

System Design Document

TRAMP Project

February 07, 2002 — 10:32

Contents

1 Overview	4
1.1 Introduction	4
2 Proposed System	6
2.1 Overview	6
2.2 Subsystems and Services	6
2.2.1 Subsystem Decomposition	6
2.2.2 Subsystem Services	8
2.2.3 Interaction of Subsystem Services	34
2.3 Hardware/Software Mapping	34
2.3.1 General System Performance	35
2.3.2 Input/Output Performance	36
2.3.3 Processor and Memory Allocation	36
2.3.4 Connectivity and Network Infrastructure	36
2.4 Persistent Data Management	37
2.5 Access control and security	37
2.6 Global software control	37
2.6.1 External control flow (between subsystems)	38
2.6.2 Concurrent control	38
2.6.3 Internal control (within a single process)	38
2.7 Boundary conditions	39
A Glossary	51
Bibliography	53

List of Figures

2.1	Dependencies between the subsystems	7
2.2	Needs and Abilities for inter-subsystem communication	9
2.3	Needs & Abilities of <i>GarageServerApp</i> , <i>WearableApp</i>	11
2.4	Needs And Abilities of tracking services and related stuff	13
2.5	Tracking subsystem class diagram	14
2.6	needs and abilties of the session subsystem	40
2.7	User Interface subsystem diagram	41
2.8	UI subsystem diagram: break-down to adapter-viewer level	42
2.9	Calibrate System	43
2.10	Handover Maintenance Task	43
2.11	Navigation	44
2.12	Request Maintenace	44
2.13	Detect Marker	45
2.14	Taskflow Transition	45
2.15	Start Garage Server Application	46
2.16	Shutdown Garage Server Application	46
2.17	Start Latop Mechanic Application	46
2.18	Shutdown Latop Mechanic Application	47
2.19	Allocation of TRAMP subsystems to hardware	47
2.20	Interfaces and Operating Systems used by TRAMP hardware	47
2.21	Start Garage-Server Use Case	48
2.22	Start Mechanic-Service Use Case	49
2.23	Do Ping Use Case	50

1 Overview

This System Design Document (SDD) presents the technical details of the TRAMP system design. More information about the specific features and the motivation for TRAMP can be found in the Requirements Analysis Document (RAD) and the Problem Statement.

This document starts with an introduction to the architecture and the design goals to be considered. Then it presents the proposed system architecture by describing the subsystem decomposition and the subsystem services. The hardware/software mapping is defined and the management of persistent data is explained. Access control and security issues are addressed. The global software control and boundary controls are described. Finally a glossary of important terms is provided.

1.1 Introduction

TRAMP is a wearable mobile system for supporting the repair and maintenance of cars and other complex systems. It is designed as a collection of components that interact flexibly with each other. Based on DWARF, a framework for the development of general Augmented Reality (AR) systems, the components of TRAMP are designed as DWARF services having needs and abilities.

TRAMP was designed considering the following design goals:

- **Multiple Users**
The system should support tasks that are performed by multiple users in concert, supplying each with the necessary information at the appropriate time.
- **Wearability**
The users should have access to the TRAMP system where they need it. For this reason the wearable system should be designed so as not to encumber the user.
- **Usability**
Since the end-user will be using the system while performing work, it is essential for the system to be intuitive and easy to use. For example the user should be able to interact with the wearable system using different modalities.
- **Reconfigurability**
The user should be able to reconfigure his wearable system on the fly to accommodate different operating environments.

1 Overview

- Scalability
The system must be scalable in terms that it can support many users communicating or retrieving information at the same time.
- Extensibility
Since the purpose of the project is to create a functional prototype rather than a complete system, it is essential for the application to be easily extensible. This is accomplished by use of various design abstraction techniques such as facade and bridge patterns.
- Reusability of Code
To minimize implementation time and improve efficiency, each part of the system has been designed as a component.

2 Proposed System

2.1 Overview

This section describes the requirements of the distinct proposed software architecture.

The system we propose is designed as a collection of services, which uses the DWARF middleware to communicate with each other. These services can run on separate hardware components e.g. wearables, laptops, that are then networked together to provide the desired functionality.

2.2 Subsystems and Services

The system is divided into several subsystems, which are described in terms of their services.

2.2.1 Subsystem Decomposition

The subsystems resulting from the subsystem decomposition are:

- Application
provides the core functionality of the system
- Context
provides location information and interprets the interactions of the user
- Network
provides the communication platform for the other subsystems
- User Interface
provides a graphical user interface
- Taskflow Engine
provides a taskflow-engine
- Expert System
provides repair information
- Persistent Data Storage
provides a data storage

