0
Exactly-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1

(*) Nach einem evtl. Server-Restart können Retries Ergebnisse verursachen

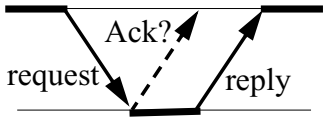
(+) Bei Verlust der *Antwort*nachricht wird diese erneut gesendet

May-be → At-least-once → At-most-once → ...
ist zunehmend aufwendiger zu realisieren!

- man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt“ wie nötig
  (Aber dieses Motto gilt natürlich nicht in allen sonstigen Lebenssituationen! Ein Sicherheitsabstand durch “besser als notwendig” ist oft angebracht!)

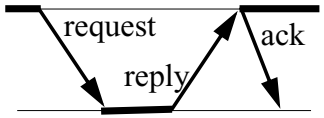
RPC-Protokolle

- *RR-Protokoll* (“Request-Reply”):



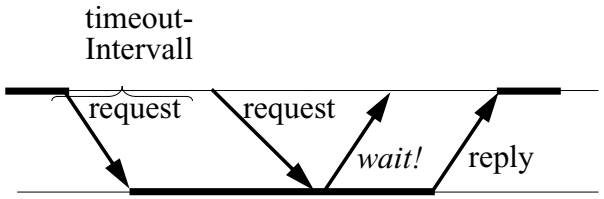
- Reply ist implizite Quittung für Request
- lohnt sich ggf. eine unmittelbare Bestätigung des Request?

- *RRA-Protokoll* (“Request-Reply-Acknowledge”):



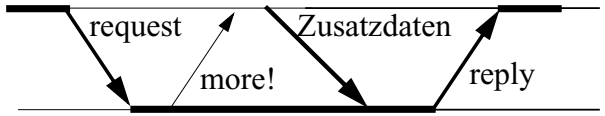
- “pessimistischer” als das RR-Protokoll
- Vorteil: Server kann evtl. gespeicherte Replies frühzeitig löschen (und natürlich Replies bei Ausbleiben des ack wiederholen)

- *Sinnvoll bei langen Aktionen / überlasteten Servern:*



“wait” = Bestätigung eines erkannten Duplikats

- *Parameter-Übertragung „on demand“*



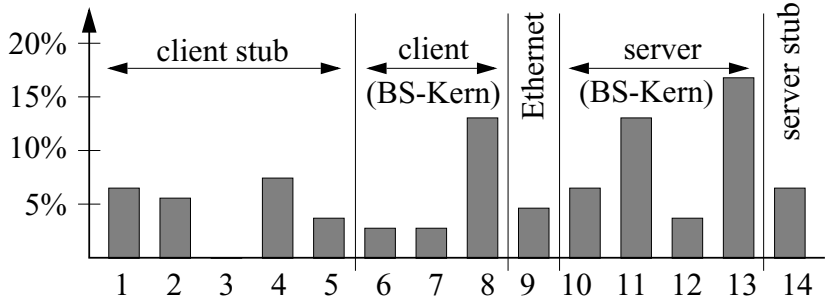
- spart Pufferkapazität
- bessere Flusssteuerung
- Zusatzdaten abhängig vom konkreten Ablauf

- *Weitere RPC-Protokollaspekte:*
  - effiziente Implementierung einer geeigneten (=?) Fehlersemantik
  - geeignete Nutzung des zugrundeliegenden Protokolls (ggf. aus Effizienzgründen eigene Paketisierung der Daten, Flusssteuerung, selektive Wiederholung einzelner Nachrichtenpakete bei Fehlern, eigene Fehlererkennung / Prüfsummen, kryptogr. Verschlüsselung...)

