Adressierung

- *Sender* muss in geeigneter Weise spezifizieren, wohin die Nachricht gesendet werden soll
  - ggf. mehrere Adressaten zur freien Auswahl (Lastverteilung, Fehlertoleranz)
  - ggf. mehrere Adressaten gleichzeitig (Broadcast, Multicast)

- *Empfänger* ist ggf. nicht bereit, jede beliebige Nachricht von jedem Sender zu akzeptieren
  - selektiver Empfang (Spezialisierung)
  - Sicherheitsaspekte, Überlastabwehr

- Probleme
  - *Ortstransparenz*: Sender weiss *wer*, aber nicht *wo* (sollte er i.a. auch nicht!)
  - *Anonymität*: Sender und Empfänger kennen einander zunächst nicht (sollen sie oft auch nicht)

Kenntnis von Adressen?

- Adressen sind u.a. Rechneradressen (z.B. IP-Adresse oder Netzadresse auf Ethernet-Basis), Portnamen, Socketnummern, Referenzen auf Mailboxes...

- Woher kennt ein Sender die Adresse des Empfängers?

  1) Fest in den Programmcode integriert → unflexibel

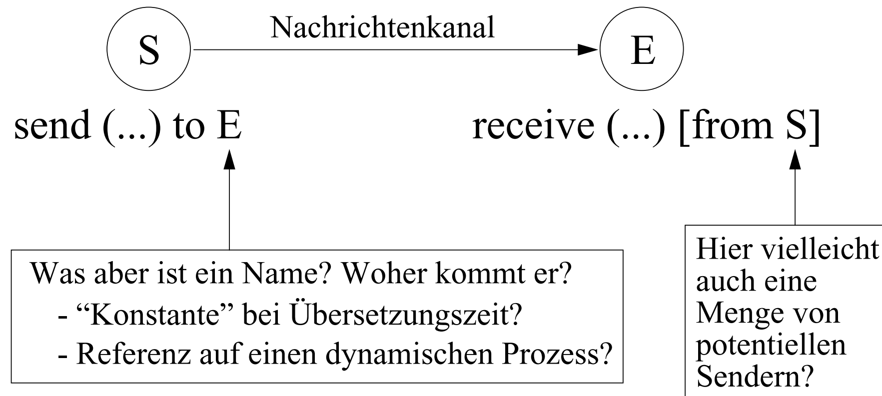
  2) Über Parameter erhalten oder von anderen Prozessen mitgeteilt

  3) Adressanfrage per Broadcast “in das Netz”
     - häufig bei LANs: Suche nach lokalem Nameserver, Router etc.

  4) Auskunft fragen (Namensdienst wie z.B. DNS; Lookup-Service)
     - wie realisiert man dies effizient und fehlertolerant?

Direkte Adressierung

- *Direct Naming* (1:1-Kommunikation):

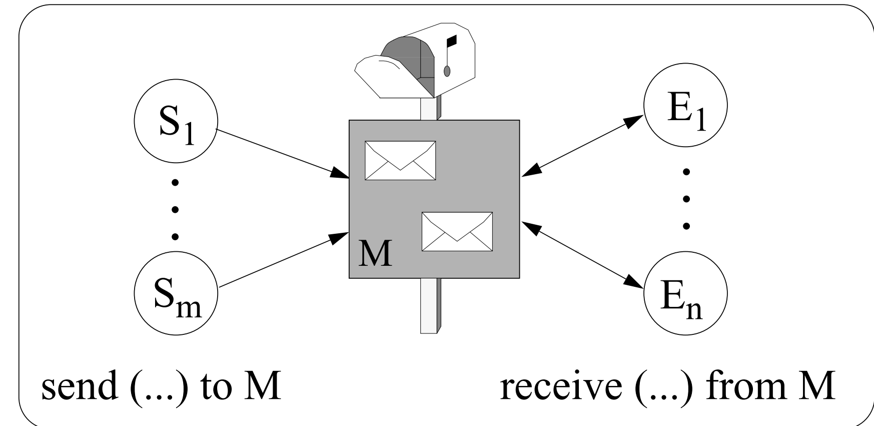


- Direct naming ist insgesamt relativ unflexibel

- Empfänger (= Server) sollten nicht gezwungen sein, potentielle Sender (= Client) explizit zu nennen
  - Symmetrie ist also i.a. gar nicht erwünscht