2 Proposed System

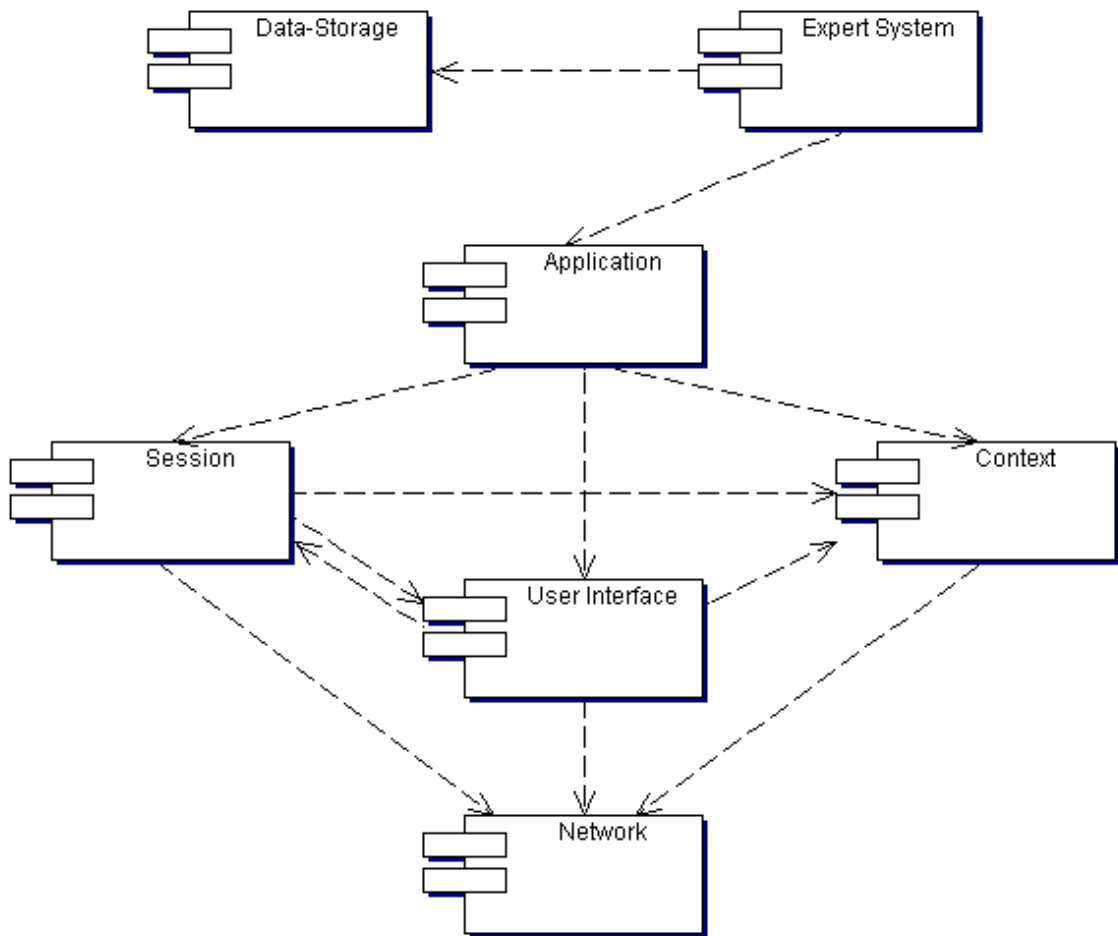


Figure 2.1: Dependencies between the subsystems

2 Proposed System

In figure 2.1 you can see the dependencies between the subsystems.

Figure 2.2 shows how the different subsystems communicate with each other and how they work together. They consist of DWARF services each having so called Needs and Abilities in the DWARF terminology. DWARF services offering the Ability A are connected to DWARF services having the Need A automatically by the DWARF framework. In this figure only those DWARF services of the subsystems which communicate with DWARF services from other subsystems are included. They, as well as the internal DWARF services of the subsystems, will be described in detail in the following sections.

2.2.2 Subsystem Services

This section describes the services each subsystem provides along with their needs and abilities.

2.2.2.1 Services of the Application team

These are services provided by Application team. The services have been collected under a common subsystem named Application.

1. Application

The Subsystem **Application** consists of three services:

- a. GarageServerApp
- b. WearableApp

The individual service consists of the following needs and abilities:

GarageServerApp Provides the core functionality that runs on the central server in the garage.

- Needs:
 - FindTaskflow
 - UserAction
 - WearableApplicationMessage
- Abilities:
 - GarageServerApplicationMessage
 - TaskflowAction
 - TriggerTaskflowTransition
 - UIEConfigurationData
 - UserInput

2 Proposed System

Version 17

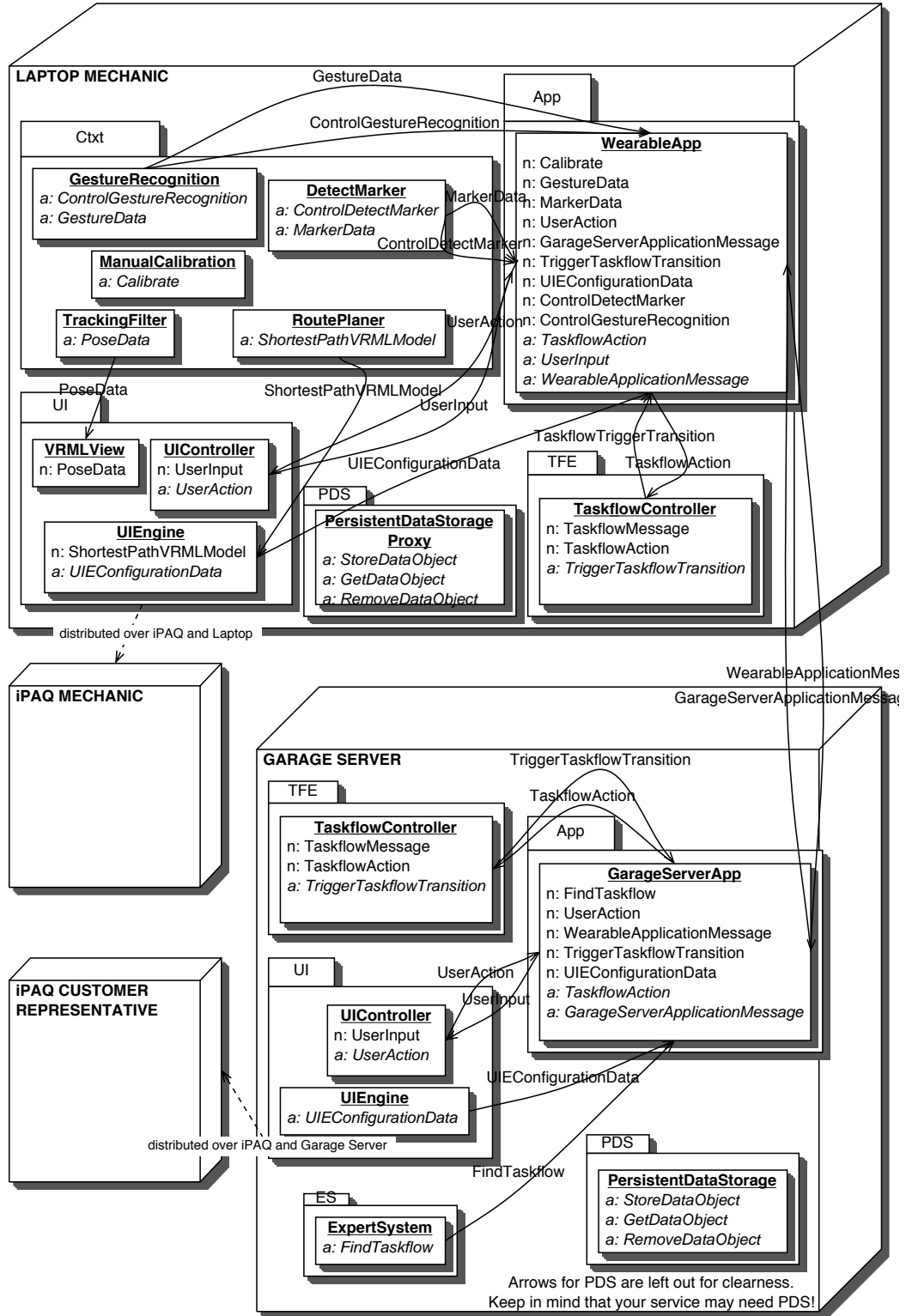


Figure 2.2: Needs and Abilities for inter-subsystem communication

2 Proposed System

WearableApp Provides the core functionality that runs on the mechanic's wearable.