RPC: Effizienz

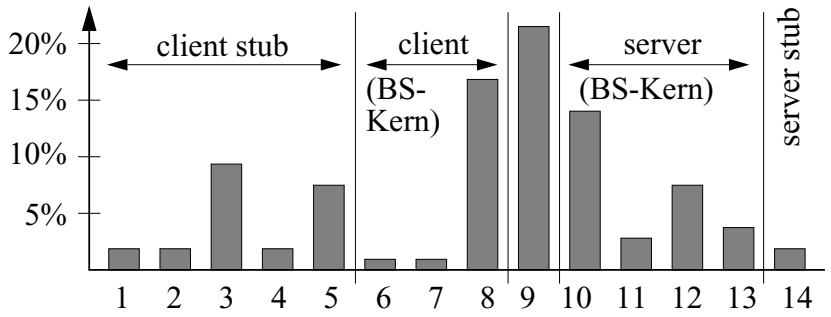
Analyse eines RPC-Protokolls durch Schroeder
(zitiert nach A. Tanenbaum)

a) Null-RPC (Nutznachricht der Länge 0, kein Auftragsbearbeitung):



1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to kernel
7. Queue packet for transmission
8. Move packet to controller over the bus
9. Ethernet transmission time
10. Get packet from controller
11. Interrupt service routine
12. Compute UDP checksum
13. Context switch to user space
14. Server stub code

b) 1440 Byte Nutznachricht (ebenfalls kein Auftragsbearbeitung):

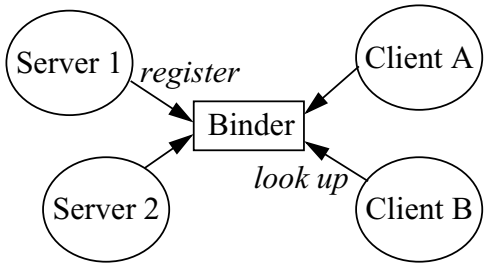


- Eigentliche Übertragung kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) keineswegs vernachlässigbar
- Bei kurzen Nachrichten: Kontextwechsel zw. Anwendung und BS wichtig
- Mehrfaches Kopieren kostet viel

RPC: Binding

- Problem: Wie werden Client und Server “gematcht”?
  - Verschiedene Rechner und i.a. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gem. Umgebung)

Dynamisches Binden mittels eines “Binders”



- Server (-stub) gibt den Namen etc. seines Services (RPC-Routine) dem Binder bekannt
  - “*register*”; “exportieren” der RPC-Schnittstelle (Typen der Parameter...)
  - ggf. auch wieder abmelden
- Client erfragt beim Binder die Adresse eines geeigneten Servers
  - “*look up*”; “importieren” der RPC-Schnittstelle

oft auch “registry” oder “look-up service” genannt

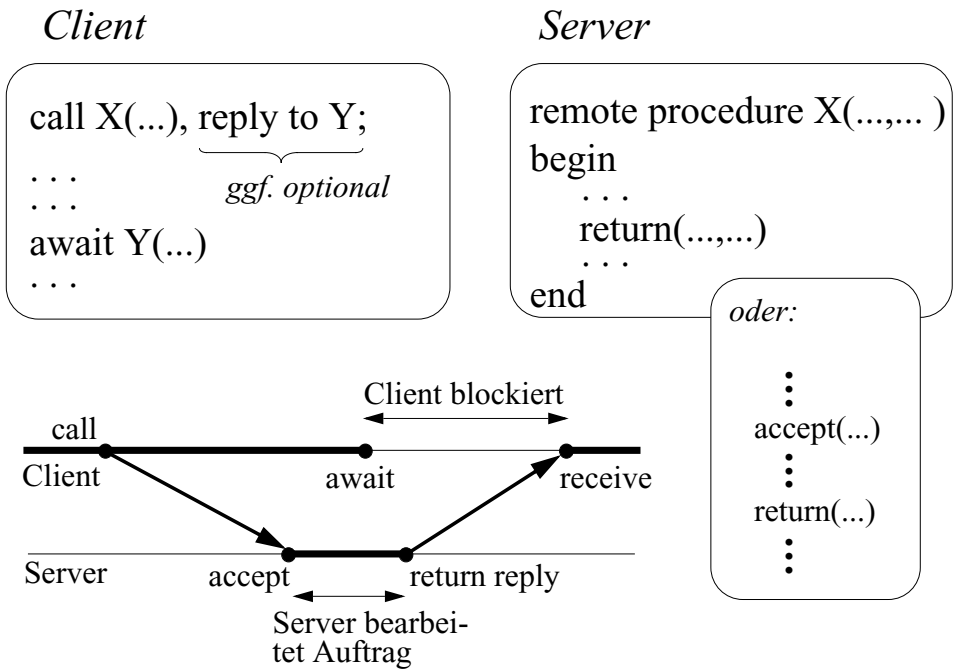
- *Vorteile*: im Prinzip kann Binder
  - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Existenz eines Services testen
  - verschiedene Versionen eines Dienstes verwalten