Indirekte Adressierung - Mailbox

- ermöglicht m:n-Kommunikation



- Eine Nachricht besitzt i.a. mehrere potentielle Empfänger
  - Mailbox spezifiziert damit eine *Gruppe* von Empfängern

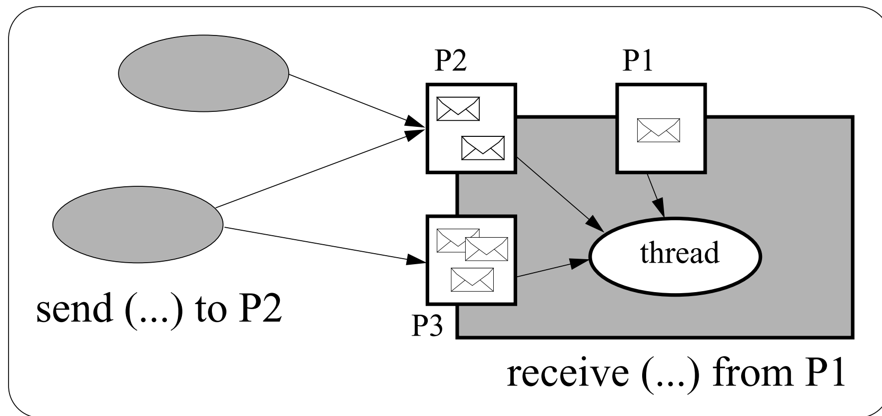
- Kann jeder Empfänger die Nachricht bearbeiten?
  - Mailbox i.a. typisiert: nimmt nur bestimmte Nachrichten auf
  - Empfänger kann sich u.U. Nachrichten der Mailbox ansehen / aussuchen...
  - aber wer garantiert, dass jede Nachricht irgendwann ausgewählt wird?

- Wo wird die Mailbox angesiedelt? (→ Implementierung)
  - als ein einziges Objekt auf irgendeinem (geeigneten) Rechner?
  - repliziert bei den Empfängern? Abstimmung unter den Empfängern notwendig (→ verteiltes Cache-Kohärenz-Problem)
  - Nachricht verbleibt in einem Ausgangspuffer des Senders: Empfänger müssen sich bei allen (welche sind das?) potentiellen Sendern erkundigen

- Mailbox muss gegründet werden: Wer? Wann? Wo?

Indirekte Adressierung - Ports

- m:1-Kommunikation
- Ports sind Mailboxes mit genau einem Empfänger
  - Port gehört diesem Empfänger
  - Kommunikationsendpunkt, der die interne Empfängerstruktur abkapselt
- Ein Objekt kann i.a. mehrere Ports besitzen

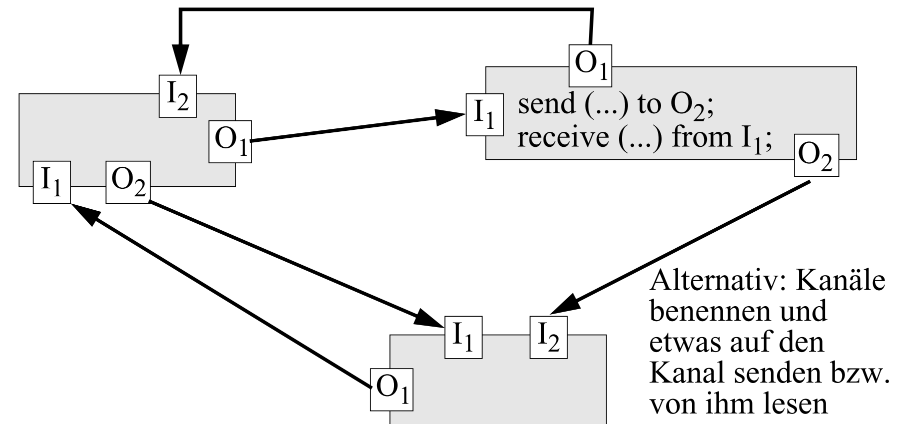


Pragmatische Aspekte (Sprachdesign etc.):

- Sind Ports statische oder dynamische Objekte?
- Wie erfährt ein Objekt den Portnamen eines anderen (dynamischen) Objektes?
  - können Namen von Ports verschickt werden?
- Sind Ports typisiert?
  - unterstützt den selektiven Nachrichtenempfang
- Grösse des Nachrichtenpuffers?
- Können Ports geöffnet und geschlossen werden?
  - genaue Semantik?