- Needs:
 - Calibrate
 - ControlDetectMarker
 - ControlGestureRecognition
 - GarageServerApplicationMessage
 - GestureData
 - MarkerData
 - TriggerTaskflowTransition
 - UIEConfigurationData
 - UserAction
- Abilities:
 - TaskflowAction
 - UserInput
 - WearableApplicationMessage

2 Proposed System

The Needs & Abilities of the services described above are shown in the following figure:

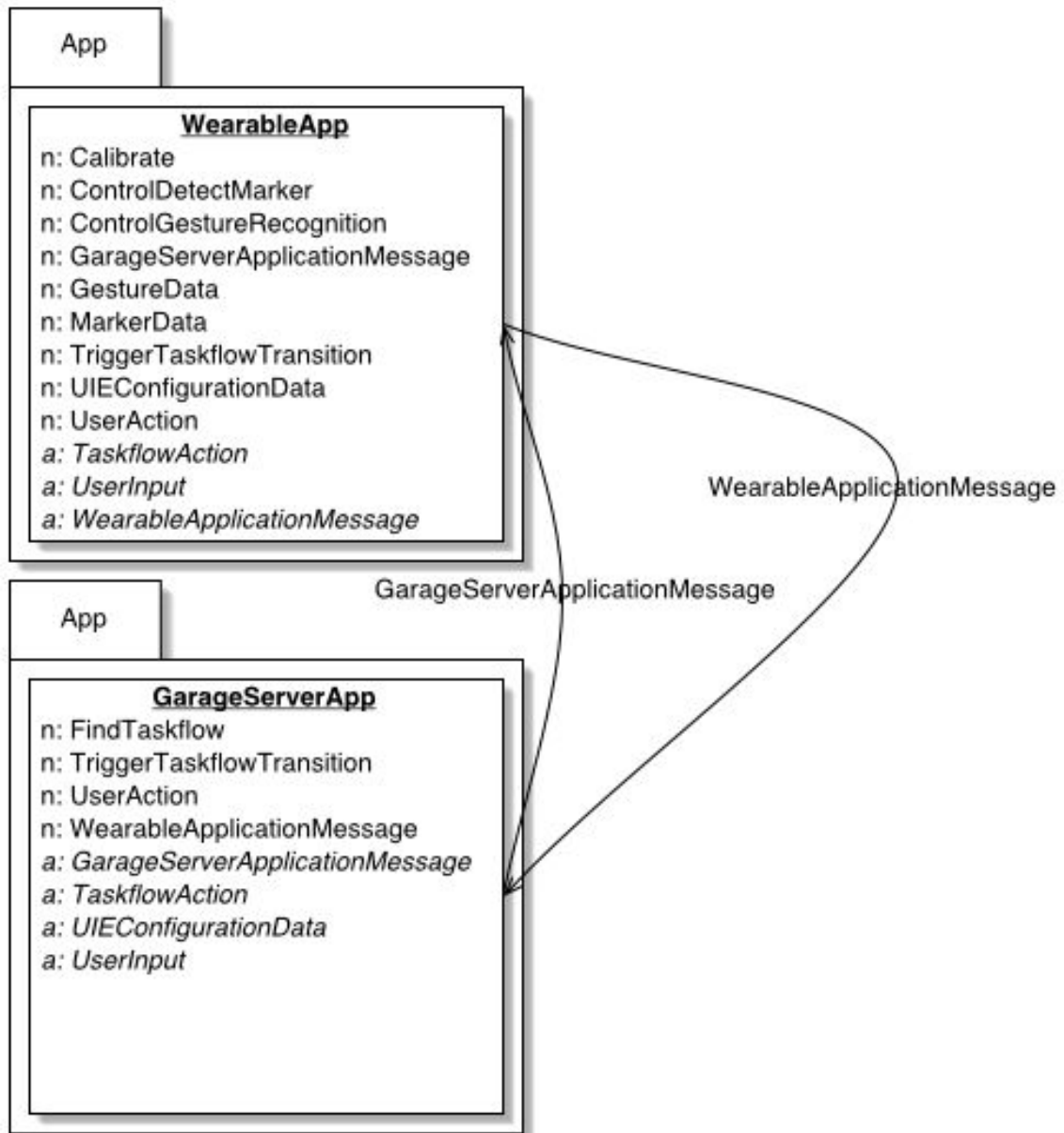


Figure 2.3: Needs & Abilities of *GarageServerApp*, *WearableApp*

2.2.2.2 Services of the Context team

The purpose of the context subsystem is to provide location information as well as interpretation of the user's interaction with the world. It's functionally split into three parts:

- i. Tracking subsystem and the world model
- ii. Gesture recognition and
- iii. Marker tracking

The world model is a hierarchically structured, abstract representation of the user and his environment. E.g., in the TRAMP scenario, it contains information about the garage, the parking lot, the mechanic's position, the administration building, etc.

Each object in the world is called a *thing*, identified by some unique ID. In the world model, things are stored in a tree like structure in which a thing, i.e. a node in the tree, can have children with objects associated with the node. The garage, for instance, could have the sub-nodes "administration building" and "parking lot", which itself contains some "parking spots" as children. These associations are set up in a way that represents the localization of objects in the real world. Moving objects such as users can be represented in dedicated areas of the world model.

Each thing has properties. One of these properties relevant to the context subsystem is each objects pose information. "Pose" stands for the combination of position and orientation vectors. For static objects in the real world, the corresponding thing contains the objects pose in its pose property. The problem arises when looking at moving objects such as the user. For moving objects, the pose information has to be updated continuously.

The tracking subsystem comes in here. Each Thing in the world can have a reference to a *TrackingModule*. A TrackingModule is the abstract representation of a service that provides location information. Examples for TrackingModules are hardware devices, which provide raw tracking data, as well as filters, which do sensor fusion on measurements from other TrackingModules. This sensor fusion is extremely important, as the TRAMP system incorporates various different tracking devices, such as a GPS tracker which only works outdoors, an inertial tracker which provides relative orientation information but suffers from numerical drift, and a marker tracker, which only works in specially prepared environments.