dann eher “Trader” oder “Broker”

- *Probleme*:
  - zentraler Binder ist ein potentieller Engpass (Binding-Service geeignet verteilen? Konsistenz!)
  - dynamisches Binden kostet Ausführungszeit

Asynchroner RPC

- andere Bezeichnung: “Remote Service Invocation”
- auftragsorientiert → Antwortverpflichtung



- Parallelverarbeitung von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Future-Variablen

- Zuordnung Auftrag / Ergebnisempfang bei der asynchron-auftragsorientierten Kommunikation?
  - unterschiedliche Ausprägung auf Sprachebene möglich
  - "await" könnte z.B. einen bei "call" zurückgelieferten "handle" als Parameter erhalten (also z.B. Y = call X(...); ... await (Y);)
  - ggf. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt-oder Exception-Routine)

- Spracheinbettung evtl. auch durch "Future-Variablen"
  - Future-Variable = handle, der wie ein Funktionsergebnis in Ausdrücke eingesetzt werden kann
  - Auswertung der Future-Variable erst, wenn unbedingt nötig
  - Blockade nur dann, falls Inhalt bei Auswertung noch nicht feststeht
  - Beispiel:

```
FUTURE future: integer;
some_value: integer;
…
future = call(…);
…
some_value = 4711;
print(some_value + future);
```


Die Socket-Programmierschnittstelle

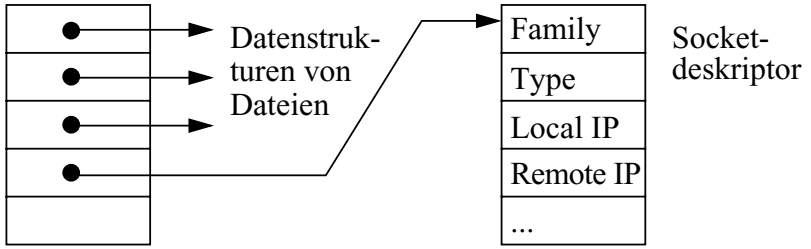
- Zu TCP (bzw. UDP) gibt es keine festgelegten "APIs"
- Bei UNIX sind dafür "sockets" als Zugangspunkte zum Transportsystem entstanden
  -diese definieren in einer Sprache (z.B. Java) dann eine Art "API"
- Semantik eines sockets: analog zu Datei-Ein/Ausgabe
  - ein socket kann aber auch mit mehreren Prozessen verbunden sein
- Programmiersprachliche Einbindung (C, Java etc.)
  - sockets werden wie Variablen behandelt (können Namen bekommen)
  - Beispiel in C (Erzeugen eines sockets):

```
int s;
s = socket(int PF_INET, int SOCK_STREAM, 0);
```