Kanäle und Verbindungen

- Neben *Eingangsports* (“in-port”) lassen sich auch *Ausgangsports* (“out-port”) betrachten



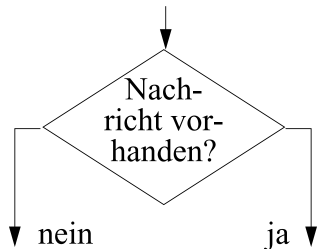
- Ports können als Ausgangspunkte für das Einrichten von *Verbindungen* (“Kanäle”) gewählt werden
- Dazu werden je ein in- und out-Port miteinander verbunden. Dies kann z.B. mit einer connect-Anweisung geschehen: **connect** p1 **to** p2
  - denkbar sind auch broadcastfähige Kanäle
- Die Programmierung und Instanziierung eines Objektes findet so in einer anderen Phase statt als die Festlegung der Verbindungen
- Grössere Flexibilität durch die dynamische Änderung der Verbindungsstruktur
- Kommunikationsbeziehung: wahlweise 1:1, n:1, 1:n, n:m

Konfigurationsphase

Varianten beim Empfangen von Nachrichten - Nichtblockierung

- Typischerweise ist ein “receive” blockierend

- Aber auch *nichtblockierender* Empfang denkbar:



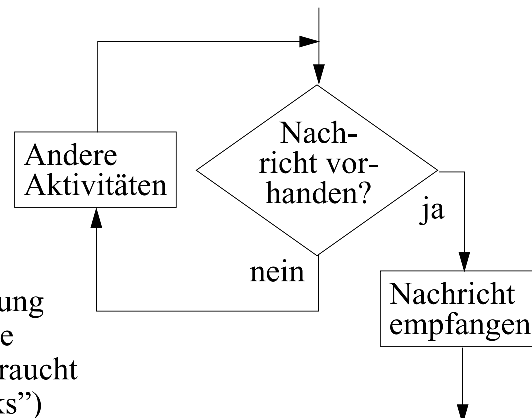
- “Non-blocking receive”

- Sprachliche Realisierung z.B. durch “Returncode” eines als Funktionsaufruf benutzten “receive”

- Aktives Warten:
  (“busy waiting”)

  - Nachbildung des blockierenden Wartens wenn “andere Aktivitäten” leer

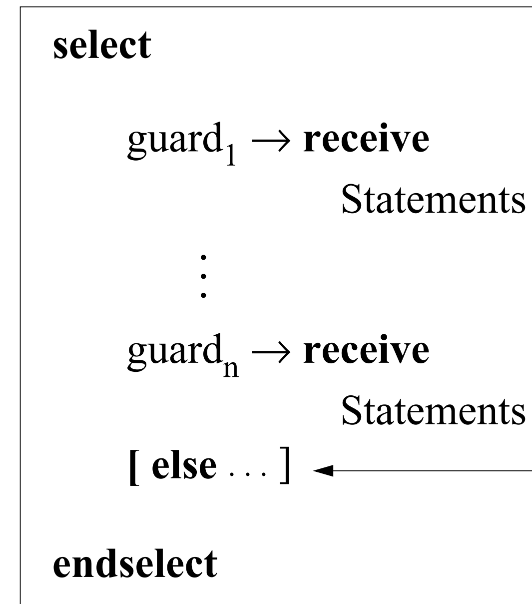
  - Nur für kurze Wartezeiten sinnvoll, da Monopolisierung der cpu, die ggf. für andere Prozesse oder threads gebraucht werden könnte (“spin locks”)



- Weitere Möglichkeit: unterbrechungsgesteuertes (“asynchrones”) Empfangen der Nachricht (→ nicht unproblematisch!)

Alternatives Empfangen

- Sprachliche Realisierung z.B. so:



else-Fall wird gewählt, wenn kein guard ‘true’ ist bzw. dort momentan keine Nachricht anliegt

- Aktives Warten durch umschliessende while-Schleife

  - bei else könnte im Fall von aktivem Warten die while-Bedingung auf false gesetzt werden, falls das Warten abgebrochen werden soll, oder es könnte mittels timer (“wait”) eine kurze Zeit gewartet werden...

