Softwarearchitektur

(Architektur: *αρχή* = Anfang, Ursprung + *tectum* = Haus, Dach)

12. Fallstudien eingebettete Systeme

Vorlesung Wintersemester 2010 / 11

Technische Universität München
Institut für Informatik
Lehrstuhl von Prof. Dr. Manfred Broy

Dr. Klaus Bergner, Prof. Dr. Manfred Broy, Dr. Marc Sihling

Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

Zusammenfassung der letzten Vorlesung

- Im Bereich der eingebetteten Systeme gibt es eine Fülle von Modellierungs- und Realisierungstechniken sowie technischen Infrastrukturen und Basiskomponenten.
- Die Standardisierung ist allerdings nicht so fortgeschritten wie bei betrieblichen Informationssystemen.
  - Höherer Kostendruck - höhere Optimierung
- In der Praxis gibt es meist einen Methodenbruch zwischen fachlicher Architektur und technischer Architektur (oder schlimmer: eine fachliche Architektur wird überhaupt nicht erstellt).
- Um ein System realisieren zu können, muss der Entwickler heutzutage in fast allen Fällen über ein umfassendes und detailliertes Verständnis der technischen Infrastruktur (inklusive Hardware) verfügen.
- Neue Ansätze basierend auf Generierung aus Modellen und Container-Architekturen sind größtenteils erst noch in der Forschung.

Fachliche und Technische Architektur

- Fachliche Architektur
  - Die Modellierung basiert auf verteilten Objekten/Komponenten, die sich gegenseitig Nachrichten schicken. Zur Spezifikation werden hauptsächlich verwendet:
    - Instanzendiagramme für die Systemkonfigurationen
    - Zustandsdiagramme für die einzelnen Komponenten
    - Sequenzdiagramme für die Interaktion zwischen Komponenten
  - Bei geschlossenen Systemen ist die Systemkonfiguration zur Laufzeit in aller Regel statisch.
- Technische Architektur
  - Hauptkonzepte und Hauptmechanismen sind
    - Parallel oder quasiparallel ausgeführte Tasks (Prozesse, Threads)
    - Kommunikationsmechanismen zwischen Tasks (Semaphore, Signale, Queues, Mailboxes, Shared Memory, RPCs, Feldbusse)
    - Basiskomponenten wie RT-Betriebssysteme und RT-ORBs und Bus-Systeme

Entwicklungsschritte bei der Festlegung der Fachlichkeit

- Business Requirements
  - meist vorgegeben durch Firmenstrategie und Marketing
- Auswahl der Basisfunktionalität
  - Auswahl Funktionen, die durch das System erbracht werden
- Festlegung der nichtfunktionalen Anforderungen
  - Qualitätsanforderungen - Architektur

Entwurfsschritte bei der Abbildung der Fachlichkeit

- Auswahl der Hardware
  - meist vorgegeben oder stark eingeschränkt durch Kosten, Größe, Stromverbrauch, Performance
- Auswahl der Basissoftware
  - Auswahl von Basiskomponenten, insbesondere eines RT-OS (falls überhaupt geeignete Komponenten existieren)
- Entwurf der Tasks (Threads, Runnables)
  - Identifikation der Tasks
  - Zuordnung von Tasks zu Prozessoren (damit implizit auch Zuordnung von Kommunikationsverbindungen zu Kommunikationsmedien)
  - Zuordnung von Objekten zu Tasks
- Entwurf der Interaktion und Kommunikation
  - Auswahl und Entwurf der Synchronisations- und Kommunikations-mechanismen
  - Zuordnung von Prioritäten zu Tasks

Erinnerung: Tasks und Multi-tasking

- Ein Task (englisch für Aufgabe) ist ein Prozess, der auf der Systemebene (Kernel) läuft.
- Task wird somit mit Prozess gleichgesetzt.
  - Oft wird ein Task auch als Subprozess angesehen, stellt also einen Thread dar.
- Durch die weiteren Begriffe Multitasking und Taskwechsel, welche sich sowohl auf Prozesse als auch auf Threads beziehen können, sollte Task also eher als Oberbegriff für Prozess und Thread gesehen werden.
- Multitasking nennt man allgemein die Technik, mehrere Prozesse (oder Threads) gleichzeitig oder quasi-gleichzeitig (d.h. in Form eines sehr eng verzahnten Hin- und Herschaltens) laufen zu lassen.
- Es werden zwei Arten von Multitasking unterschieden:
  1. Kooperatives Multitasking
  2. Präemptives Multitasking

Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

Basisarchitekturen

- Es gibt einige grundlegende Architekturen für geschlossene lokale Echtzeit-Systeme, die sich in ihrer Komplexität und in den erzielbaren Reaktionszeiten unterscheiden.
- Einfache Systeme, die auf wenige Ereignisse reagieren müssen und/ oder geringe Anforderungen an die Reaktionszeiten und die Behandlung von Prioritäten haben, kommen mit einer einfachen Architektur und elementaren Betriebssystemkonzepten aus.
- Die vier im Folgenden vorgestellten Basisarchitekturen sind (nach zunehmender Komplexität angeordnet):
  - Round-Robin-Architektur
  - Round-Robin-Architektur mit Interrupts
  - Task-Queue Scheduling-Architektur
  - RTOS-Architektur

Klassifikation und Beispiele nach [Si99]

Round-Robin-Architektur

Grundprinzip

- Ein einziger Task fragt in einer Hauptschleife synchron der Reihe nach alle Ereignisquellen ab und ruft jeweils die entsprechenden Operationen für die Behandlung der Ereignisse auf.
- Operationen können auch unbedingt (ohne zu Grunde liegendes externes Ereignis) aufgerufen werden.
- Die maximale Reaktionszeit für die Ausführung einer Operation ist jeweils durch die maximale Ausführungszeit der Hauptschleife vorgegeben.

Beispielcode

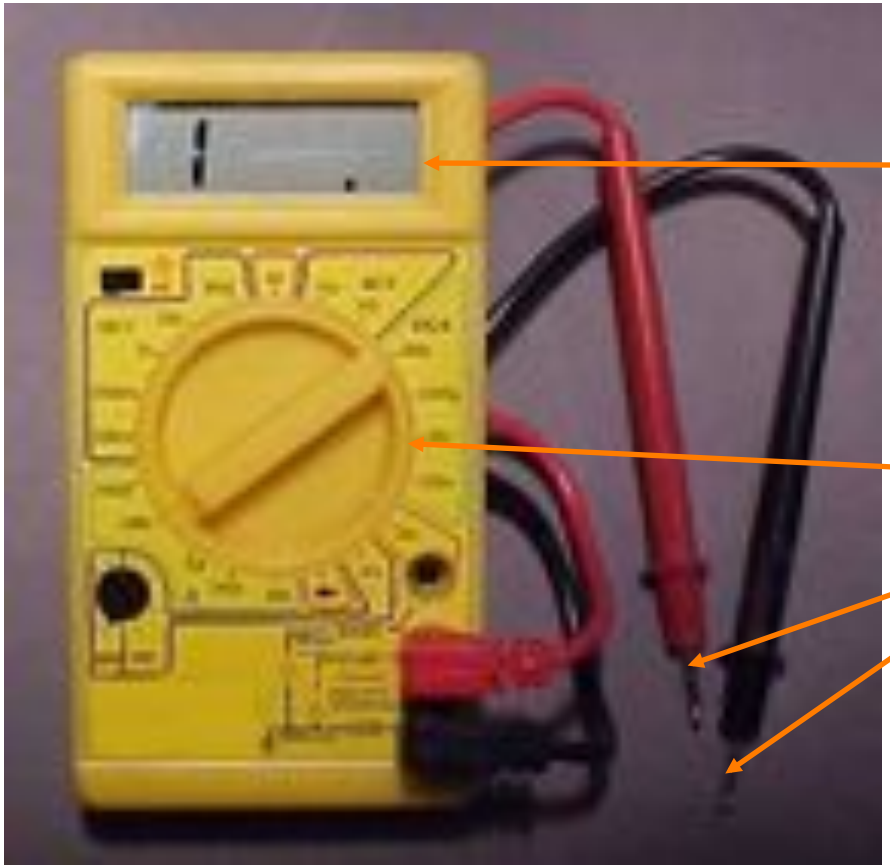
```
void main(void) {
  while (true) {
    if ( <EreignisA> ) {
      // behandle EreignisA
    }
    if ( <EreignisB> ) {
      // behandle EreignisB
    }

    ...

    if ( <EreignisZ> ) {
      // behandle EreignisZ
    }
  }
}
```

Beispiel für möglichen Einsatz

Digitales Multimeter



In jedem Durchlauf:

- Aktualisierung der Anzeige
- Auslesen der Ereignisquellen
  - Drehschalterposition
  - Messergebnis

Vor- und Nachteile der Round-Robin-Architektur

- Vorteile
  - Sehr einfach zu implementieren
  - Benötigt keine Basis-Software
  - Geeignet für ressourcenbeschränkte (Single-Prozessor-)Geräte
- Nachteile
  - Wenn einzelne Ereignisbehandlungen zu lange brauchen, kann die Architektur unpraktikabel oder untauglich sein.
  - Wenn die geforderten Reaktionszeiten auf Ereignisse kürzer als die Zeit für die Ausführung der Hauptschleife sind, gehen Ereignisse verloren oder werden zu spät behandelt.
  - Die Architektur ist nur bis zu einem bestimmten Punkt erweiterbar: Wenn neue Ereignisquellen und Ereignis-behandlungsoperationen hinzukommen, kann die Architektur versagen.