All these connections - Filter-TrackingModules with hardware TrackingModules, and Things in the world with these Filter-TrackingModules, have to be set up by some dedicated TrackingManager component, which also provides an interface for calibration issues. Calibration is an important point, especially for the inertial tracker, which needs initial pose estimation to work correctly.

To sum up - at system startup, the application has to ask the TrackingManager for calibration. Afterwards, it can query the world model service for pose data of the user.

The inertial tracker can be used for different purposes: It can not only serve as a relatively exact tracking device - its orientation information can also be used to recognize special movements of the user - *gestures*. The Gesture Recognition service, as soon as it is activated, continuously queries the inertial tracker and interprets the users movement. If a characteristic

2 Proposed System

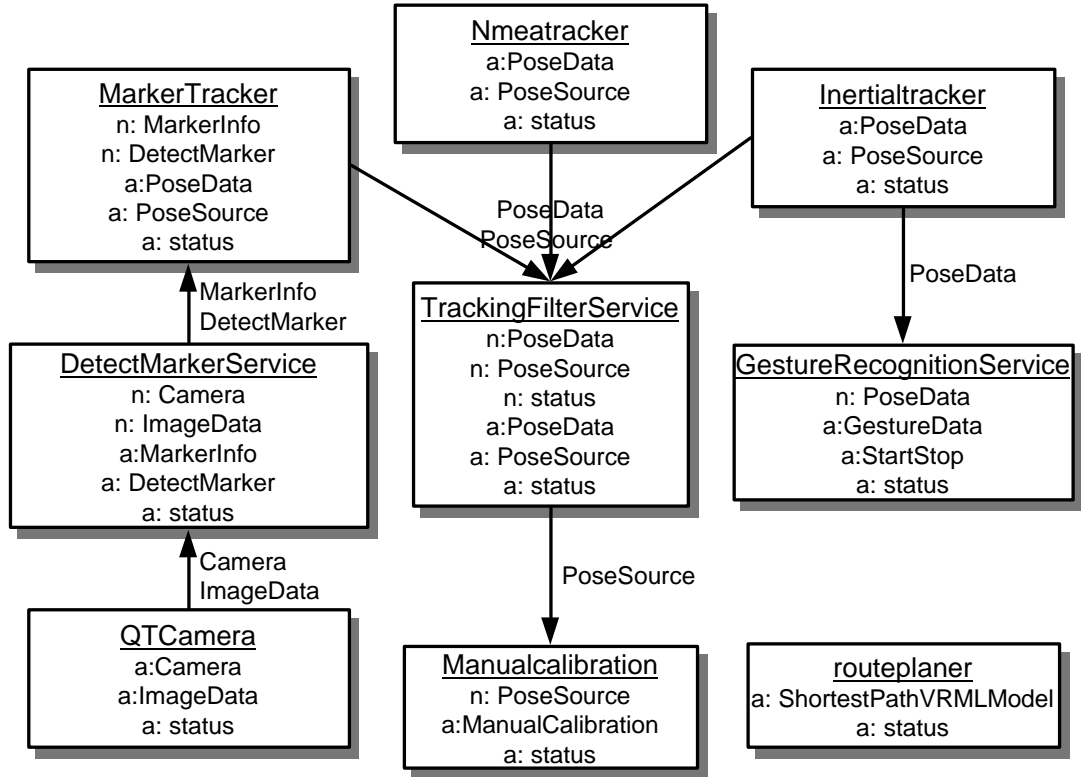


Figure 2.4: Needs And Abilities of tracking services and related stuff

movement pattern, associated with a gesture, is recognized, it issues the description of the recognized gesture to the event bus, where it is forwarded to other modules.

The *Detect marker* service utilizes the camera, represented by the CameraData service, to scan the user's environment for predefined optical marks, so-called *markers*. The appearance of a marker can have different meanings: It can be interpreted just as an optical sign to signalize the user that he is entering some special area (maybe some control room). It could also be some hint for the system that the user has manipulated his environment: In the car maintenance scenario, the mechanic opens the damaged car's fuse box, which is then recognized by TRAMP and causes it to continue with the next instruction. So the opening of the fuse box makes some marker appearing - the marker tracker detects this manipulation and issues this observation to the Taskflow engine, which in term interprets the manipulation and continues with the next step of the user's current task.

The Detection of Markers is computationally expensive, so services interested in the observation of a marker need to order the Detect Markers service to watch out for the specified marker only when required.

