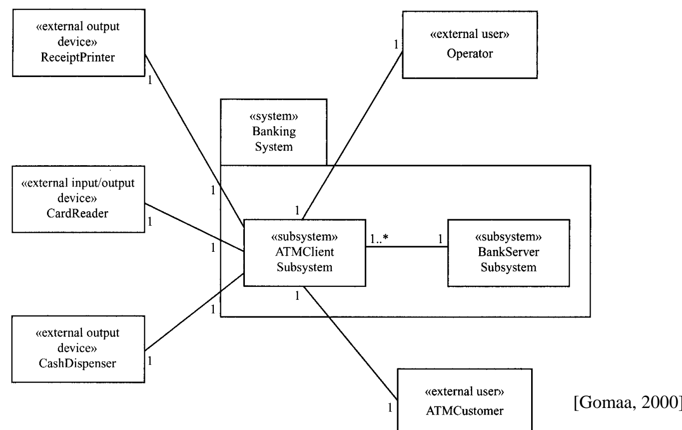
REAL-TIME DISTRIBUTED SYSTEMS DETAILED DESIGN METHODOLOGY (2)

Software Architecture Design

The approach taken to decompose the system into subsystems, components, interfaces and their interconnections is referred to as developing the **software architecture**. Specific guidelines are required for concurrent, real-time and distributed applications.

A number of architectural styles or patterns can be defined for this area:

1. Client/Server - simple service provider/multiple clients, e.g. ATM banking system:



[Gomaa, 2000]

2. Layered Abstraction - simple interfaces at lower levels rising to complex functionality, e.g. ISO network protocol stack.

3. Communicating Tasks - a network of concurrent tasks with separate threads of control optionally sharing data.

System Decomposition

The general approach to system decomposition is guided overall by the principle of information hiding and separation of concerns:

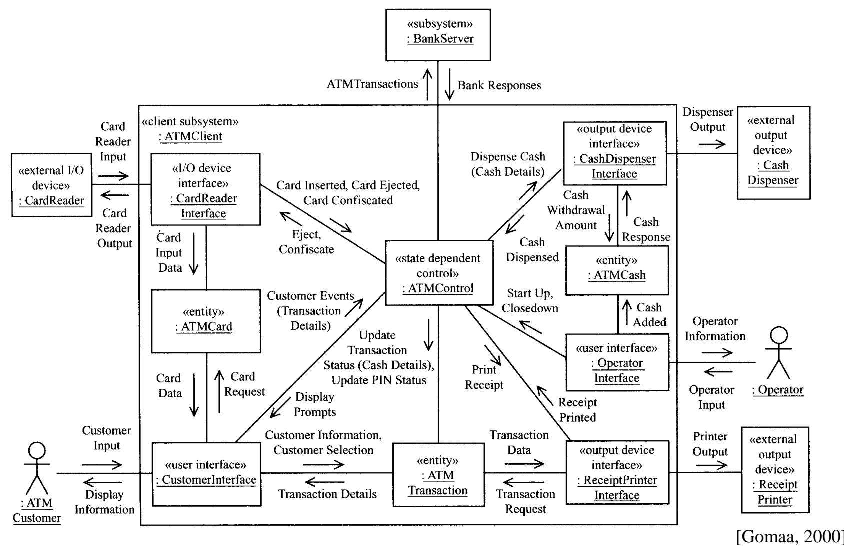
1. Subsystems need to be as independent of each other as possible - low coupling
2. Linkages between objects in the subsystems should be high - high cohesion
3. Once subsystems and their interfaces have been defined, detailed subsystem design can proceed independently.
4. Use case object interaction models form the basis of subsystem design
5. Objects appearing in multiple use cases must be allocated to one subsystem (the one with the strongest coupling to the object).

Subsystem Software Architecture

To determine the subsystems requires a transition from analysis to initial design via integrating parts of the analysis model - all the collaboration diagrams developed for each use case are combined into a **consolidated collaboration diagram**. This phase was referred to in earlier OO methodologies as "robustness analysis" but COMET emphasises the dynamic analysis model through the message communication interfaces.