Round-Robin-Architektur mit Interrupts

- Im Vergleich zu Round-Robin gibt es zusätzlich Interrupt-Handler. Sie werden hardwareunterstützt asynchron aufgerufen, wenn der Prozessor bestimmte Ereignisse feststellt.
- In den Interrupt-Handlern läuft nur der Code, der für die schnelle Reaktion auf Ereignisse erforderlich ist. Interrupts (und damit die Interrupt-Handler) können unterschiedliche Prioritäten haben.
- Wenn kein Interrupt-Handler aktiv ist, ruft der Hauptschleifen-Task Task-Operationen für die weitere Verarbeitung auf.
  - Die Task-Operationen basieren auf den Datenobjekten, die von den Interrupt-Handlern abgelegt worden sind.
  - Damit ergibt sich das Problem, dass sowohl Interrupt-Handler als auch Task-Operationen koordiniert auf die gleichen Datenobjekte zugreifen müssen. Dies kann durch kurzzeitiges Ausschalten der Interrupts in der Hauptschleife oder durch reentrante Datenstrukturen gelöst werden.
  - In der Task-Hauptschleife können auch länger dauernde Operationen ausgeführt werden, ohne die Reaktionszeit zu beeinträchtigen.

Code bei Round-Robin-Architektur mit Interrupts

Interrupt-Handler

```
boolean E1 = false;
boolean E2 = false;
...

void interrupt handleE1() {
  // behandle E1
  E1 = true;
}
void interrupt handleE2() {
  // behandle E2
  E2 = true;
}
...
```

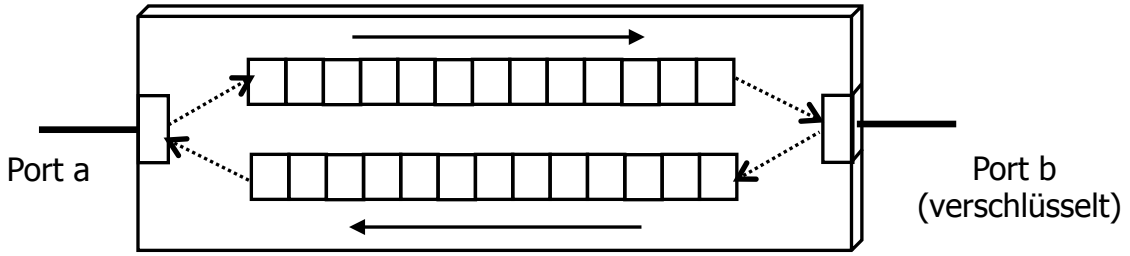
Task-Hauptschleife

```
void main(void) {
  while (true) {
    if (E1) {
      E1 = false;
      // behandle E1
    }
    if (E2) {
      E2 = false;
      // behandle E2
    }
    ...
}}
```

Welcher Code in den Interrupt-Handlern bzw. in den Task-Operationen der Task-Hauptschleife liegt, ist eine Entwurfsentscheidung.

Beispiel für möglichen Einsatz

Encryption Bridge



- Eingabezeichen an den Ports müssen innerhalb einer definierten Reaktionszeit gelesen und in die intern verwalteten Queues geschrieben werden (bevor das nächste Eingabezeichen ankommt). Dies geschieht innerhalb von Interrupt-Handlern.
- Die Operationen zum Anfügen und Auslesen von Zeichen an die / aus den Queues sind reentrant, damit sie sowohl von den Interrupt-Handlern als auch von der Hauptschleife gerufen werden können.
- Die Ver- und Entschlüsselungsroutinen arbeiten in der Hauptschleife zeitlich entkoppelt von den Interrupt-Handlern.

Vor- und Nachteile von Round-Robin mit Interrupts

- Vorteile
  - Relativ einfach zu implementieren
  - Benötigt keine Basis-Software
  - Geeignet für ressourcenbeschränkte (Single-Prozessor-)Geräte
  - Gut erweiterbar mit zusätzlichen Interrupt-Handlern.
  - Interrupt-Handler können unterschiedliche Prioritäten besitzen
- Nachteile
  - Zugriff auf gemeinsam genutzte Daten von Interrupt-Handlern und Hauptschleife muss von Hand koordiniert werden
  - Es gibt keine Prioritäten für den Task-Code in der Hauptschleife. Die maximale Reaktionszeit für eine Task-Operation ist gleich der Summe der Ausführungszeiten aller Task-Operationen in der Hauptschleife (plus zusätzlich auftretende Zeit für Interrupts).

Task-Queue Scheduling Architecture

- Diese Architektur ist eine Erweiterung der Round-Robin-Architektur mit Interrupts.
- Anstatt den Task-Code in einer Hauptschleife immer in der gleichen Reihenfolge auszuführen, stellen die Interrupt-Handler Bearbeitungsanforderungen für Task-Operationen in eine Task-Queue ein.
- Die Bearbeitungsanforderungen werden gesteuert von einem Scheduler abgearbeitet. Dies geschieht nicht notwendigerweise in der Reihenfolge, in der sie in die Task-Queue eingestellt wurden.
- Die Vor- und Nachteile entsprechen im Wesentlichen denen der Round-Robin-Architektur mit Interrupts, mit folgenden Änderungen:
  - Zusätzliche Komplexität durch nötige Realisierung des Schedulers.
  - Senkung der maximalen Reaktionszeit für Task-Operationen in der Hauptschleife (Ausführungszeit der längsten Task-Operation, plus zusätzlich auftretende Zeit für Interrupts).

RTOS (real-time operating system) Betriebssystem

- Ein Echtzeitbetriebssystem oder auch RTOS (real-time operating system) ist ein Betriebssystem mit
  - zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitbedingungen (vorhersagbares Zeitverhalten - dies betrifft vor allem die Bereiche Scheduling und Speicherverwaltung) und die Vorhersagbarkeit des Prozessverhaltens.
  - Echtzeitbetriebssysteme müssen zusätzliche Fehlererkennungsmechanismen unterstützen.
- Gängige Architekturen
  - Micro-Kernel: Diese Architektur basiert darauf, dass der eigentliche Betriebssystemkern als Task mit niedrigster Priorität realisiert ist und der Echtzeit-Kernel das Scheduling übernimmt. Dabei besitzen die Echtzeit-Prozesse die höchste Priorität. Das bringt minimale Latenzzeiten mit sich.
  - Nano-Kernel: Ähnlich dem Micro-Kernel-Ansatz, jedoch besteht hier die Möglichkeit, neben dem eigentlichen Echtzeit-Kernel eine beliebige Anzahl anderer Betriebssystem-Kernel laufen zu lassen.

RTOS-Architektur

- Die RTOS-Architektur basiert auf einem Echtzeit-Betriebssystem (**R**eal-**T**ime **O**perating **S**ystem).
- Wie in den beiden vorigen Architekturen wird zwischen Code in Interrupt-Handlern und Code in Tasks unterschieden.
- Die Kommunikation zwischen Interrupt-Handlern und Task-Code erfolgt über die Mechanismen des Betriebssystems (explizit über Signale, Queues, Mailboxes etc. oder implizit über Shared Memory).
- Anstatt alle Task-Operationen in einer Hauptschleife eines Tasks zu realisieren, werden sie nun in eigenständige Tasks verlagert.
- Das Betriebssystem kann einen Task während seiner Laufzeit unterbrechen, um zu einem anderen, höher priorisierten zu wechseln.
  - Damit lässt sich die Reaktionszeit für Task-Operationen auf die Zeit senken, die das RTOS für den Wechsel zu dem betreffenden Task benötigt.