2 Proposed System

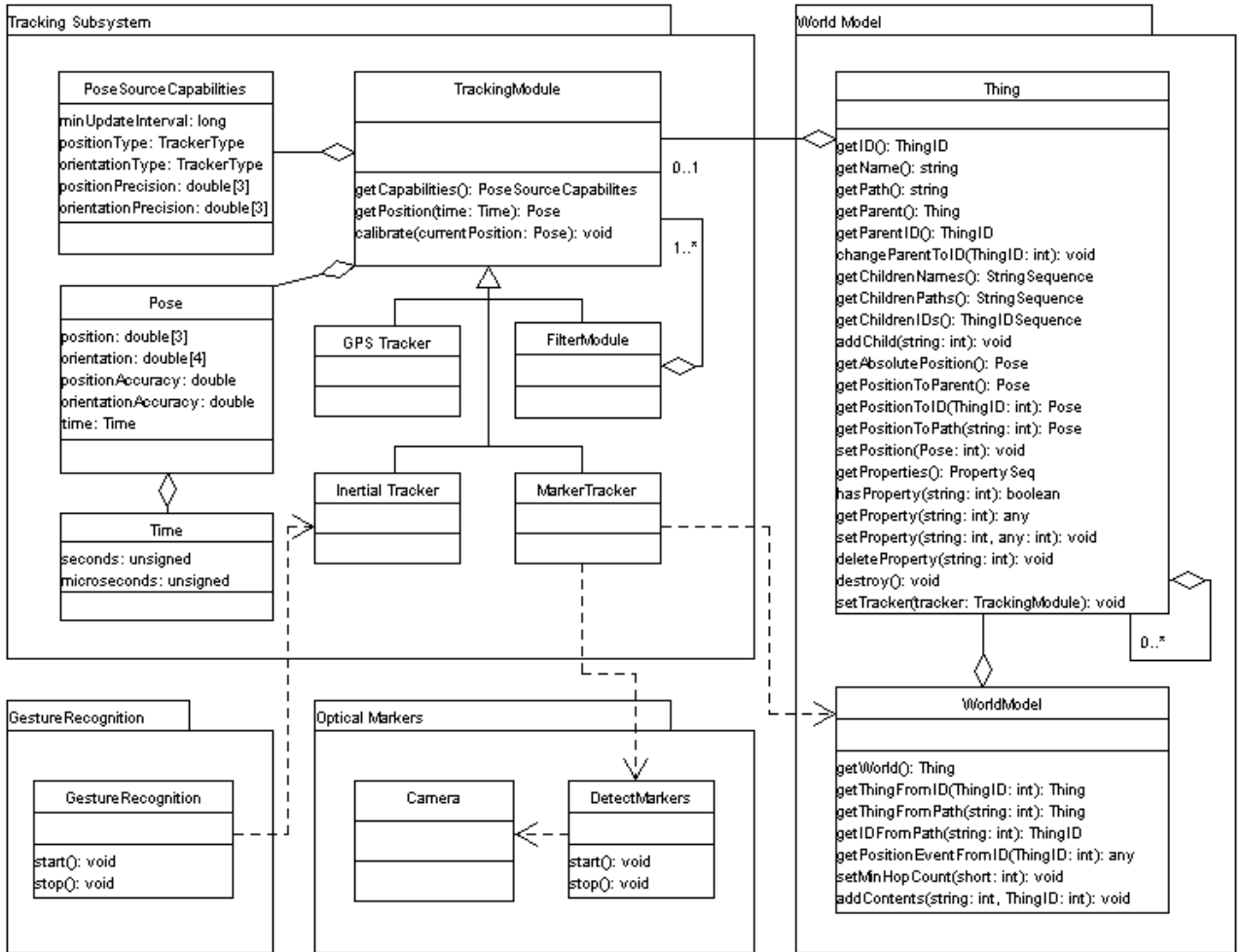


Figure 2.5: Tracking subsystem class diagram

The *Marker Tracker* - which connects the Detect Markers service and the tracking subsystem - has knowledge of the world model. It looks up all markers possibly visible at the users current position, and gives them to Detect Markers service to watch.

To do so, it needs some initial position estimate - afterwards it can update its position measurements continuously, as long as at least one marker is visible. The Marker Tracker also utilizes the fact that it knows the size of the detected marker (saved in the world model). This information is used to compute the estimated position of the user relative to the marker.

In the subsequent sections, each of the services mentioned above is explained in greater detail.

ManualCalibration The "ManualCalibration" module mainly provides the means to calibrate every Tracking Module from the tracking subsystem service. The corresponding calibrate() method needs the unique ID of the specific tracker (provided in form of a reference

2 Proposed System

of the corresponding DWARF module) and returns a boolean value indicating a successful or failed calibration attempt. Every tracker is calibrated by calling its own `calibrate()` method and providing a initial pose as parameter.

In addition to that the module implements a function "GetTrackersToCalibrate" that queries every tracker registered with the tracking subsystem and returning a complete list of the ID numbers representing tracking modules that need to be calibrated. Whether a tracking module has to be calibrated or not is determined by the abilities of the module.

For calibrating a given device, the user will go through several steps of a calibration routine (e.g. "turn straight north"). In order to provide the user with the correct sequence of commands, the User Interface service needs more information about how to calibrate a given device. Therefore the Calibration module could return a "CalibrationInformation" object for each tracking module referenced by an ID number. The easiest approach would be to return the type of the tracking device (e.g. compass) and let the UI service determine the command sequence to calibrate this tracker.

A possible flow of events for a complete calibration might be as follows:

- i. The User initializes the calibration of a tracker
 - ii. The UI service queries the Calibration module for all tracking devices that need to be calibrated
 - iii. The UI service then receives a list of ID numbers. For each ID number in that list it
 - a) queries the Calibration module for the calibration information corresponding to the given ID
 - b) displays the calibration sequence specified by the received information object
 - c) calls the `calibrate` method of the Calibration module with the given ID of the tracker and the actual step of the calibration sequence.
- Needs:
 - **PositionData** to find modules that need to be calibrated or can provide reference position
 - Abilities:
 - **ManualCalibration** to provide access to the ManualCalibration modul

World model The world model, mainly described in [12], is the main device for communicating position information. Each Thing in the world model can be tracked by some "Tracking Module" service. The user of the application is also a Thing in the world, so if the application needs position information about the user, it queries the World Model service, which then in term queries the referenced Tracking Module service. Within the context subsystem, the World Model service is used by the Marker Tracker. The world model data is loaded from an XML file, which is retrieved from the persistent data management (??). Although the World Module should know as much as possible about the user's environment, the main objective

2 Proposed System

of the service is to be independent from a central database. This has the disadvantage, that the World Model can see only a small "window" of the environment, but this demand is absolutely central when designing a distributed system as TRAMP is.

- Needs:
 - **PositionData** to connect Things with TrackingModules
- Abilities:
 - **WorldModel** to provide access to the world model database

Tracking Manager This service is responsible for setting up the tracking infrastructure. Different tracking devices need to be connected to the appropriate Things in the world model. As there can be different Tracking Modules giving pose data for the same Thing, the Tracking Manager is responsible for adding a suitable filtering module for data fusion. This is extremely important, as the system incorporates very different tracking devices as GPS, which only works outdoors, and the Marker Tracker, which only works in an environment equipped with markers.

All tracking devices and the "Tracking Manager" services inherit the TrackingModule interface, which provides description about the tracking capabilities as well as means for calibration and querying for pose information. This pose information is always accompanied by a time stamp and a error variance prediction.

- Needs:
 - **PositionData** to find out which TrackingModules are running
- Abilities:
 - *None*

Gesture Recognition With this service, simple head movements can be recognized (with the help of an inertial tracker mounted on the user's head) and interpreted as user commands. A module which wants to make use of this feature has to subscribe to an "input command" channel via the CORBA notification service and to start the recognition by calling the *startRecognition* method. As soon as either a head nodding or shaking (meaning yes or no) is detected, the gesture recognition service pushes an event containing the gesture type (and, if possible, a measure of the reliability) to this channel. The notification service takes care of sending the event to each of the subscribers by invoking a specified method. If a module doesn't need gesture recognition anymore, it has to call the *endRecognition* method.

Note that the phases in which gesture recognition is active should be kept as short as possible to avoid waste of computation time. Due to that, the recognition will automatically stop after a default time period (e.g. 20 seconds). This default value can be substituted by passing a parameter to the *startRecognition* method, which makes it possible to realise something like "Look for gestures for the next 5 seconds" easily.