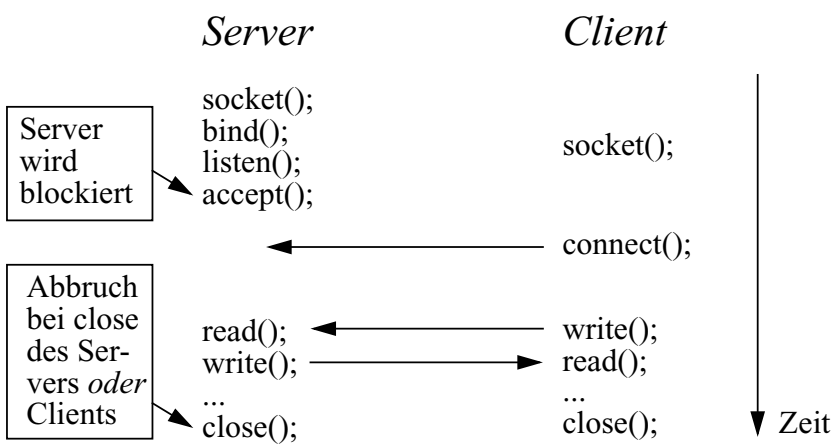
"Family": Internet oder nur lokale Domäne

"Type": Angabe, ob TCP verwendet ("stream"); oder UDP ("datagram")

- Bibliotheksfunktion "socket" erzeugt einen Deskriptor
  - wird innerhalb der Filedeskriptortabelle des Prozesses angelegt
  - Datenstruktur wird allerdings erst mit einem nachfolgenden "bind"-Aufruf mit Werten gefüllt (binden der Adressinformation aus Host-Adresse und einer "bekannten" lokalen Portnummer an den socket)



Client-Server mit Sockets (Prinzip)



- Voraussetzung: Client kennt die IP-Adresse des Servers sowie die Portnummer (des Dienstes)
  - muss beim connect angegeben werden
- Mit “listen” richtet der Server eine Warteschlange für Client-connect-Anforderungen ein
  - Auszug aus der Beschreibung: *“If a connection request arrives with the queue full, tcp will retry the connection. If the backlog is not cleared by the time the tcp times out, the connect will fail”*
- Accept / connect implementieren ein “Rendezvous”
  - mittels des 3-fach-Handshake von TCP
  - bei “connect” muss der Server bereits listen / accept ausgeführt haben
- Rückgabewerte von write bzw. read: Anzahl der tatsächlich gesendeten / empfangenen Bytes
- Varianten: Es gibt ein select, ein nicht-blockierendes accept etc., vgl. dazu die (Online-)Literatur

Ein Socket-Beispiel in C

- Verwendung von sockets in C erfordert u.a.
  - Header-Dateien mit C-Daten-Strukturen, Konstanten etc.
  - Programmcode zum Anlegen, Füllen etc. von Strukturen
  - Fehlerabfrage und Behandlung
- Socket-Programmierung ist ziemlich “low level”
  - etwas umständlich, fehleranfällig bei der Programmierung
  - aber “dicht am Netz” und dadurch ggf. manchmal von Vorteil
- Zunächst der Quellcode für den Client:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711
#define BUF_SIZE 1024

main(argc,argv)
int   argc;
char *argv[];
{
    int                 sock, run;
    char                buf[BUF_SIZE];
    struct sockaddr_in  server;
    struct hostent     *hp;
    if(argc != 2)
    {
        fprintf(stderr,”usage: client
                       <hostname>\n”);
        exit(2);
    }
```

Socket-Beispiel: Client