Code bei RTOS-Architektur

Interrupt-Handler

```
void interrupt handleE1() {
  // behandle E1
  setSignal(E1);
}

void interrupt handleE2() {
  // behandle E2
  setSignal(E2);
}

...
```

Tasks

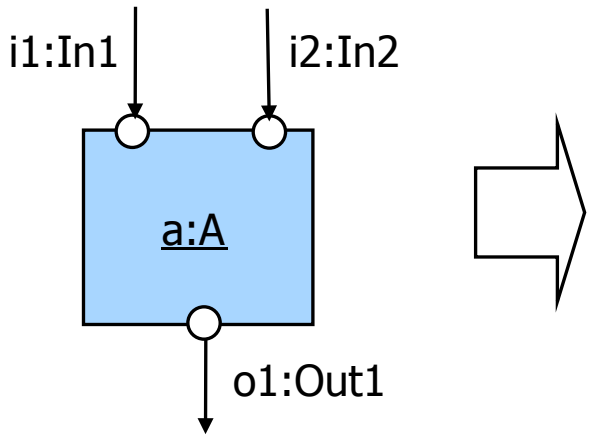
```
void task1(void) {
  while (true) {
    waitForSignal(E1);
    // behandle E1
}}

void task2(void) {
  while (true) {
    waitForSignal(E2);
    // behandle E2
}}
```

Anstelle einfacher Signale können auch andere Mechanismen (etwa Nachrichtenversand über eine Queue) Task-Operationen auslösen.

Aufbau eines Tasks bei der RTOS-Architektur

Verteilte Objekte aus der fachlichen Architektur deren Verhalten durch einen Zustandsautomat beschrieben wird, lassen sich schematisch in Tasks umsetzen.




```
void taskA(void) {
  // Deklaration privater Daten

  // Initialisierung, falls
  // erforderlich

  while (true) {
    // auf Ereignis warten:
    // Signal, Message, ...

    switch (<Ereignis>) {
      case <i1> :
        ... send(<o1>) ...
      case <i2> :
        ...
}}}
```

Entwurf der Tasks bei der RTOS-Architektur

- So viele Tasks wie möglich, um
  - eine direkte Abbildung der fachlichen Architektur zu ermöglichen (ein Task pro verteiltem Objekt der fachlichen Architektur),
  - die Modularität und Flexibilität der Architektur zu erhöhen,
  - Datenobjekte in Tasks kapseln zu können,
  - den Zugriff auf Hardware-Geräte kapseln zu können,
  - differenziertere Prioritäten zu ermöglichen,
  - die geforderten Reaktionszeiten einhalten zu können.
- So wenige Tasks wie möglich, um
  - die Kosten für die Kommunikation zwischen Tasks bzw. den Zugriff auf gemeinsame Datenobjekte zu minimieren,
  - den Verwaltungsaufwand für den Task-Wechsel zu minimieren,
  - Speicherplatz für Stacks zu sparen,
  - Test und Debugging zu vereinfachen.

In der Praxis: So wenige Tasks wie möglich, so viele wie nötig.

Vor- und Nachteile der RTOS-Architektur

- Vorteile
  - Einfache und klare Implementierung
  - Senkung der Reaktionszeiten für Task-Operationen im Vergleich zu den drei anderen Architekturen durch erhöhte Asynchronität
  - Erweiterbarkeit durch modulares Hinzufügen zusätzlicher Tasks und Interrupt-Handler (mit relativ stabilen Reaktionszeiten)
  - Möglichkeit der Kommunikation zwischen Tasks (und Interrupt-Handlern) über Nachrichten statt über Zugriff auf gemeinsame Daten
  - Code unverändert sowohl auf Single-Prozessor-Rechnern als auch auf Multi-Prozessor-Rechnern ausführbar
- Nachteile
  - Benötigt Echtzeitbetriebssystem
    - Erhöhter Speicherbedarf
    - Erhöhte Softwarekosten bzw. Aufwand für Eigenentwicklung
    - Aufwand für Task-Wechsel und RTOS-Aufgaben führt insgesamt zu niedrigerem Durchsatz (bzw. erfordert schnelleren Prozessor)

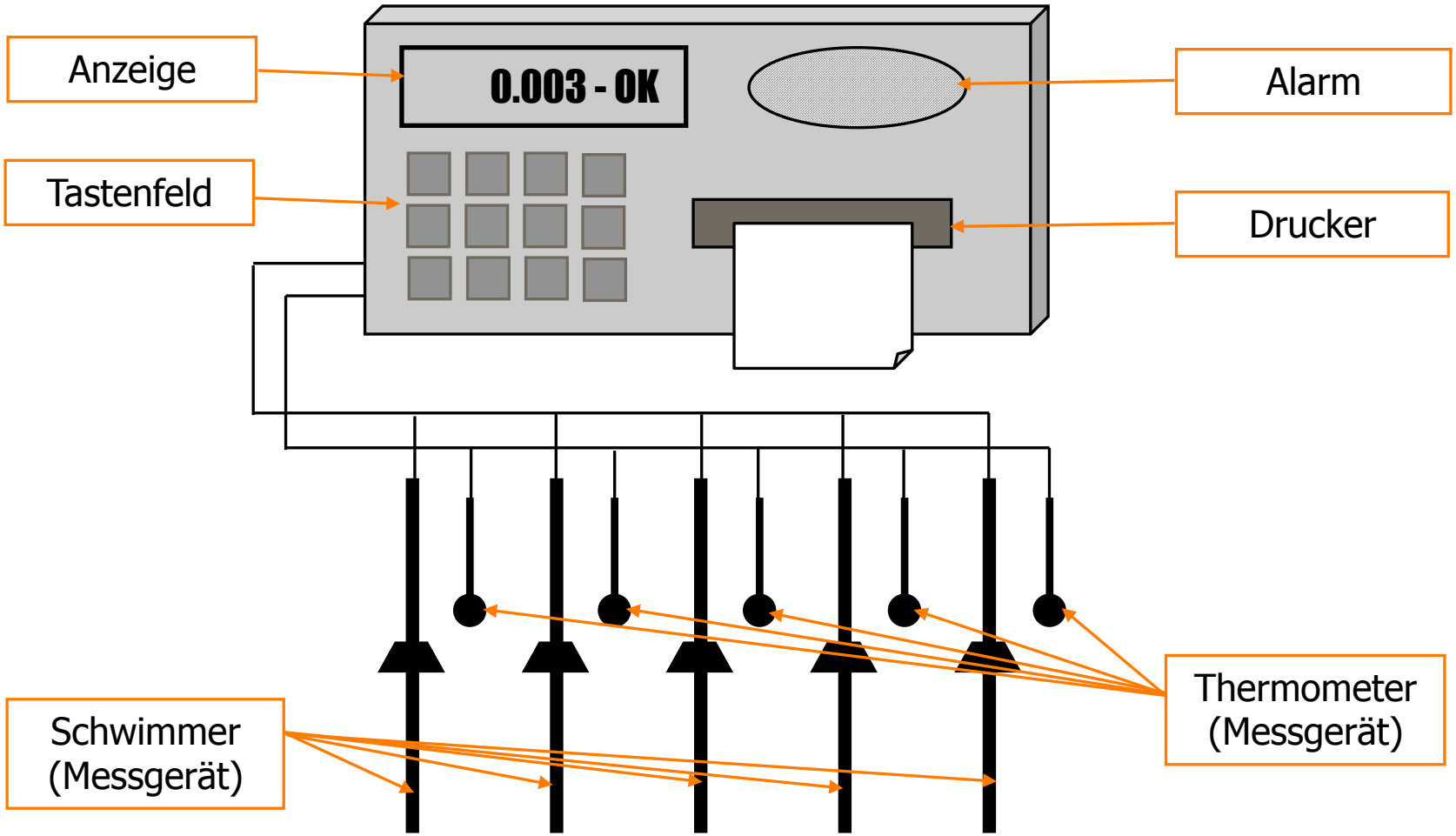
Vergleich der Basisarchitekturen

	Prioritäten	**schlechteste Reaktionszeit für Task-Operationen**	**Stabilität der Reaktionszeiten bei Änderungen des Codes**	**Einfachheit**
Round-Robin	keine	Ausführungszeit sämtlichen Codes	schlecht	sehr einfach
Round-Robin mit Interrupts	Prioritäten für Interrupt-Handler, danach alle Task-Operationen gleich	Ausführungszeit aller Task-Operationen (plus Interrupt-Handler)	gut für Interrupt-Handler, schlecht für Task-Operationen	einfach, aber koordinierter Zugriff auf gemeinsame Datenobjekte
Task-Queue Scheduling	Prioritäten für Interrupt-Handler und danach für Task-Operationen	Ausführungszeit der längsten Task-Operation (plus Interrupt-Handler)	relativ gut	einfach, aber Scheduler und koordinierter Zugriff auf gemeinsame Datenobjekte
RTOS	Prioritäten für Interrupt-Handler und danach für Task-Operationen	keine (plus Interrupt-Handler)	sehr gut	komplex (allerdings versteckt im RTOS)

Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

Underground Tank Monitoring System

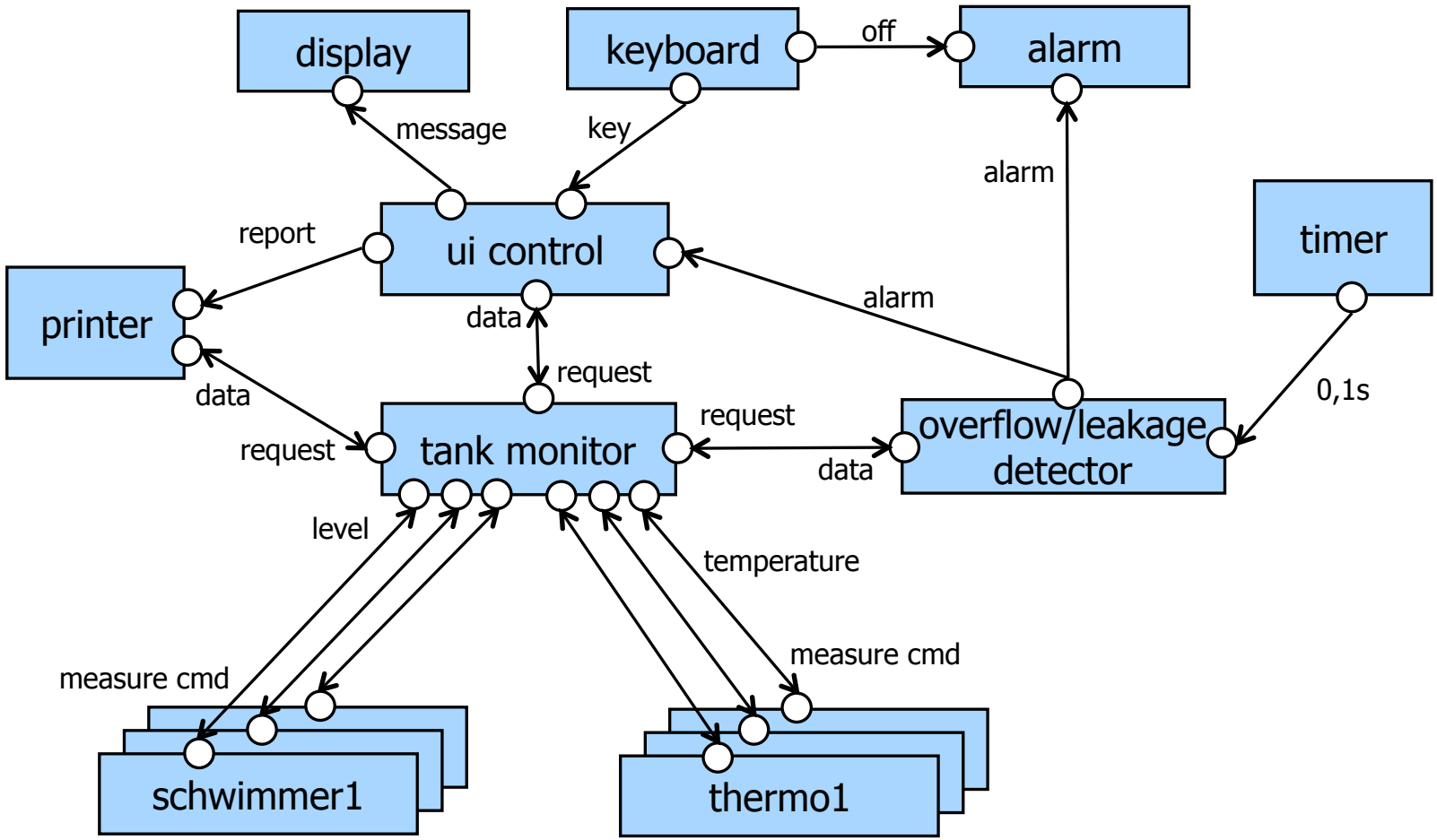


Beispiel nach [Si99]

Anforderungen: Überwachung von bis zu acht Tanks

- Berechnung des Tankinhalts auf Basis von Füllstand und Temperatur
  - Kommandogesteuertes Messen des Füllstands für einen Tank
  - Auslesen der Temperaturdaten zu einem Tank
- Kontinuierliche Überwachung der Tankstände
  - Erkennung von Lecks (wenn Inhalt über Stunden hinweg langsam sinkt)
  - Überlaufgefahr, wenn Füllstand rapide steigt
- Ein- und Ausgabe über Tastenfeld und Anzeige
  - Eingabe von Befehlssequenzen aus mehreren Tastendrücken
  - einfache Menüsteuerung sowie Ausgabe von Alarm-Meldungen
- Druck-Ausgabe
  - zum Ausdrucken von Füllstandshistorien etc.
- Alarmglocke
  - Auslösung bei Erkennung eines Lecks oder bei Überlaufgefahr
  - eigene Taste auf dem Tastenfeld zum Abschalten des letzten Alarms

Fachliche Architektur - Instanzensicht



Sonstige Anforderungen

- Hardware-Aufbau
  - Relativ langsamer 8-bit-Mikrocontroller
  - Interrupts treffen ein
    - von den Schwimmern nach Senden eines Mess-Kommandos, wenn der Füllstand ausgelesen werden kann (Float Interrupt),
    - vom Drucker nach Druck einer empfangenen Druckzeile (Printer Interrupt),
    - vom Tastenfeld nach Betätigen einer Taste (Button Interrupt),
    - von der internen Uhr je nach Einstellung (Timer Interrupt).
  - Ein- und Ausschalten der Alarmglocke erfolgt über Setzen eines Bits.
- Echtzeitanforderungen
  - Zur Überlauferkennung muss das Auslesen des Füllstands einige Male pro Sekunde erfolgen.
  - Der Drucker druckt ca. zwei bis drei Zeilen pro Sekunde.
  - Das Auslesen der Füllstände der Tanks kann nur sequenziell erfolgen.
  - Der Mikroprozessor benötigt für die Berechnung des Tankinhalts auf Basis von Füllstand und Temperatur etwa 5 Sekunden!

Grundlegende Entwurfsentscheidungen

- Erkennung von Lecks und Überläufen
  - Für die langfristige Erkennung von Lecks wird die Historie der berechneten Tankinhalte verwendet.
  - Die kurzfristige Erkennung von Überläufen erfolgt auf Basis der aktuellen Füllstände mehrmals pro Sekunde.
- Verwendung eines Echtzeitbetriebssystems
  - Wenn kein RTOS verwendet wird, müssten sämtliche Operationen mit einer Reaktionszeit unter 5 Sekunden in Interrupts aufgerufen werden (z.B. Update des Displays, Formatierung und Druck von Reports).
  - Verwendung eines RTOS führt zu einem wesentlich klareren Entwurf:
    - Kurze Interrupt-Handler garantieren kurze Reaktionszeiten.
    - Anschlussaktivitäten werden prioritätsgesteuert in Tasks ausgeführt.
  - Erhältlichkeit eines geeigneten RTOS für den 8-bit-Prozessor muss gegeben sein.