2 Proposed System

- Needs:
 - **PositionData** from the inertial tracker
- Abilities:
 - **GestureData** on the event bus

Detect Marker The Detect Marker service can be used to analyze the "ImageData" of an optical input device ("Camera"). These inputs can be searched for optical markers, which are well-known and easily recognizable graphical structures attached to things in the real world. If a marker is detected, it is possible to calculate its position and orientation. The quality of the results depends on the quality of the marker, its distance, the quality of the input device, and the duration of an unobscured visibility.

Many different types of optical markers are possible, ranging from a big red blinking light to something as sophisticated as a bar-code. The current implementation of the "Detect Marker" service is limited to markers complying the capabilities of the ARToolkit [6]. The ARToolkit is a software library for Augmented Reality which particularly supports the detection of certain markers. ARToolkit markers are just a sheet of paper (about letter size) with a black square including a white square in the middle. That white square can contain any black and white image that fits in, and is suitable for identifying the marker. Of course the structures of the image may not be too fine, so they can be interpreted by ARToolkit correctly. A good marker should be detected within a distance, which is up to 10 times the marker's size.

A service wanting to use "Detect Marker" must register with it, and specify the marker to be detected. After a marker was registered, the service which registered it is notified every time the marker is detected. A marker being visible over a certain period of time, can cause many notifications during this period. Several markers registered by several services can be registered at a time, if system resources allow it. All services registered will be notified, which can be more than one per marker. If a service does not need to know about a marker any more, it can deregister it and will not be notified any more.

- Needs:
 - **ImageData** from a camera
- Abilities:
 - **MarkerData** - provides position, orientation, etc. of optical markers

Routeplaner The Route Planer service provides information of the fastest path from the user's position to a destination or optionally from an arbitrary starting point to a destination. It is possible to restrict the allowed paths by providing another parameter indicating which type of movement is allowed. This way it is possible for the user to obtain the fastest path he can use with his car.

The Route Planer is able to create a sequence of waypoints, each waypoint containing its position as well as a description of how to proceed to the next waypoint (e.g. "Turn right

2 Proposed System

and cross the bridge.”). Internally some kind of a graph containing a representation of a part of the world’s infrastructure is used to retrieve a valid path from the starting pose to the destination pose. The path description is automatically extracted from this graph, too.

- Needs:
 - **WorldModel** to find the user’s current position
- Abilities:
 - **RoutePlanner** to make this service available to other modules

Tracking Module The Tracking Module is the super-class which is inherited by all other modules in the tracking system.

- Needs:
 - *None*
- Abilities:
 - PositionData - provides pose data - NOT public, other subsystems have to query the world model!

Marker Tracker The ”Marker Tracker” service uses optical marker tracking for (rough) position estimation. It queries the ”World Model” service for the last position estimation, as well as for markers available in the current area, and then orders the ”Detect Marker” service to watch these markers. As the ”Marker Tracker” is registered to the ”Detect Marker” service, it constantly receives information about the detection of the markers. The absolute position of each marker is known from the ”World Model” and the position relative to the camera can be calculated by the ”Marker Tracker” and transmitted with the ”MarkerData”. This allows calculating the absolute Position of the camera which is used for ”Marker Detection”, which usually will be mounted to the subject of the tracking effort.

The precision of the calculated Position depends on many factors: the quality of the markers, the camera, the ”Detect Markers” service, the precision of the data in the ”World Model”, and so on. It is especially important that the real world is equipped with enough markers, which surely is not possible everywhere. However this Tracker might be very usefull in complex indoor environments, which often obscure the other tracking-devices (e.g GPS, magnetic compass), but offer enough opportunities to attach markers. Often industrial buildings already have several signs attached to important places (e.g. icons marking a starway, or number codes discribing floors and corridors), which could be used by the ”MarkerTracker”, if a ”Detect Marker” service can recognize them. Anyway, to make this Tracker really usefull, some intelligent filtering and cooperation with other Trackers is unavoidable.

- Needs:
 - **MarkerData** from the Detect Marker service for receiveing the marker parameters

2 Proposed System

- **WorldModel** to find markers that might be visible from the user’s current position
- Abilities:
 - **PositionData**

GPS Tracker The ”GPS Tracker” is an implementation of the ”TrackingModule” service that wraps the GPS and compass hardware driver and provides the raw pose information to be pulled by the World Model or some filter TrackingModule service. This service is part of the tracking infrastructure set up by the Tracking Manager service.

- Needs:
 - *None*
- Abilities:
 - **PositionData**

Inertial Tracker The ”Inertial Tracker” service wraps the Inertial tracker hardware driver and provides the orientation information to be pulled by the World Model or other tracking modules. This service is also being monitored by the ”Gesture recognition” service, which uses the orientation information for detecting movement patterns and recognizing them as gestures.

- Needs:
 - *None*
- Abilities:
 - **PositionData**

Camera The camera service wraps the operating-system- and hardware-specific camera driver and provides ImageData to other modules. As this requires high-bandwidth transportation, data is transferred via shared memory. This service is used by the Detect Marker service.

- Needs:
 - *None*
- Abilities:
 - **ImageData**

2 Proposed System

Central data structures: Pose and PoseSourceCapabilities The Pose struct is the central data structure for communicating position and orientation information. It contains two 3D vector representations for position and orientation data, vectors indicating the (current) predicted error variance of all these measurements as well as a time stamp.

In the PoseSourceCapabilities, each TrackingModule can describe itself for connection purposes. It contains information about the update interval, which is important, since a filter needs to predict, when new measurements are available. It also indicates, whether the position and orientation measurements are absolute or relative in the first (velocity) or second (acceleration) derivative. This is important, since it has a major impact on the numerical drift involved in each measurement, and thus on the need for calibration. For each measurement dimension it stores the (static) predicted error variance, which is set to infinity if measurements for this dimension are not computed by the TrackingModule.

2.2.2.3 Services of the Network Subsystem

The Services of Network Subsystem can be grouped into the following three types, Communication, Location, Service Management, which maps to the three subsystems of DWARF middleware: Communication System, Location System, Service Management System.

Communication The communication subsystem encapsulates the functionality of communication between Services. It manages communication resources such as event channels, shared memory blocks and TCP sockets.

The communication subsystem creates Connectors when instructed to do so by the Service Manager. A Connector manages Communication Resources at one end of a connection between one Service's Need and another Service's Ability. The Service Manager can actively tell the Connector to connect to another Service, or it can tell it to passively wait for incoming connections. Once a connection has been established, the Service Manager passes the Service's Need or Ability a reference to the Connector.