```
    /* create socket */
    sock = socket(AF_INET,SOCK_STREAM,0);
    if(sock < 0)
    {
        perror(“open stream socket”);
        exit(1);
    }
    server.sin_family = AF_INET;
    /* get internet address of host specified by command line */
    hp = gethostbyname(argv[1]);
    if(hp == NULL)
    {
       fprintf(stderr,”%s unknown host.\n”,argv[1]);
       exit(2);
    }
    /* copies the internet address to server address */
    bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
    /* set port */
    server.sin_port = PORT;
    /* open connection */
    if(connect(sock,&server,sizeof(struct sockaddr_in)) < 0)
    {
       perror(“connecting stream socket”);
       exit(1);
    }
    /* read input from stdin */
    while(run=read(0,buf,BUF_SIZE))
    {
       if(run<0)
       {
          perror(“error reading from stdin”);
          exit(1);
       }
       /* write buffer to stream socket */
       if(write(sock,buf,run) < 0)
       {
          perror(“writing on stream socket”);
          exit(1);
       }
    }
    close(sock);
}
```


Socket-Beispiel: Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711                    /* random port number */
#define MAX_QUEUE 1
#define BUF_SIZE 1024

main()
{
    int sock_1,sock_2;     /* file descriptors for sockets */
    int rec_value, length;
    char buf[BUF_SIZE];
    struct sockaddr_in server;

    /* create stream socket in internet domain*/
    sock_1 = socket(AF_INET,SOCK_STREAM,0);
    if (sock_1 < 0)
    {
       perror(“open stream socket”);
       exit(1);
    }
    /* build address in internet domain */
    server.sin_family = AF_INET;
    /* everyone is allowed to connet to server */
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = PORT;
    /* bind socket */
    if(bind(sock_1,&server,sizeof(struct sockaddr_in)))
    {
       perror(“bind socket to server_addr”);
       exit(1);
    }
```

Socket-Beispiel: Server (2)

```
    listen(sock_1,MAX_QUEUE);
    /* start accepting connection */
    sock_2 = accept(sock_1,0,0);
    if(sock_2 < 0)
    {
       perror("accept");
       exit(1);
    }
    /* read from sock_2 */
    while (rec_value=read(sock_2,buf,BUF_SIZE))
    {
       if(rec_value<0)
       {
          perror("reading stream message");
          exit(1);
       }
       else
          write(1,buf,rec_value);
    }
    printf("Ending connection.\n");
    close(sock_1); close(sock_2);
}
```

- Sinnvolle praktische Übungen (evtl. auch in Java):

  1) Beispiel genau studieren; Semantik der socket-Operationen etc. nachlesen: Bücher oder Online-Dokumentation (z.B. von UNIX)

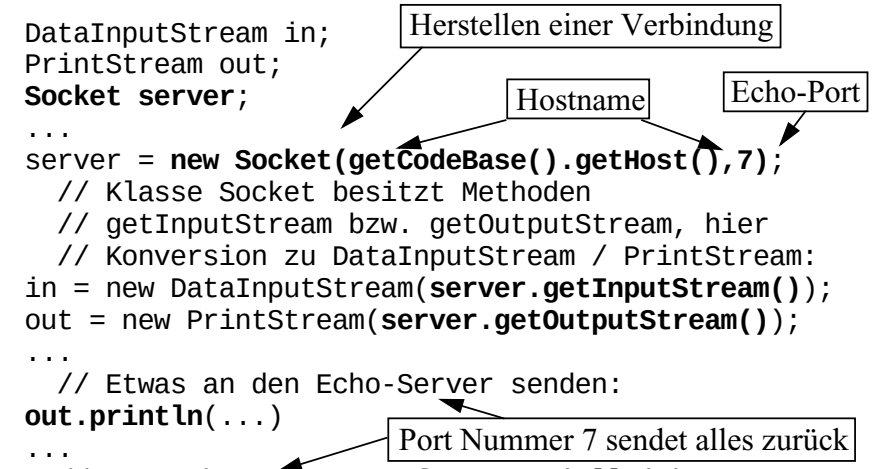
  2) Varianten und andere Beispiele implementieren, z.B.:
     - Server, der zwei Zahlen addiert und Ergebnis zurücksendet
     - Produzent / Konsument mit dazwischenliegendem Pufferprozess (unter Vermeidung von Blockaden bei vollem Puffer)
     - Server, der mehrere Clients gleichzeitig bedienen kann
     - Trader, der geeignete Clients und Server zusammenbringt
     - Messung des Durchsatzes im LAN; Nachrichtenlängen in mehreren Experimenten jeweils verdoppeln

Übungsbeispiel: Sockets unter Java

(auf den nächsten 5 Seiten)

- Auch unter Java lassen sich Sockets verwenden
  - sogar bequemer als unter C
  - Paket java.net.* enthält u.a. die Klasse "Socket"
  - Streamsockets (verbindungsorientiert) bzw. Datagrammsockets