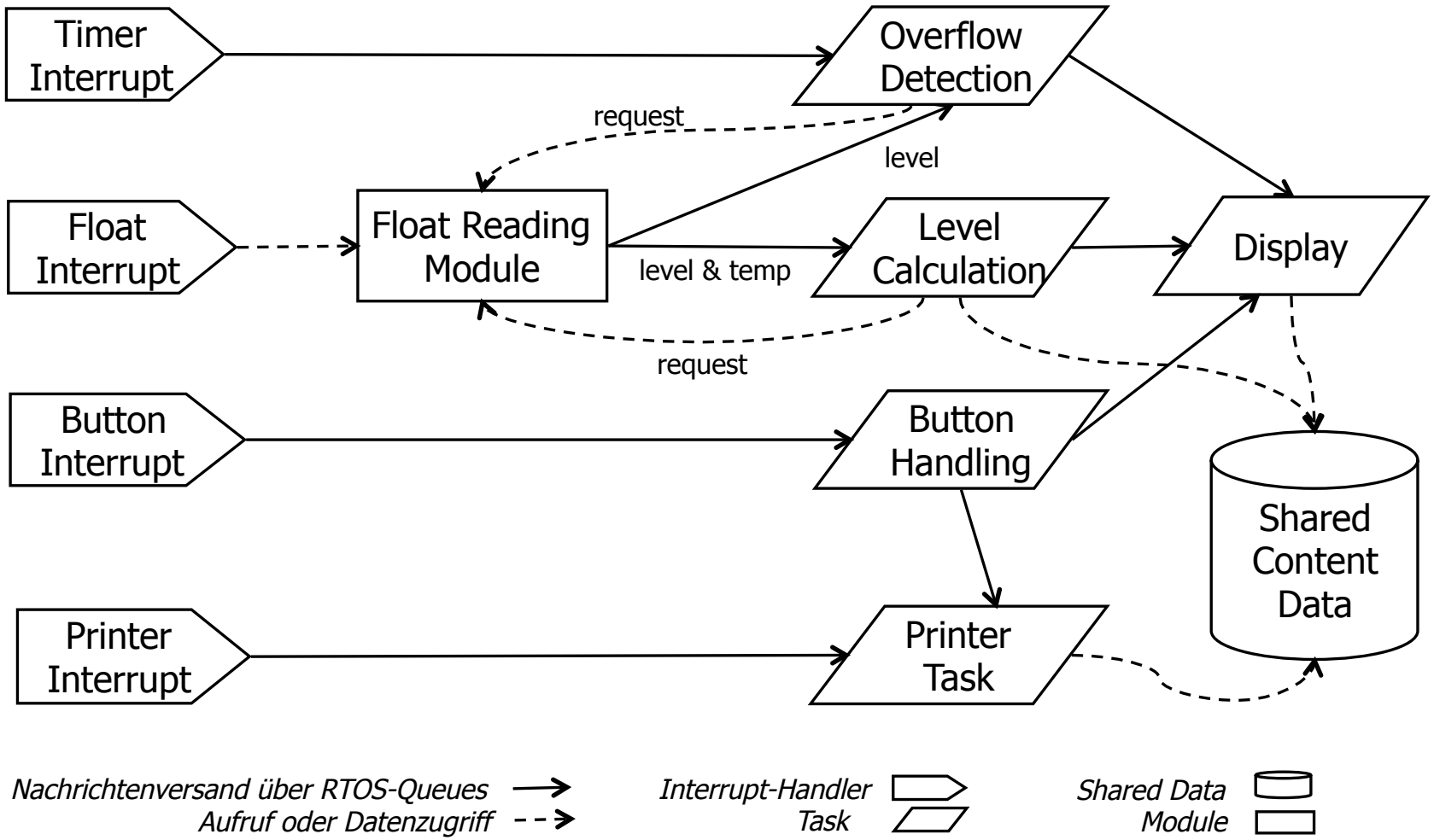
Entwurf der Tasks I

- **Level Calculation Task** zur Berechnung des Tankinhalts
  - Separater Hintergrund-Task für die langwierige Berechnung des Tankinhalts (Hauptgrund für Auswahl der RTOS-Architektur).
  - Ein Task für sämtliche Messgeräte, da das Auslesen der Füllstände über die Schwimmer-Hardware nur sequenziell möglich ist und keine besonderen Anforderungen an die Reaktionszeit bestehen.
- **Overflow Detection Task** zur Erkennung von Überläufen
  - Interrupt-gesteuerter Aufruf hoher Priorität jede 1/10 Sekunde.
  - Benötigt wie Level Calculation Task Zugriff auf die Mess-Hardware. Zugriff kann über Semaphor synchronisiert werden.
- **Button Handling Task** zur Behandlung der Tastatur
  - Entgegennahme von Tastendrücken im Interrupt-Handler.
  - Komplexe Berechnung des Dialogzustands für Menüsteuerung in hochpriorisiertem Task statt im Interrupt-Handler.

Entwurf der Tasks II

- **Display Task** zur Verwaltung der Anzeige
  - Koordiniert den Zugriff auf die Anzeige-Hardware und verwaltet den Dialogzustand.
- **Printer Task** zur Formatierung von Reports
  - Interrupt-Handler für Erkennung der Zeilenfortschaltung
  - Formatierung der Reports in Task mittlerer Priorität
  - Verwendung von RTOS-Queues als Drucker-Warteschlange
- Kein eigener Task zur Verwaltung der Alarmglocke.
  - Alarmglocke hat außer Ein/Aus keinerlei Zwischen-Zustände, für deren Verwaltung ein eigener Task sinnvoll wäre.
  - Ein- und Ausschalten kann deshalb jeweils von jedem Task aus erfolgen, der Zugriff auf die Alarmglocke braucht.
- Kein eigener Task zur Verwaltung der errechneten Tankinhalt-Daten.
  - Schneller und unkomplizierter Datenzugriff von Level Calculation Task, Display Task und Printer Task über Shared Memory.
  - Synchronisation über Semaphor führt nicht zu langen Wartezeiten.

Resultierende Task-Architektur



Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

Unterschiede zu geschlossenen lokalen Systemen

Die meisten Architekturprinzipien und Konzepte aus der geschlossenen lokalen Welt sind übertragbar. Zusätzlich sind zu berücksichtigen:

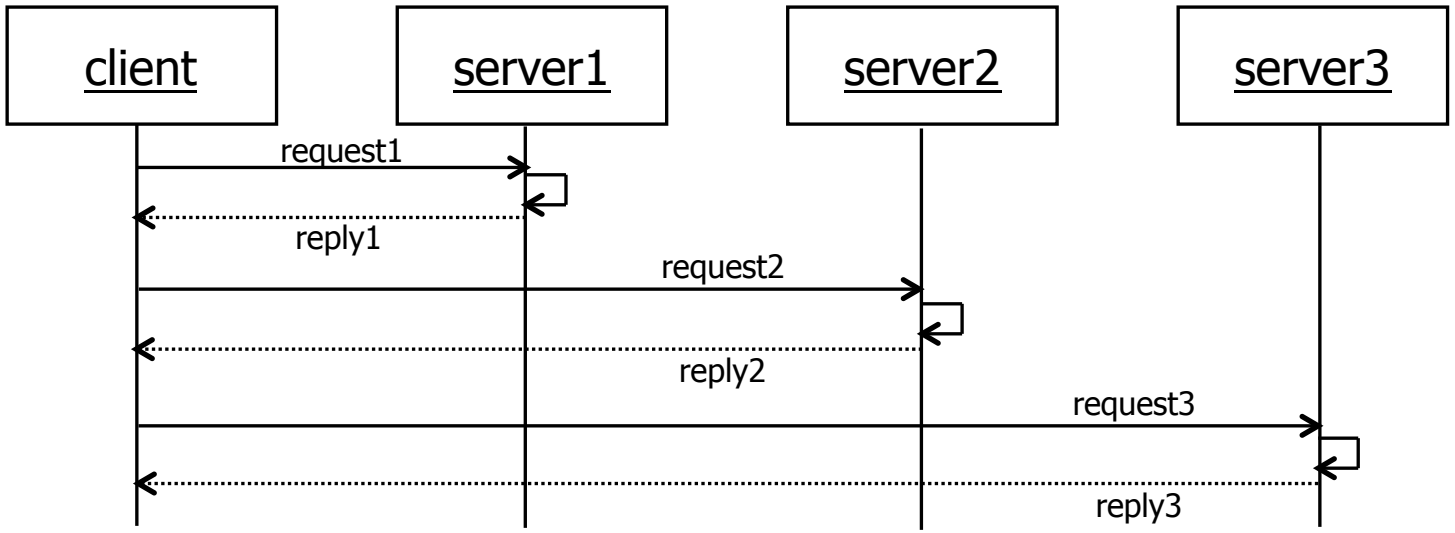
- Höhere Latenz und beschränkte Bandbreite bei der Kommunikation
  - Zu den Reaktionszeiten müssen jeweils die erforderlichen Übertragungszeiten hinzugerechnet werden.
  - Die Bandbreite der Verbindung kann Anzahl und Umfang der gesendeten Nachrichten einer bestimmten Priorität beschränken.
  - Hieraus können zusätzliche Einschränkungen für den Entwurf der Task-Architektur ergeben: Falls eine schnelle Reaktion erforderlich ist, müssen Tasks gegebenenfalls auf einem Mikrocontroller „vor Ort" ausgeführt werden.
- Ausfall von Komponenten oder Kommunikationsverbindungen
  - Ausfälle müssen erkannt und behandelt werden (vgl. die Patterns aus der vorigen Vorlesung).
  - Das System muss auch bei Ausfällen von Teilen sicher bleiben und sollte noch möglichst viel Funktionalität anbieten (graceful degradation).

Formen den Kommunikation

- **Synchrone Kommunikation**: Unter synchroner Kommunikation versteht man einen Modus der Kommunikation, bei dem die Kommunikationspartner (Prozesse) beim Senden oder beim Empfangen von Daten immer synchronisieren, also warten (blockiert), bis die Kommunikation abgeschlossen ist.
  - Wird sowohl beim Senden als auch beim Empfangen gewartet (der Sender stellt also eine Anfrage und wartet auf Antwort), so entspricht das einem Rendezvous der beiden beteiligten Prozesse.
  - Das Blockieren des Prozesses wird intern durch geeignete Mechanismen zur Prozesssynchronisation erreicht.
- **Asynchrone Kommunikation**: Modus der Kommunikation, bei dem das Senden und Empfangen von Daten zeitlich versetzt und ohne Blockieren des Prozesses durch bspw. Warten auf die Antwort des Empfängers - dies erfordert explizit oder implizit Warteschlangen.