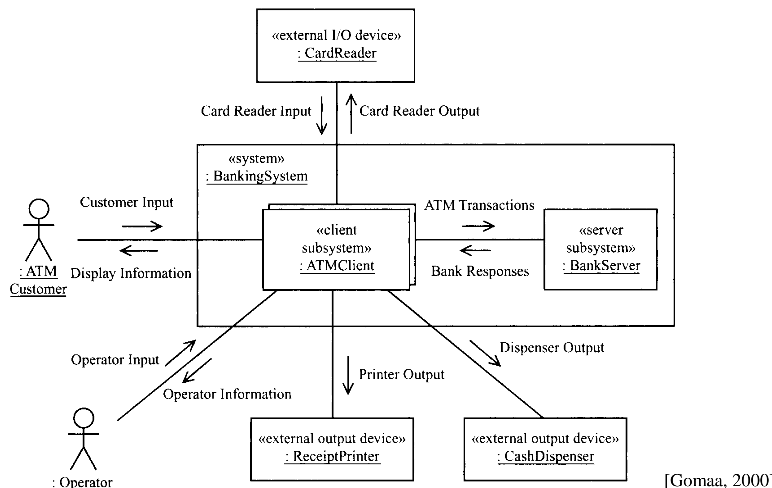
The consolidated collaboration diagram shows all message communications: both the main and all alternative use-case sequences. If the diagram becomes too complex then aggregate messages can be substituted, e.g. *Cruise Control Messages* (with an associated directory) can be used for the *Cruise Control* object.

Given a consolidated collaboration diagram, subsystems can be identified and from these subsystem collaboration diagrams constructed, e.g. for the ATM Banking system:



[Gomaa, 2000]

A higher level system collaboration diagram can then be constructed which hides the subsystems objects and interactions and only shows subsystems and subsystem interactions, e.g:



[Gomaa, 2000]

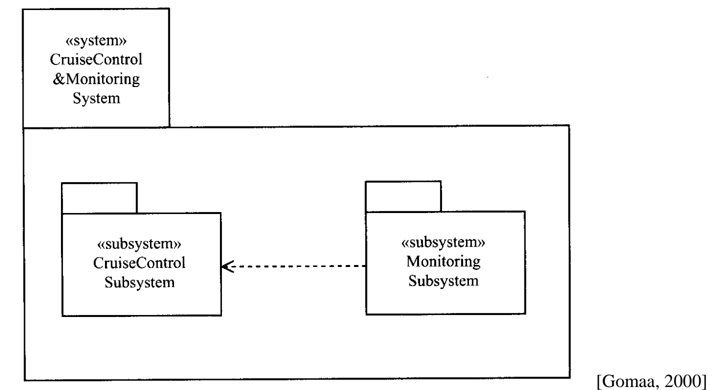
Subsystem Design Guidelines

The principles of information and separation of concerns also underpin subsystem design guidelines:

1. Aggregate/Composite objects should be in the same subsystem
2. Geographically separated objects should be in different subsystems
3. Clients and Servers should be in different subsystems
4. User interfaces are usually separate subsystems
5. External objects should interface to only one subsystem
6. A control object and its entity and interface objects it controls should be in the same subsystem.
7. Entity objects should be in the same subsystem as objects that update it.