- Beispiel:

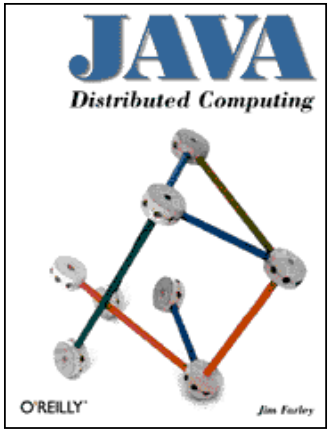



```
DataInputStream in;
PrintStream out;
Socket server;
...
server = new Socket(getCodeBase().getHost(),7);
  // Klasse Socket besitzt Methoden
  // getInputStream bzw. getOutputStream, hier
  // Konversion zu DataInputStream / PrintStream:
in = new DataInputStream(server.getInputStream());
out = new PrintStream(server.getOutputStream());
...
  // Etwas an den Echo-Server senden:
out.println(...)
...
  // Vom Echo-Server empfangen; vielleicht
  // am besten in einem anderen Thread:
String line;
while((line = in.readLine()) != null)
  // line ausgeben
...
server.close;
```

- Zusätzlich: Fehlerbedingungen mit Exceptions behandeln ("try"; "catch")
  - z.B. "UnknownHostException" beim Gründen eines Socket

Client-Server mit Sockets in Java

- Beispiel aus dem Buch
  Java Distributed Computing
  von Jim Farley (O'Reilly)



- Hier der Client:

```
import java.lang.*;
import java.net.*;
import java.io.*;

public class SimpleClient
{
  // Our socket connection to the server
  protected Socket serverConn;

    public SimpleClient(String host, int port)
      throws IllegalArgumentException {
    try {
      System.out.println("Trying to connect to "
      + host + " " + port);
      serverConn = new Socket(host, port);
    }
    catch (UnknownHostException e) {
      throw new IllegalArgumentException
        ("Bad host name given.");
    }
    catch (IOException e) {
      System.out.println("SimpleClient: " + e);
      System.exit(1);
    }

    System.out.println("Made server connection.");
  }
```

Konstruktor


```
public static void main(String argv[]) {
  if (argv.length < 2) {
    System.out.println ("Usage: java \
       SimpleClient <host> <port>");
    System.exit(1);
  }

  int port = 3000;
  String host = argv[0];
  try { port = Integer.parseInt(argv[1]); }
  catch (NumberFormatException e) {}

  SimpleClient client  =  new SimpleClient(host, port);
  client.sendCommands();
}

public void sendCommands() {
  try {
    DataOutputStream dout =
      new DataOutputStream(serverConn.getOutputStream());
    DataInputStream din =
      new DataInputStream(serverConn.getInputStream());

    // Send a GET command...
    dout.writeChars("GET goodies ");
    // ...and receive the results
    String result = din.readLine();
    System.out.println("Server says: \"" + result + "\"");
  }
  catch (IOException e) {
    System.out.println("Communication SimpleClient: " + e);
    System.exit(1);
  }
}

public synchronized void finalize() {
  System.out.println("Closing down SimpleClient...");
  try { serverConn.close(); }
  catch (IOException e) {
    System.out.println("Close SimpleClient: " + e);
    System.exit(1);
  }
}
```

Host- und Port-nummer von der Kommandozeile

Wird vom Garbage-Collector aufgerufen, wenn keine Referenzen auf den Client mehr existieren ('close' ggf. am Ende von 'sendCommands')