Verteilte synchrone Kommunikation

- Die verteilte synchrone Kommunikation hat gewisse Ähnlichkeiten mit der ebenfalls synchronen Round-Robin-Architektur und teilt einige ihrer Vor- und Nachteile:
  - einfach zu realisieren und zu verstehen
  - lange Gesamt-Reaktionszeit (keine Parallelverarbeitung)
  - eine hängende Komponente bringt das gesamte System zum Stehen



Verteilte asynchrone Kommunikation

- Sowohl die erhöhte Latenz als auch der Ausfall von Komponenten lassen sich durch verteilte asynchrone Kommunikation teilweise in den Griff bekommen:
  - Asynchrone Kommunikation erlaubt es dem Versender, weiter zu arbeiten, während eine Nachricht verschickt wurde. Damit lassen sich Wartezeiten und zusätzliche Taskwechsel vermeiden.
  - Bei einem temporären Ausfall einer Empfänger-Komponente kann weitergearbeitet werden. Bei Zwischenspeicherung der Nachrichten in Queues kann er sogar transparent überbrückt werden.
- Grundsätzlich lassen sich zwei Kommunikationsarten unterscheiden:
  - Der Sender kennt den Empfänger und kann direkt mit ihm kommunizieren (Beispiel: CORBA Messaging).
  - Die Kommunikation erfolgt indirekt über zwischengeschaltete Nachrichtenkanäle, bei denen sich die Partner als Sender oder Empfänger registrieren (Beispiele: CORBA Event Service, CORBA Notification Service, Java Messaging Service).

CORBA Messaging: One-Ways

- Wird standardmäßig von CORBA durch das Schlüsselwort oneway unterstützt, allerdings ohne jede Auslieferungsgarantie: „best try".
- Asynchronous Messaging unterstützt über Policies feinere Abstufungen und macht damit das oneway-Schlüsselwort obsolet:
  - SYNC_NONE: kein Blockieren, keine Auslieferungsgarantie (wie oneway)
  - SYNC_WITH_TRANSPORT: Blockieren, bis an Transportschicht übergeben
  - SYNC_WITH_SERVER: Blockieren, bis an Server-ORB übergeben
  - SYNC_WITH_TARGET: Blockieren, bis vollständig ausgeführt (normaler Aufruf)

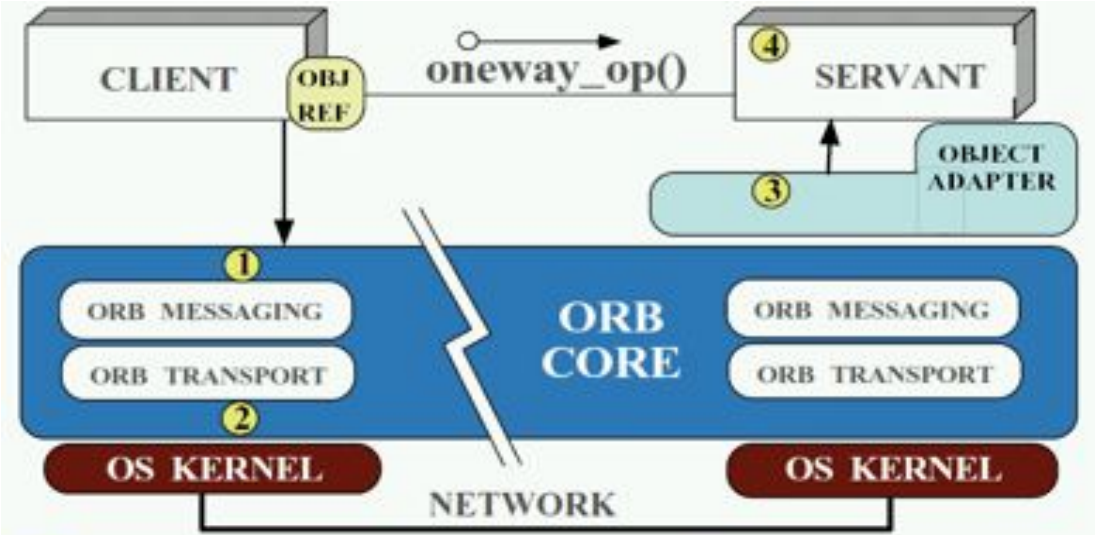
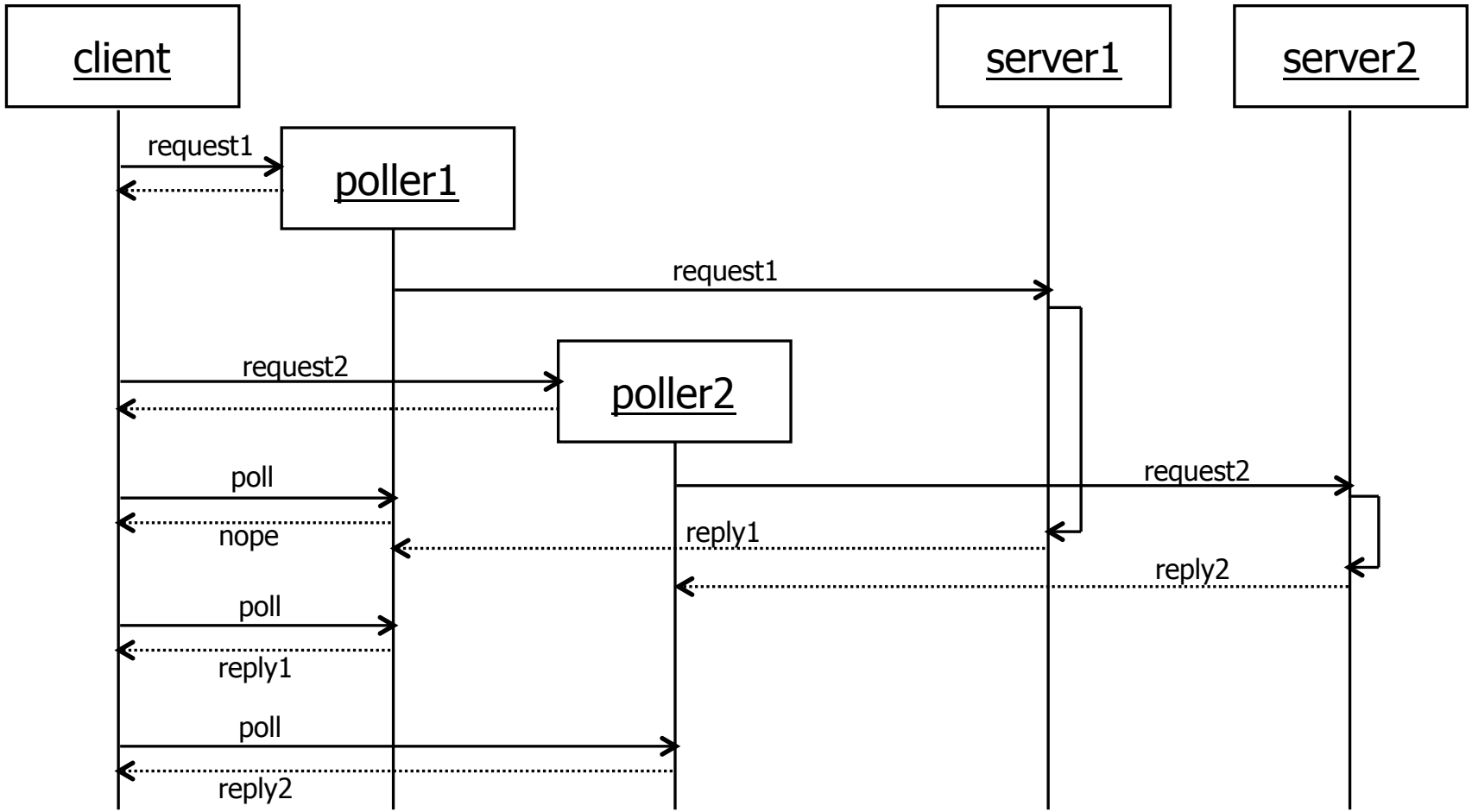


Bild aus [SV03]