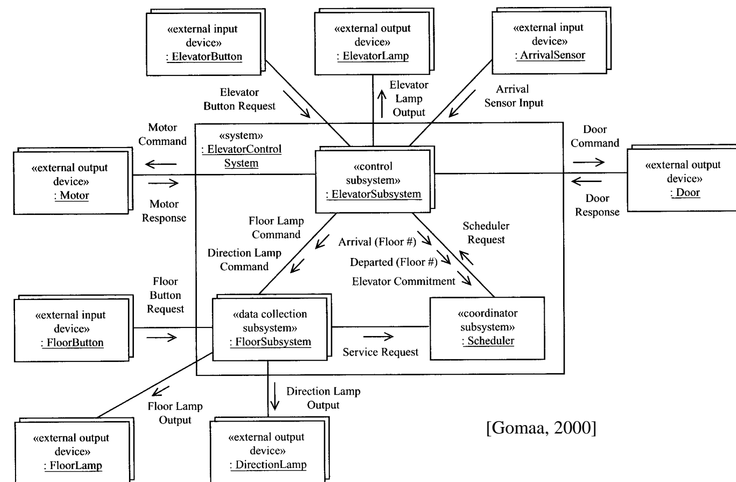
Common Subsystem Types

1. Control - receives inputs, generates outputs and is usually state-dependent, e.g:



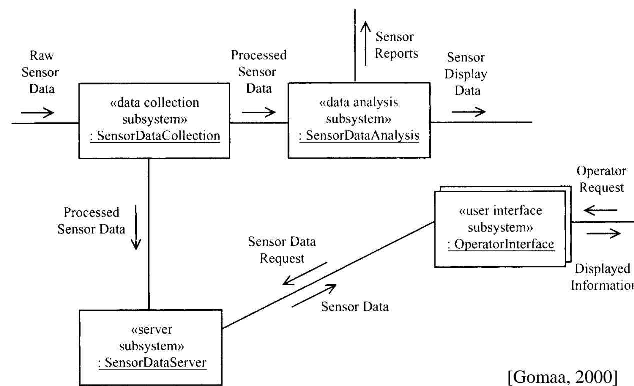
[Gomaa, 2000]

2. Coordinator - usually where there are multiple control subsystems a coordinator is required, e.g:



[Gomaa, 2000]

3. Data Collection - a subsystem that collects raw data from the environment and preprocesses it for use by other subsystems, e.g:

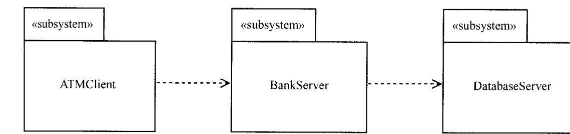


[Gomaa, 2000]

4. Data analysis - analyzes data, provides reports
5. Server - provides a service to other subsystems
6. User Interface - provides a display of system information and gathers user input
7. System services - not application specific, but generic to the platform or operating system

Example: Bank ATM system

A typical three-tier client/server application:



[Gomaa, 2000]

Subsystem Decomposition - static modelling re-examination

The conceptual static model (with class diagrams obtained from requirements analysis) developed earlier in the methodology can be revisited with the consolidated collaboration diagram to review instantiations of objects from classes and relationships between classes. Association navigabilities can also be assigned or reviewed.

Distributed Architecture Design

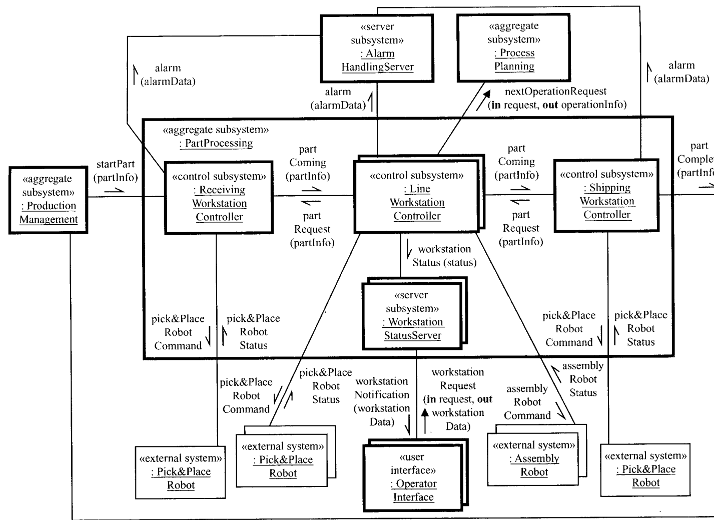
In COMET, the approach is to provide a concurrent message-based design that is highly configurable → the architecture is highly portable across different system configurations and platforms.