The Service's Need or Ability then determines the protocol that the Connector supports, and extracts a reference to the protocol-specific communication resources from the Connector. It then accesses these communication resources directly, bypassing the Connector, to communicate with the other Service.

The following Connector of Communication Subsystem will support two major types of data transfer services in the TRAMP project: Transfer small amount of data(such as navigation information), Transfer big amount of data(such as video/audio streams, bulk data).

- The CorbaObjExporter Connector allows a Service to export a CORBA object. The exporter supports the SvcConnCorbaObjExporter interface, which has only one method setObject. This method passes the Connector a reference to the object which the service wish to export.
- The CorbaObjImporter Connector allows a service to import a CORBA reference to a CORBA object. The importer supports the SvcConnCorbaobjImporter interface, which has only one method getObject. This method returns a reference to the object that the other Service has exported.

2 Proposed System

- The `NotifyStructuredPushSupplier` Connector supports the `SvcConnNotifyStructuredPushSupplier` interface, with only one method `getProxyConsumer`. An event supplier can connect itself with this method.
- The `NotifyStructuredPushConsumer` Connector supports the `SvcConnNotifyStructuredPushConsumer` interface, with only one method `getProxySupplier`. An event consumer can connect itself with this method.
- The `RTPServer` Connector supports the `RTPServer` interface, with the method `RTPServerSend`, which enables the sending of real time streaming data from the server's side.
- The `RTPClient` Connector supports the `RTPClient` interface, with the method `RTPClientReceive`, which enables the receiving the real time streaming data from the client's side.
- The `HTTPServer` Connector supports the `HTTPServer` interface, with the method `HTTPServerPost`, which enables the sending of unstructured data from the server's side.
- The `HTTPClient` Connector supports the `HTTPClient` interface, with the method `HTTPClientReceive`, which enables the receiving of unstructured data from the client's side.
- The `SharedMemoryReader` Connector supports the `SharedMemoryReader` interface, with the method `SharedMemoryReaderRead`, which enables the concurrent writing of same data block in the shared memory.
- The `SharedMemoryWriter` Connector supports the `SharedMemoryWriter` interface, with the method `SharedMemoryWriterWrite`, which enables the writing of same data block in the shared memory with help from Semaphore.

The technical implementation of the different services are listed in the following subsections:

- **Transfer Navigation Information**
This service can be used by the context team or other teams which need to transfer small but well defined information packages like three-dimensional coordinates with nearly stable and guaranteed delivery time. For this CORBA Event Services should be used. The idea is to encapsulate the data in a CORBA Event and provide a rapid propagation to requesting components without establishing administration overhead. The idea is to generate a constant flow of fine grained information packages which can be consumed by the receiver or discarded. The information source supplies a uniform stream of information available for all subscribers and doesn't wait for feedback except de-registration. This means that a push model of information transfer is employed. Because there is no feedback from the consumers the source has not to care about whether the information is actually used or not. The consumer has the choice to decide individually about the acceptance of an information package depending on the currently available sources. This enables the consumer to choose between different sources with maybe different qualities of service. It is possible to be registered with a bunch of source

2 Proposed System

which satisfies all the same need at the same time. Time stamps of packages can be used as one way for the consumer to select between packages of different source and ensure good quality of consumed information. Other kinds on quality of service control can be applied also. Using time stamp information makes it also possible to keep the list of possible sources up-to-date. Either with a ranking of currently available sources according to their delays or cost by removing information sources not further available. So in the end this service will regularly propagate given navigation information to all registered users and does not wait for feedback. To keep the subscribers list up-to-date there may be either a regularly re-registration of the consumers objects or an active check performed by the source.

- **Transfer Information Package**

The Transfer Information Package service is responsible for the technical data transfer from the database manager, the input system to the output system. The technical data is grouped into two types: the first is about the car's maintenance records, which includes the car's serial number, type, manufacturer, etc. ; the second is about the technical instruction, which the mechanic should follow in repairing the car. Both types share the same characteristic: the data is complex, which can only be implemented with record-like data. Due to the distributed structure, network communication in our system, the data is to be encapsulated in the CORBA Events, and transferred between different objects.

The CORBA Event Service decouples the communication between objects. The Event Service defines two roles for objects: the supplier role and the consumer role. Suppliers produce event data and consumers process event data. Event data are communicated between suppliers and consumers by issuing stand CORBA requests.

There are two approaches to initiating event communication between suppliers and consumers, and two orthogonal approaches to the form that the communication can take.

The two approaches to initiating event communication are called the push model and the pull model. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

The communication itself can be either generic or typed. In the generic case, all communication is by means of generic push or pull operations that take a single parameter that packages all the event data. In the typed case, communication is via operations defined in IDL. Event data is passed by means of the parameters, which can be defined in any manner desired.

In order to support the communication between multiple suppliers and multiple consumers asynchronously, an event channel is introduced in Event Service. An event channel is an intervening object, which is both a consumer and a supplier of events. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests.

- **Transfer With High Bandwidth**

This section describes the technology we are going to use to transfer data demanding

2 Proposed System

a high transportation bandwidth with real-time properties. The represented methods of transmission mainly address continuous media, which e.g. is provided packed into an audio/video stream, real-time-control or distributed simulation. We have got to pay principal attention to timing reconstruction, loss detection, security and content identification.

Two investigated technologies offer high-performance properties combining many of the above mentioned requirements. The following passages will introduce you to the Realtime Transport Protocol and the idea behind (Distributed) Shared Memory. RTP stands for Realtime Transport Protocol and is treated by the AVT WG (Audio-Video Transport Working Group) within the IETF (Internet Engineering Task Force).

The Realtime Transport Protocol is a widely used Internet-standard protocol supporting the transmission of real-time data such as audio and video over packet switched networks. Since the approval of RTP by the Instructional and Electronics Support Groups (IESG) as an internet proposed standard on November 22 in 1995, it is also known as RFC 1889 (A Transport Protocol for Real-Time Applications) and RFC 1890 (RTP Profile for Audio and Video Conferences with Minimal Control). Since RTP is transport-independent, it can be put on almost any transport layer protocol such as UDP/IP (User Datagram Protocol/Internet Protocol), CLNP ConnectionLess Network Protocol), IPX (Internetwork Packet Exchange), AAL5/ATM (Asynchronous Transfer Mode Adaptation Layer 5) and others. Its usability include media-on-demand as well as interactive services like Internet telephony.