  - else-Fall kann auch einfach das leere Statement enthalten

- Typischerweise *nichtblockierend*; aber blockierend, wenn else-Alternative ganz fehlt

Zeitüberwachte Kommunilation

- Empfangsanweisung soll maximal (?) eine gewisse Zeit lang blockieren ("timeout")
  - z.B. über return-Wert abfragen, ob Kommunikation geklappt hat

- Sinnvoll bei:
  - Echtzeitprogrammierung
  - Vermeidung von Blockaden im Fehlerfall (etwa: abgestürzter Kommunikationspartner)

    → dann sinnvolle Recovery-Massnahmen treffen ("exception")
    → timeout-Wert "sinnvoll" setzen!

    Quelle vielfältiger Probleme...

- Timeout-Wert = 0 kann ggf. genutzt werden, um zu testen, ob eine Nachricht "jetzt" da ist

---

- Analog ggf. auch für synchrones (!) *Senden* sinnvoll

  → Verkompliziert zugrundeliegendes Protokoll: Implizite Acknowledgements kommen nun "asynchron" an...

Zeitüberwachter Nachrichtenempfang

- Möglicher Realisierung:
  - Durch einen Timer einen *asynchronen Interrupt* aufsetzen und Sprungziel benennen
  - Sprungziel könnte z.B. eine Unterbrechungsroutine sein, die in einem eigenen Kontext ausgeführt wird, oder das Statement nach dem receive

  → "systemnahe", unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

- Sprachliche Einbindung besser z.B. so:

receive ... **delay** t ← Blockiert maximal t Zeiteinheiten (Vorsicht!)

select
    guard$_1$ → **receive** ...
    ⋮
    delay t → Anweisungen ... ← Wird nach *mind.* t Zeiteinheiten ausgeführt, wenn bis dahin noch keine Nachricht empfangen
endselect

- Genaue Semantik beachten: Es wird *mindestens* so lange auf Kommunikation gewartet. Danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Programms verstreichen!

- Frage: sollte "delay 0" äquivalent zu "else" sein?

Selektives Empfangen

≠ alternatives!

- Bedingung an den *Inhalt* (bzw. Typ, Format,...) der zu empfangenden Nachricht

- Dadurch werden gewisse ("unpassende") Nachrichten einfach ausgeblendet

- Bedingung wird oft vom aktuellen Zustand des Empfängers abhängen

---

- Vorteil bei der Anwendung:

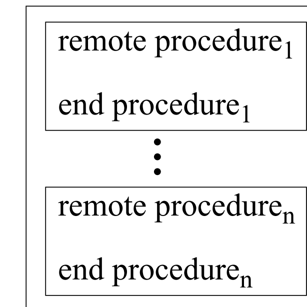
  Empfänger muss nicht mehr alles akzeptieren und zwischenspeichern, sondern bekommt nur das, was ihn momentan interessiert

Implizites Empfangen

- Keine receive, select...-Anweisung, sondern Spezifikation von Routinen, die bei Vorliegen einer Nachricht ausgeführt ("angesprungen") werden

- z.B. RPC:

bzw. asynchrone Variante oder "Remote Method Invocation" bei objektorientierten Systemen



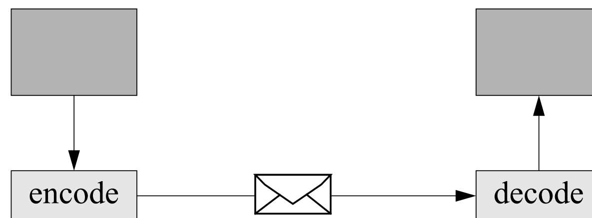
- Analog auch für den "Empfang" einer Nachricht *ohne Antwortverpflichtung* denkbar

---

- *Semantik*:
  - Interne Parallelität?
    - Mehr als eine gleichzeitig aktive Prozedur, Methode, thread... ("Routine") im Empfänger?
    - Vielleicht sogar mehrere Instanzen der gleichen Routine?
  - Atomare Routinen?
    - Wird eine aktive Routine ggf. unterbrochen, um eine andere aktivierte auszuführen?

Kommunizierbare Datentypen?

- Werte von “klassischen” einfachen Datentypen
  - int, character, string, floating point,...
- Kompatibilität in heterogenen Systemen
  - Grösse von int?
  - Format von floating point?
  - höherwertiges Bit links oder rechts?
  - ...