A component-based development approach assists this aim by designing each subsystem as a component (i.e. an active object with a well-defined interface). It should be designed to be self-contained and be reusable.

Designing Distributed Subsystems

1. Decompose into subsystems that can potentially execute on separate nodes (use subsystem structuring criteria - define components and interfaces)
2. Decompose subsystems into concurrent tasks and information hiding objects
3. Map instances of the design onto a distributed hardware configuration

To design distributed subsystems, the consolidated collaboration diagrams showing all systems objects and their interactions is the most useful for decomposition. Objects are most generally geographically grouped into **composite subsystems**, or logically/functionally grouped into **aggregate subsystems** (which may span geographical locations), e.g:



[Gomaa, 2000]

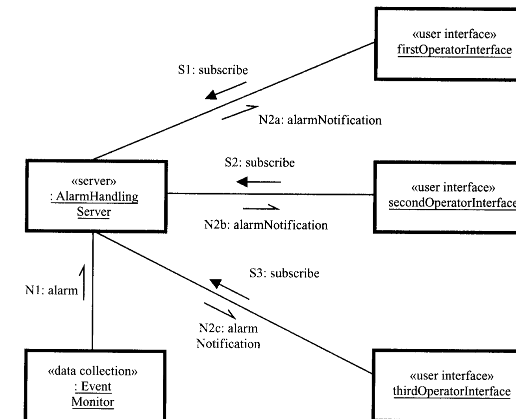
Although the subsystem structuring criteria as defined earlier are used to guide decomposition, additional criteria are also needed to specifically guide decomposition into distributed subsystems:

1. Proximity to the physical data source - fast access to local data
2. Localized autonomy - receiving general direction, but provides lower level control and monitoring
3. Performance - time-critical functions are confined to single nodes
4. Specialized hardware - interface has to be local, or provides dedicated processing capability
5. User Interface & Servers (as mentioned in previous criteria)

Designing Subsystem Interfaces

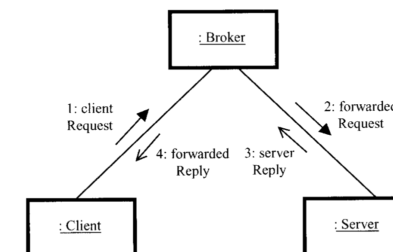
Essentially, two types can be used:
1. Loosely-coupled (asynchronous) messages - via FIFO queues or priority message queues.
2. Tightly coupled (synchronous) messages - common in client/server architectures (e.g. RPC or RMI)

Which are seen in various distributed communication patterns:
1. Subscription/Notification and Group messaging communications - one-to-many (broadcast or multicast), e.g:



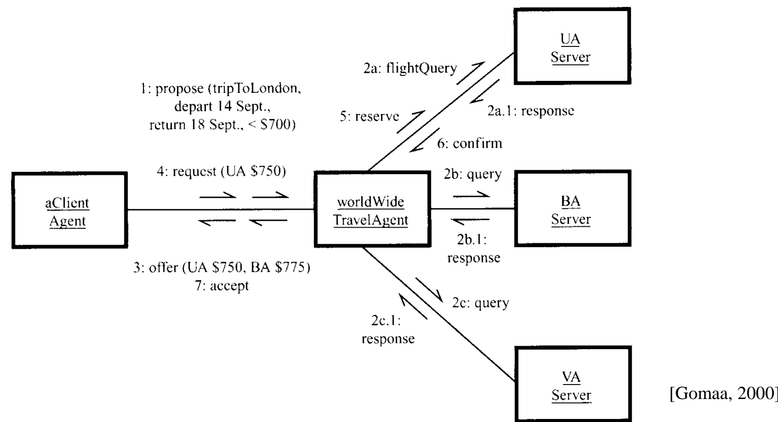
[Gomaa, 2000]

2. Object Broker communications - client/server intermediary providing location transparency, e.g:



[Gomaa, 2000]