Der Server

```
import java.net.*;
import java.io.*;
import java.lang.*;

public class SimpleServer {
  protected int portNo = 3000;
  protected ServerSocket clientConnect;

  public SimpleServer(int port) throws
      IllegalArgumentException {
    if (port <= 0)
      throw new IllegalArgumentException(
       "Bad port number given to SimpleServer constructor.");

    // Try making a ServerSocket to the given port
    System.out.println("Connecting server socket to port");
    try { clientConnect = new ServerSocket(port); }
    catch (IOException e) {
     System.out.println("Failed to connect to port " + port);
      System.exit(1);
    }

    // Made the connection, so set the local port number
    this.portNo = port;
  }

  public static void main(String argv[]) {
    int port = 3000;
    if (argv.length > 0) {
      int tmp = port;
      try {
        tmp = Integer.parseInt(argv[0]);
      }
      catch (NumberFormatException e) {}
      port = tmp;
    }

    SimpleServer server = new SimpleServer(port);
    System.out.println("SimpleServer running on port " +
       port + "...");
    server.listen();
  }
```

Default-Port, an dem der Server auf eine Client-Verbindung wartet

Socket, der Verbindungs-wünsche entgegennimmt

Konstruktor

Portnummer von Kommandozeile

Aufruf der Methode "listen" (siehe unten)


```
public void listen() {
    try {
      System.out.println("Waiting for clients...");
      while (true) {
        Socket clientReq = clientConnect.accept();
        System.out.println("Got a client...");
        serviceClient(clientReq);
      }
    }
    catch (IOException e) {
      System.out.println("IO exception while listening.");
      System.exit(1);
    }
  }

  public void serviceClient(Socket clientConn) {
    SimpleCmdInputStream inStream = null;
    DataOutputStream outStream = null;
    try {
      inStream = new SimpleCmdInputStream
        (clientConn.getInputStream());
      outStream = new DataOutputStream
        (clientConn.getOutputStream());
    }
    catch (IOException e) {
      System.out.println("SimpleServer: I/O error.");
    }
    SimpleCmd cmd = null;
    System.out.println("Attempting to read commands...");
    while (cmd == null || !(cmd instanceOf DoneCmd)) {
      try { cmd = inStream.readCommand(); }
      catch (IOException e) {
        System.out.println("SimpleServer (read): " + e);
        System.exit(1);
      }

      if (cmd != null) {
        String result = cmd.Do();
        try { outStream.writeBytes(result); }
        catch (IOException e) {
          System.out.println("SimpleServer (write): " + e);
          System.exit(1);
        }
      }
    }
  }
}
```

Warten auf connect eines Client, dann Gründen eines Sockets

Von DataInput-Stream abge-leitete Klasse

Klasse SimpleCmd hier nicht gezeigt

Schleife zur Entgegennahme und Ausführung von Kommandos

finalize-Methode hier nicht gezeigt

Java als “Internet-Programmiersprache”

- Java hat eine Reihe von Konzepten, die die Realisierung verteilter Anwendungen erleichtern, z.B.:
  - Socket-Bibliothek zusammen mit Input- / Output-Streams
  - Remote Method Invocation (RMI): Entfernter Methodenaufruf mit Transport (und dabei Serialisierung) auch komplexer Objekte
  - CORBA-APIs
  - eingebautes Thread-Konzept
  - java.security-Paket
  - plattformunabhängiger Bytecode mit Klassenlader (Java-Klassen können über das Netz transportiert und geladen werden; Bsp.: Applets)

Im Vergleich zu RPC: Nicht notw. Master-Slave, sondern peer-to-peer

Damit z.B. Realisierung eines “Meta-Protokolls”: Über einen Socket vom Server eine Klasse laden (und Objekt-Instanz gründen), was dann (auf Client-Seite) ein spezifisches Protokoll realisiert. (Vgl. “mobiler Code”, Jini...)