CORBA Messaging: Asynchronous Method Invocation

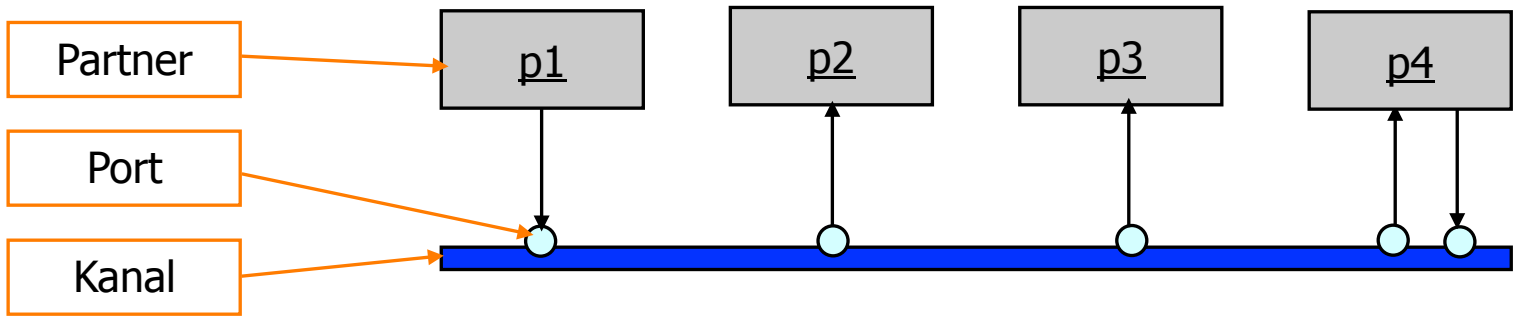
- Neben dem Versenden von One-Way-Nachrichten gibt es auch die Möglichkeit, Methodenaufrufe mit Rückgabewerten asynchron aufzurufen (AMI).
- CORBA Messaging bietet hier zwei Möglichkeiten:
  - Rückgabewert ist ein lokales `Poller`-Objekt, das dann vom Client aktiv abgefragt werden kann, bis die Antwort angekommen ist. Der Client kann auch entscheiden, am `Poller` zu blockieren.
  - Der Client übergibt eine Callback-Methode, die vom ORB aufgerufen, wenn die Antwort angekommen ist. Dazu muss der Client die Kontrolle an den ORB abgeben.
- Durch die Nutzung dieser Möglichkeiten kann der Entwickler des Clients die Parallelität erhöhen. Dazu muss er seine Anwendung nicht in unterschiedliche Threads aufteilen, nur um jeweils auf die Rückkehr der blockierenden synchronen Aufrufe zu warten.

Beispiel für Nutzung von AMI mit Polling



CORBA Event Service

- Entkopplung der Kommunikationspartner – diese müssen sich nicht gegenseitig kennen, sondern registrieren sich an einem gemeinsamen Kommunikationskanal
- m:n-Kommunikation – wenn mehrere Empfänger an dem Kanal registriert sind, wird die gesendete Nachricht an alle verschickt
- Kommunikationspartner können gleichzeitig Sender und Empfänger sein.
- Nachrichten können aktiv ausgelesen werden (Pull) oder über Callbacks zugestellt werden (Push). Weiterhin lassen sich Filter angeben.
- Die Dienstgüte lässt sich über Policies regeln (auch zur Spezifikation von Echtzeitanforderungen).



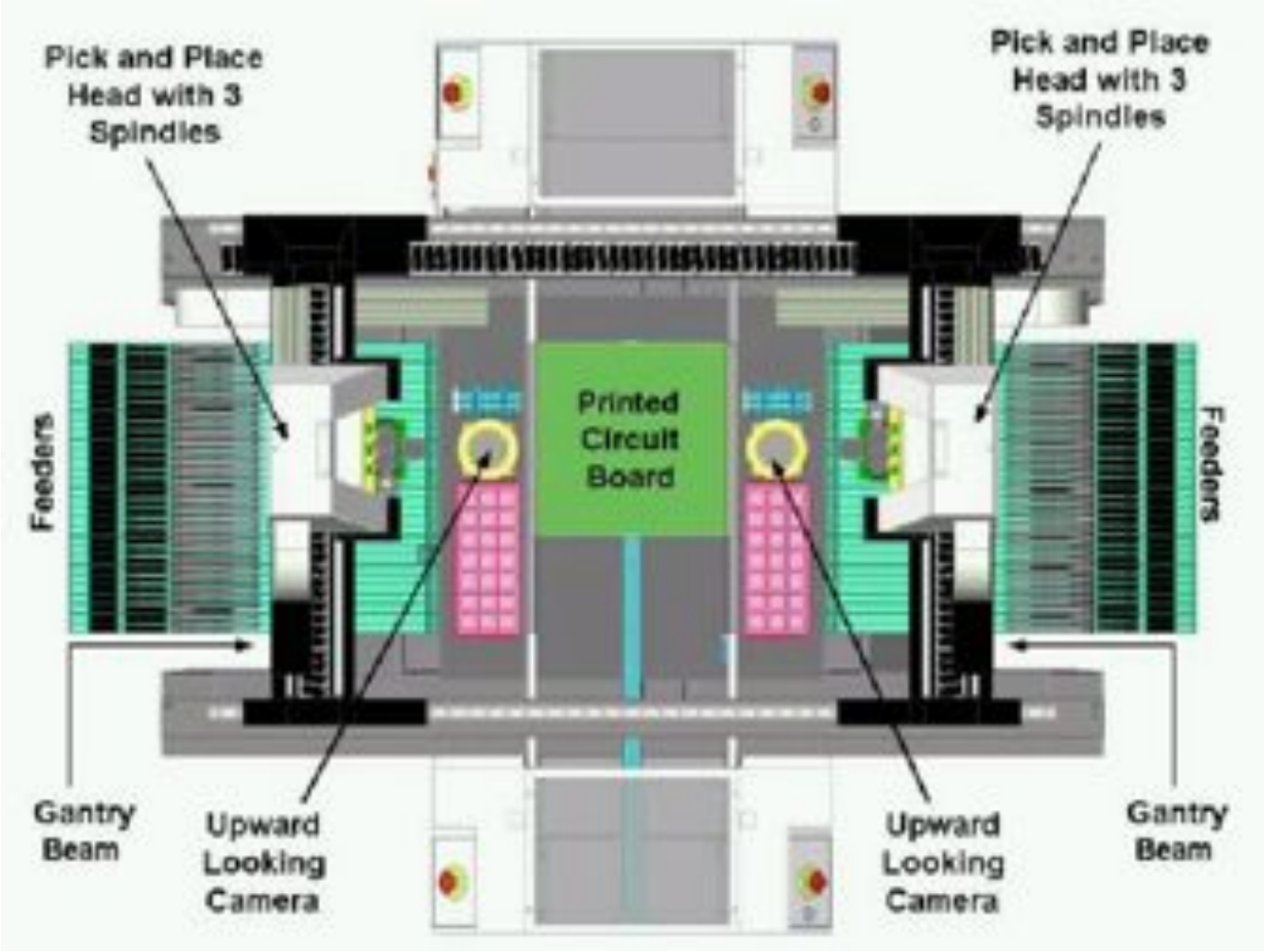
Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

SMT Assembly Equipment

Beispiel aus
[JS01,BT00]

SMT Assembly Equipment



Anforderungen: Platzierung von SMT-Bauteilen

- Platzierung von Bauteilen in Surface-Mount-Technology auf Leiterplatten mit hoher Geschwindigkeit und Genauigkeit.
- Detailaufgaben
  - Steuerung der Bauteil-Versorgung (Feeder)
  - Kontrolle zweier Roboterarme, die abwechselnd 3-5 Bauteile aus den Feedern holen und sie dann auf der Leiterplatte platzieren
  - Bilderfassung zur Lagekorrektur der aktuell von den Roboterarmen gehaltenen Bauteile
  - Zeit für Armbewegung von Feeder zu Platine: 200ms.
  - Zeit für Armbewegung von Kamera zu Platine: 50ms.
  - Steuerung über GUI auf zugehörigem Rechner
- Hardware
  - 5 Pentium-Prozessoren, 7 Spezialprozessoren, 30 Mikrocontroller
  - Vollständige Vernetzung sämtlicher Prozessoren (über Feldbus?)

Software-Architektur

- Basiert auf CORBA-ORB TAO und nutzt die folgenden CORBA-Kommunikationsmechanismen:
  - Two-Way Synchronous Calls
  - Asynchronous Method Invocation
  - Real-Time Event Service
- Das Kommunikationsnetz ist durch ein proprietäres Pluggable Protocol Framework unter dem ORB gekapselt.
- Zur Verwaltung von Threads werden die standardisierten Schnittstellen und Mechanismen des ORBs genutzt. Damit müssen Anwendungsprogrammier keine proprietären Schnittstellen des zugrundeliegenden RTOS nutzen.
- Zur Steuerung, Überwachung und Wartung der Maschine über eine GUI wird über einen integrierten Mini-Web-Server ein Java-Applet versandt, das dann über CORBA mit dem System kommuniziert.