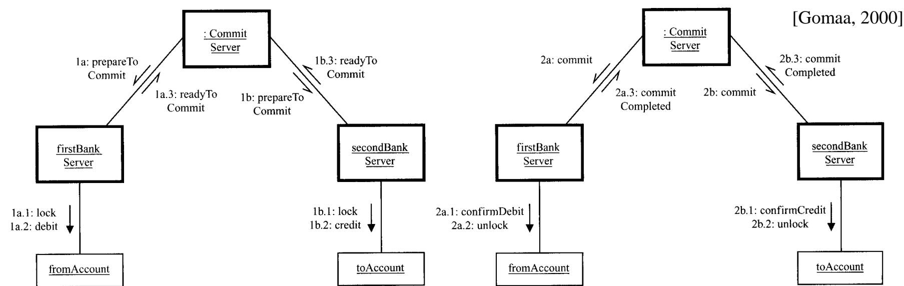
3. Negotiated communications - common in multi-agent systems (client agents act on behalf of a user to negotiate with servers), e.g:



[Gomaa, 2000]

4. Transaction based communications - usually indivisible operations on a server (i.e. all operations are performed or none), e.g: **two-phase commit protocol**:

[Gomaa, 2000]

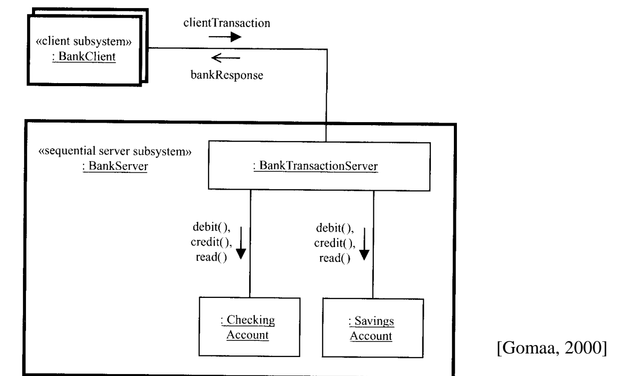


Note where there may be some delay between the phases, it is usual to prefix the protocol with a *query* only transaction followed by a *reservation* transaction (with the two-phase commit).

Server Subsystem Design & Data Distribution

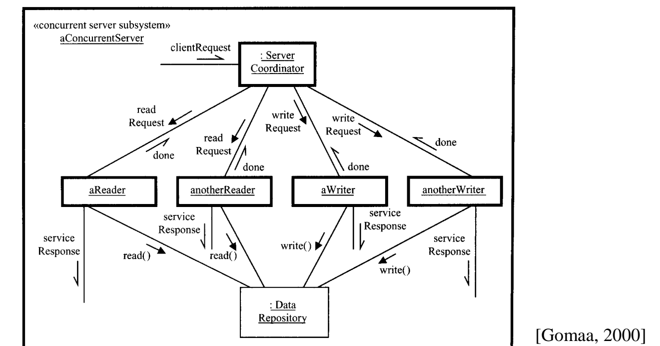
The purpose of servers is usually to encapsulate some data abstraction object(s) and provide a means of access to clients. The server can be either be sequential or concurrent:

1. Sequential Server: simple client request response after invoking the appropriate operations on the data abstractions, e.g:



[Gomaa, 2000]

2. Concurrent Server: where server demand is likely to be high, access must be made concurrent with appropriate synchronisation between reader and writer object via a coordinator, e.g:



[Gomaa, 2000]

Note that the service response is asynchronous and usually implemented as a **callback** from the server to the client when the service request has been met.

Both sequential and concurrent server subsystems encapsulate centralized data abstractions, but higher performance systems usually require some form of data distribution as well.

In a distributed server, data collected locally is held locally and served locally. For performance reasons, this data may need to be replicated in more than one server. Data must then be updated at regular intervals to ensure it is sufficiently up-to-date.