- Das UDP-Protokoll kann mit “Datagram-Sockets” verwendet werden, z.B. so:

```
try {
  DatagramSocket s = new DatagramSocket();
  byte[] data = {'H','e','l','l','o'};
  InetAddress addr = InetAddress.getByName("my.host.com");
  DatagramPacket p = new DatagramPacket(data,
    data.length, addr, 5000);
  s.send(p);
}
catch (Exception e) {
  System.out.println("Exception using datagrams:");
  e.printStackTrace();
}
```

Port-Nummer

  - entsprechend zu “send” gibt es ein “receive”
  - InetAddress-Klasse repräsentiert IP-Adressen
  - diese hat u.a. Methoden “getByName” (klassenbezogene Methode) und “getAddress” (instanzbezogene Methode)
  - UDP ist verbindungslos und nicht zuverlässig (aber effizient)

URL-Verbindungen in Java

- Java bietet einfache Möglichkeiten, auf “Ressourcen” (i.w. Dateien) im Internet mit dem HTTP-Protokoll lesend und schreibend zuzugreifen
  - falls auf diese mittels einer URL verwiesen wird
- Klasse “URL” in java.net.*
  - auf höherem Niveau als die Socket-Programmierung
  - Sockets (mit TCP) werden vor dem Anwender verborgen benutzt
- Beispiel: zeilenweises Lesen einer Textdatei
  - aber auch hier noch diverse Fehlerbedingungen abfangen!

```
// Objekt vom Typ URL anlegen:
URL myURL;
myURL = new URL("http", ..., "/Demo.txt");
...
DataInputStream instream;
instream = new DataInputStream(myURL.openStream());
String line = "";
while((line = instream.readLine()) != null)
  // line verarbeiten
...
```

hier Hostname angeben

Name der Datei

- Es ist auch möglich, Daten an eine URL zu senden
  - POST-Methode, z.B. an ein CGI-Skript
- Ferner: Information über das Objekt ermitteln
  - z.B. Grösse, letztes Änderungsdatum, HTTP-Header etc.

Übungsbeispiel: Ein Bookmark-Checker

(auf den nächsten 2 Seiten)

```
import java.io.*; import java.net.*;
import java.util.Date; import java.text.DateFormat;

public class CheckBookmark {

  public static void main (String args[]) throws
       java.io.IOException, java.text.ParseException {

    if (args.length != 2) System.exit(1);

    // Create a bookmark for checking...
    CheckBookmark bm = new CheckBookmark(args[0], args[1]);

    bm.checkit(); // ...and check

    switch (bm.state) {
      case CheckBookmark.OK:
        System.out.println("Local copy of " +
           bm.url_string + " is up to date"); break;
      case CheckBookmark.AGED:
        System.out.println("Local copy of " +
           bm.url_string + " is aged"); break;
      case CheckBookmark.NOT_SUPPORTED:
        System.out.println("Webserver does not support \
           modification dates"); break;
      default: break;
    }
  }

  String url_string, chk_date;
  int state;

  public final static int OK = 0;
  public final static int AGED = 1;
  public final static int NOT_SUPPORTED = 2;

  CheckBookmark(String bm, String dtm) // Constructor
  { url_string = new String(bm);
    chk_date = new String(dtm);
    state = CheckBookmark.OK;
  }
```



```
  public void checkit() throws java.io.IOException,
       java.text.ParseException {

    URL checkURL = null;
    URLConnection checkURLC = null;

    try { checkURL = new URL(this.url_string); }
    catch (MalformedURLException e) {
      System.err.println(e.getMessage() + ": Cannot \
          create URL from " + this.url_string);
      return;
    }

    try {
      checkURLC = checkURL.openConnection();
      checkURLC.setIfModifiedSince(60);
      checkURLC.connect();
    }

    catch (java.io.IOException e) {
      System.err.println(e.getMessage() + ": Cannot \
          open connection to " + checkURL.toString());
      return;
    }

    // Check whether modification date is supported
    if (checkURLC.getLastModified() == 0) {
      this.state = CheckBookmark.NOT_SUPPORTED;
      return;
    }

    // Cast last modification date to a "Date"
    Date rem = new Date(checkURLC.getLastModified());

    // Cast stored date of bookmark to Date
    DateFormat df = DateFormat.getDateInstance();
    Date cur = df.parse(this.chk_date);

    // Compare and set flag for outdated bookmark
    if (cur.before(rem)) this.state = CheckBookmark.AGED;
  }
}
```