Einsatz der CORBA-Kommunikationsmechanismen

- Two-Way Synchronous Calls
  - Hauptvorteil: Einfaches Programmiermodell
  - Standard-Kommunikationsmittel für Anwendungsfälle, bei denen keine hohen Performance-Anforderungen bestehen
  - Vorhersagbarkeit und damit Echtzeitfähigkeit durch ORB und verwendetes Transportprotokoll gegeben
- CORBA Messaging (Asynchronous Method Invocation)
  - Vereinfachung des Programmiermodells sowie Verringerung des Aufwands für Task-Wechsel im Vergleich zu der expliziten Verwaltung mehrerer Threads (Nutzung des Callback-Modells)
  - Hohe Performance-Anforderungen erfordern Parallelität
  - Fire-and-Forget-Semantik z.B. bei Bilderfassung und Feeder; Rückmeldungen von Fehlern etc. können später erfolgen
  - Spezifikation von TimeOut-Policies erlaubt kurzzeitiges Abkoppeln von Teilen der Maschine, ohne dass der Anwendungscode umgeschrieben werden müsste

Einsatz der Kommunikationsmechanismen

- Real-Time Event Service
  - Einfaches Programmiermodell: Nachrichten können verschickt werden, ohne dass sich die Nachrichten-Quelle darum kümmern müsste, wer die Nachricht erhalten wird
  - Gute Erweiterbarkeit durch einfaches Hinzufügen zusätzlicher Nachrichten-Empfänger
  - Filterung von Nachrichten durch den Event Service vermindert die Last bei den Nachrichtenempfängern und auf dem Netz

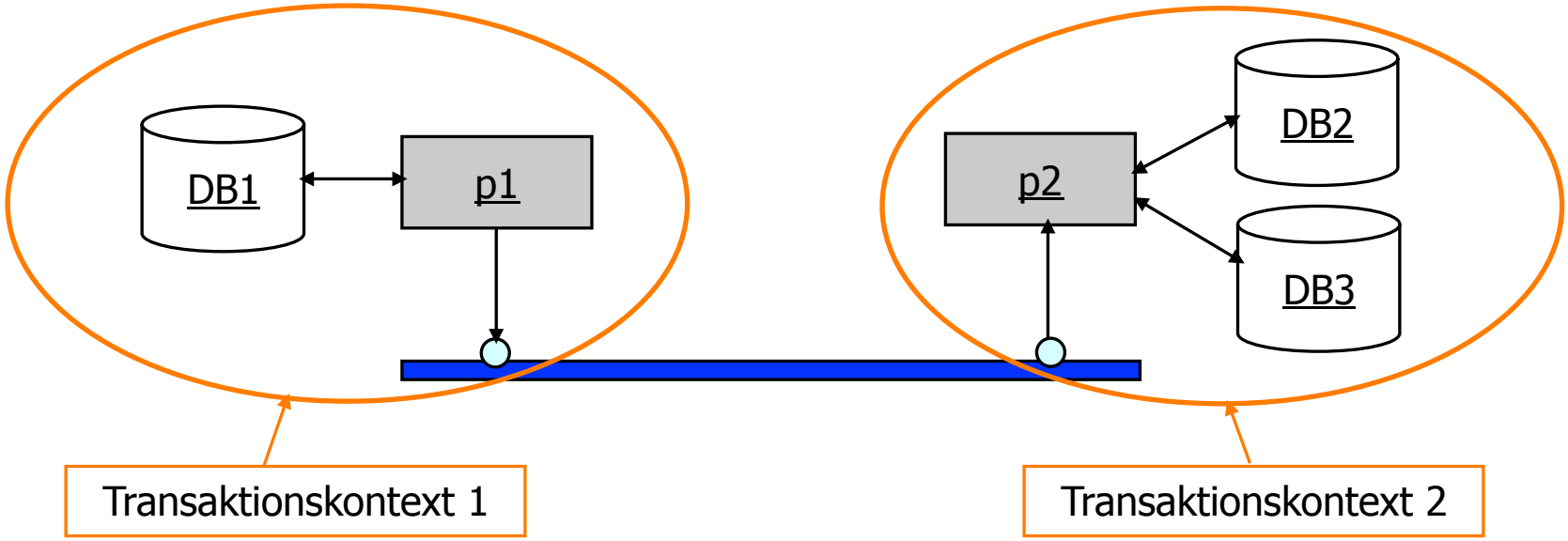
Das Beispiel zeigt, dass es nach dem Stand der Technik bereits möglich ist, sehr anspruchsvolle verteilte geschlossene Systeme weitgehend auf Basis standardisierter Basistechniken zu bauen.

Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

Asynchronous Transactional Messaging (ATM)

- Die asynchrone Übertragung von Nachrichten über Event Services wird auch in Informationssystemen oft eingesetzt.
- Dabei sind das Ablegen von Nachrichten in einen Kanal und das Herausnehmen von Nachrichten aus einem Kanal durch lokale Transaktionen geschützt.
- Wenn der Kanal die Daten sicher speichert und weiterleitet, kann man damit in vielen Anwendungen auf globale Transaktionen verzichten.



ATM versus Globale Transaktionen

ATM

+ Entkopplung der Partner bei der Kommunikation
+ Flexible Konfiguration von Routing, Filtering etc.
+ Performanz und Skalierbarkeit
+ Robustheit bei temporären Ausfällen von Ressourcen
+ Einsatz bewährter Standard-Komponenten (z.B. MQSeries, Tibco, JMS-Server)
– Verteilte Realisierung von Geschäftsvorfällen
– Kein global einheitlicher Datenzustand

Globale Transaktionen

+ Einfache Realisierung von Geschäftsvorfällen
+ Global einheitlicher Datenzustand
– Kommunikationsaufwand durch Two-Phase-Commit-Protokoll
– Macht globale Sperren auf Objekten erforderlich
– Mangelnde Skalierbarkeit
– Sämtliche Ressourcen müssen für Erfolg gleichzeitig verfügbar sein
– Nicht von allen Ressourcen unterstützt

Fazit: Globale Transaktionen sind meist nur sehr lokal einsetzbar.

Offene verteilte Systeme - Ausblick

- Neue Klasse von Systemen
  - Wechselnde Geräte kommunizieren über dynamisch aufgebaute Verbindungen (beispielsweise über Internet oder über Funk).
  - Neue Anwendungen lassen sich zur Laufzeit installieren.
- Neue Aspekte
  - Erkennung und Einbindung von Partnern und Diensten
  - Sichere Identifikation von Partnern, beispielsweise über Public-Key-Infrastrukturen (PKI)
  - Behandlung von temporären Ausfällen durch asynchrone Kommunikation mit Zwischenspeicherung von Nachrichten
  - Zeitlich begrenzte Reservierung von Ressourcen (Leases mit Timeout)
  - Konfiguration und Update von Anwendungen zur Laufzeit
- Beispiele
  - Mobilfunknetze als Beispiel für hochgradig dynamische Systeme
  - JINI als Vorschlag für eine Software-Infrastruktur zur Realisierung von Diensten in verteilten offenen Systeme

Inhalt

- Rekapitulation
- Architekturen für geschlossene lokale Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: Underground Tank Monitoring System
- Architekturen für geschlossene verteilte Systeme
  - Architekturkonzepte und Entwurfstechniken
  - Beispiel: SMT Assembly Equipment
- Abstecher und Ausblicke
  - Asynchronous Transactional Messaging
  - Offene verteilte Systeme
- Zusammenfassung

Zusammenfassung

- Für geschlossene lokale Echtzeit-Systeme gibt es einige Standard-Architekturen, die sich in ihrer Komplexität und in den erzielbaren Reaktionszeiten unterscheiden.
- Je mehr Asynchronität eine Architektur bietet, desto besser sind die erzielbaren Reaktionszeiten und die erzielbare Parallelität.
- Die Hauptaufgaben bei der Abbildung der fachlichen Architektur auf die technische Architektur sind der Entwurf der Tasks und die Auswahl der Kommunikationsmechanismen.
- Nach dem Stand der Technik ist es möglich, sehr anspruchsvolle eingebettete Echtzeit-Systeme weitgehend auf Basis standardisierter Basistechniken zu bauen.
- Asynchrone Architekturen sind nicht nur für eingebettete Systeme wichtig, sondern werden sehr häufig auch im Bereich der betrieblichen Informationssysteme eingesetzt.

Literaturhinweise

[PM07] Peter Marwedel: Eingebettete Systeme - Eine Einführung. Springer, Berlin 2007

[BP03] Bruce Powell: Real-Time Design Patterns, Real-Time Design Patterns, Addison-Wesley 2003.

[BP99] Bruce Powell: Real-Time UML 2nd Edition – Developing Efficient Objects for Embedded Systems, Addison-Wesley, 1999.

[BT00] Bruce Trask: Using CORBA Messaging, Real-Time Event Service and ORB Concurrency Models in a Real-Time Embedded System, Präsentationsfolien zum OMG-Workshop „RT and Embedded Systems 2000", Contact Systems, 2000.

[JS01] Jon Siegel: CORBA for Real-Time Systems, http://www. biztool.com/magazines/ java/archives/0510/Siegel/index.html, SYS-CON Media, 2001.

[Si99] David E. Simon: An Embedded Software Primer, Addison-Wesley, 1999.

[SV03] Doug Schmidt, Steve Vinoski: CUJ and C++ Report Columns on Distributed Object Computing, http://www.cs.wustl.edu/ ~schmidt/report-doc.html, 2003.

[DT10] David Trachtenherz: Eigenschaftsorientierte Beschreibung der logischen Architektur eingebetteter Systeme. Vieweg+Teubner 2010