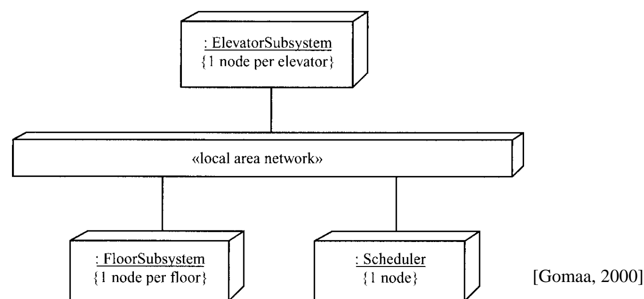
System Configuration

The final step in the COMET methodology is to map an instance of the distributed application to a physical architecture (the target system):

1. Define component instances - components may have multiple instances in the application and require unique IDs
2. Interconnect component instances - the architecture defines component communications but component IDs may need to be exchanged.
3. Map component instances (logical nodes) to physical nodes.

Example: Distributed Elevator Control System

On the deployment diagram, each component instance is allocated to a node:



[Gomaa, 2000]

Testing Issues in Real-Time (and Distributed) Systems

Real-Time distributed software is critically dependent on the timing and sequencing of inputs, and as these typically come from the physical world there are many possible values and combinations of inputs possible → a very large number of test cases.

Real-Time distributed software (particularly when some parts are embedded) is also usually more complex than conventional software due to the requirement to operate without human intervention → must incorporate a larger number of exception handling routines to support recovery from unexpected events → complexity also results in a large number of test cases.

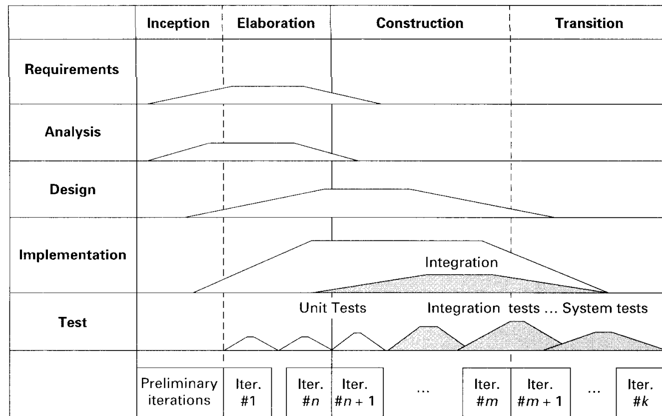
The operation of real-time distributed software across various processing nodes concurrently → interactions between the processors may be highly non-deterministic → also produces a very large number of test cases from all the possible inter-leaving and sequencing of messages between processors.

Additional variations in the performance of the underlying communications network also impacts the timing and sequencing of messages between processors.

Apart from analysis based testing methods, either from performance analysis of the design, or formal correctness proofs, systematic testing of a manageable set of statistically selected test cases is usually required so that probabilistic measures of software failure can be inferred from limited testing.

The most widespread testing technique is to construct a physical environment simulator to generate as many realistic possible variations of typical inputs as possible to test the real-time distributed system. The simulator may also be able to produce conditions which are not easily created in the physical world but represent scenarios that the system must be able to handle.

In terms of management of the testing process, the breakdown into phases of unit testing and integration testing is still applicable, e.g. from the Unified Process:



[Rational, 2000]

Unit testing is largely able to following conventional software engineering approaches (i.e. driven by generating test cases from use case requirements scenarios since units are self-contained and shouldn't require operation across distributed nodes or exhibit complex temporal interactions with the physical world (otherwise unit decomposition has not been very good!).

Integration testing requires that tests include:

- Temporal variations between software units executing concurrently on single processor nodes (which involves the scheduler and inter-task communications).
- Temporal variations between message delays for software units messaging each other on different distributed nodes.

Integration testing is largely design architecture driven *through selection and perturbation of critical event sequences* since testing must also confirm that all units conform to the architectural design assumptions. Integration testing must also cover *robustness* of the system as the integration load is increased on the hardware architecture.

System testing largely tests against the requirements and takes a more external view that overall performance requirements are met.