- Vereinbarung einer *Standardrepräsentation* (z.B. XDR)
- marshalling (encode / decode) kostet Zeit

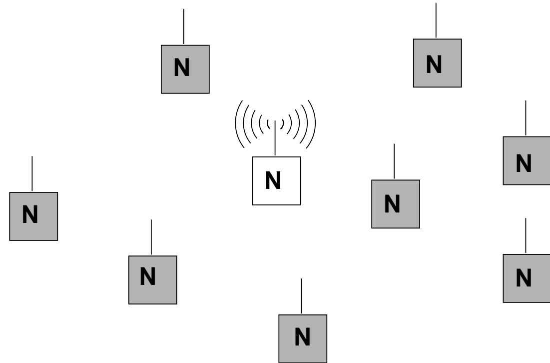
- Was ist mit *komplexen Datentypen* wie
  - Records, Strukturen
  - Objekte
  - Referenzen, Zeiger
  - Zeigergeflechte

  - sollen Adressen über Rechner- / Adressraumgrenzen erlaubt sein?
  - sollen Referenzen symbolisch, relativ... interpretiert werden? Ist das stets möglich?
  - wie wird Typkompatibilität sichergestellt?
- Ggf. “Linearisieren” und ggf. Strukturbeschreibung mitschicken (u.U. “sprachunabhängig”)
- Sind (Namen von) Ports, Prozessen... eigene Datentypen, deren Werte versendet werden können?

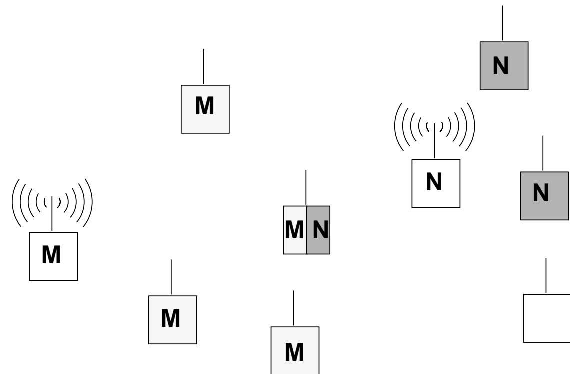
“first class objects”

Gruppen-kommunikation

Gruppenkommunikation



Broadcast: Senden an die *Gesamtheit* aller Teilnehmer

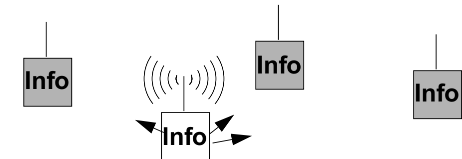


Multicast: Senden an eine *Untergruppe* aller Teilnehmer

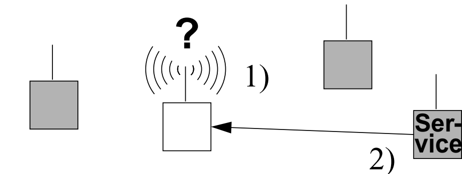
- entspricht Broadcast bezogen auf die Gruppe
- verschiedene Gruppen können sich ggf. überlappen
- jede Gruppen hat eine Multicastadresse

Anwendungen von Gruppenkommunikation

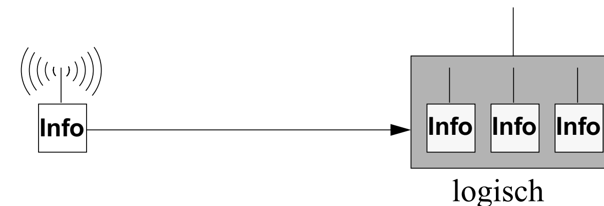
- *Informieren*
  - z.B. Newsdienste, Konferenzsysteme etc.



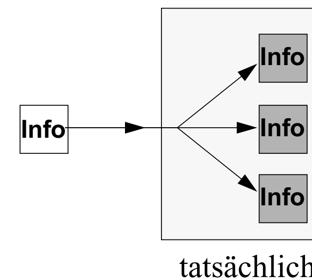
- *Suchen*
  - z.B. Finden von Objekten und Diensten



- *"Logischer Unicast"* an replizierte Komponenten



logisch



tatsächlich

Typische Anwendungsklasse von Replikation: Fehlertoleranz