RTP is composed of a data and a control part implemented by RTCP (Real-Time Control Protocol). RTCP provides mechanisms to evaluate quality of data delivery, which is based on periodically sent RTCP packets returning with information about streaming quality and the number of users participating. The Real-Time Control Protocol is also used to verify these users by interpreting the RTCP SDES (source description) packets, which contain information about the participants like the name, telephone number, email address and so on (source identification). Furthermore RTCP supports gateways like audio and video bridges as well as multicast-to-unicast translators, and takes care of the synchronization of different media streams.

RTSP (the Real Time Streaming Protocol) is an application-level protocol designed to work with lower-level protocols like RTP. It establishes and controls streams of continuous audio and video media between the media servers and the clients. Using RTSP the client is able to request a media stream by telling the server to setup a session and to send the requested data. If a conference is already established, the media server can be asked to participate by playing back media or to record the conference. If additional media is becoming available during a transmission, the server can notify the client about it and add it to the transmission queue. RTSP provides the same services on streamed audio and video just as HTTP (Hypertext Transfer Protocol) does for text and graphics. As it uses a similar syntax, the most extension mechanisms to HTTP can be applied to RTSP, so there is also an URL, which identifies each media stream. While HTTP is an asymmetric protocol where only the client issues a request and the server responds to it, using RTSP both the media server and the client can issue streaming requests. Shared memory (SM) provides the fastest possible communication between processes running on the same machine. A shared memory segment is part of the main memory,

2 Proposed System

which can be accessed by several processes at the same time. We will use this technic to provide transparent memory areas for delivering huge data amounts between independent processes. Operating system functions of Unix and its descendants are able to create and administrate shared memory segments. However access to these segments are not automatically synchronized, as a matter of fact implementing semaphores, which enable mutual exclusion on critical memory areas are necessary. Semaphores and the operations on them are also made available by the Unix operating system, so perhaps it is not necessary to create their functioning on our own. Processes, which want to reserve a closed semaphore are blocked and unblocked by the operating system as soon as the semaphore is opened again. The advantage of letting the OS handle the administration of semaphores results in higher performance, as processes do not have to run an active waiting loop. To work with a shared memory segment, it is necessary to attach it to the address rooms of the processes, that are going to make use of it. Afterwards semaphores handle synchronization and mutual exclusion. After the service life of the shared memory segment it is detached from the participating process address rooms.

There is an open-source shared memory library available called MM, which was originally written for use inside upcoming Apache webserver moduls. At this time we do not know yet, whether we are going to use the MM library, which claims to simplify the usage of shared memory between forked processes under Unix platforms. The Distributed Shared Memory (DSM) model provides a common address space that is shared among all processors in a distributed memory multiprocessor or a computer network system. It lets programs access data in traditional virtual memory and supports both parallel applications and groups of applications in which shared data items can be accessed directly. In DSM systems data blocks can be moved between secondary memory and main memory as well as between main memories of different nodes. When a process requests access to data located in the shared address space, a mapping manager mirrors the shared memory address to the physical memory. The mapping manager is implemented either in the operating system kernel or as a runtime library routine.

Problems, that are often encountered when using DSM are synchronization failures and data inconsistencies. If our system uses DSM, it will most likely handle huge data blocks, which even increases these failures and requires support for fault tolerance. A popular approach, the Backward Error Recovery (BER) enables an application which encounters an error to restart its execution from an earlier, error-free state.

There are already a couple of different implementations of DSM available, most of them are open-source. The Distributed Computing Environment (DCE) software of the Open Software Foundation (OSF) allows distributed programming using DSM with the main emphasis on security issues. SAM, also a distributed shared memory system, is a portable run-time application, which also supports a global name space, access to shared information organized in user-defined structures, and automatic caching and prefetching of shared data.

- **Transfer Requested Bulk Data**

We will also provide a service for transferring requested bulk (unstructured) Data over the Network. This service will expect a series of characters as input and will transmit them to the other side using a buffered transfer mode. As this data (for example the technical manuals formerly known as IETM) may be rather large, we put our main

2 Proposed System

focus on efficient bandwidth utilisation during the transfer; also, the user should be able to use the incoming data while it is being retrieved and should not have to wait for the transfer to finish, nor should the entire document have to be present in the RAM at any time. A stream oriented interface will be provided to make the service easy to use.

As an additional feature (depending on the technology employed), the client may be able to request only the portions of a file he needs. Note though that this service will have no knowledge about the structure of the information transmitted, so the parts will have to be specified in terms of byte ranges, or other high level services may be layered on top of this service, which parse and 'understand' particular file types. For implementing this service, we will use a lightweight, TCP-based protocol with low bandwidth overhead. The *hypertext transfer protocol* (HTTP) appears to be a good choice, as it fulfills all the mentioned specifications and is a globally accepted and widely used internet protocol that has received *proposed standard* status by the *internet engineering task force* (IETF). Among others, the most popular internet service, the world wide web, is powered by the http protocol. While the original standard HTTP/1.0 (as specified in RFC 1945 www.ietf.org/rfc/rfc1945.txt) is very simple and easy to implement, version HTTP/1.1 (RFC 2616) adds some interesting features that may prove useful for us, for example the aforementioned limiting of file parts to be transferred. As an added plus, it should be easy to use one of the numerous free function libraries for the low level transfer.

Location The location subsystem provides the basic functionality of locating Services. The location subsystem does not know about communication protocols, connectors, Services, Needs, or Abilities. It simply deals with Offers and Requests. Offers consist of a location and a set of attributes, and Requests are predicates over these attributes. The location subsystem periodically tries to find Offers to match its Requests. The Service Manager creates Offers and Requests from each Service's description and registers them with the Service Locator. It maps a Service's Abilities onto Offers, telling the location subsystem to advertise these on the network. Analogously, it maps a Service's Needs onto Requests, which the location subsystem tries to answer. The location subsystem is generally not directly used by other Services. Instead, it is used by the Service Manager. The DWARF service access the Service Location Functionality only indirectly via the description of their needs and abilities.

The technical implementation of the different services are listed in the following subsections:

- Service Location

One main area of functionality for the middleware is locating services. Note that the term *location* refers to logical locations such as network addresses, not to physical *positions*. Many services have abilities that they can offer to other services. For example, trackers provide position data. The middleware must advertise these Services on the network so that other Services can use them. Conversely, the middleware must discover services that have been advertised, so that the HMD, for example, can receive position data from a tracker.

Several new protocols have been developed in the last few years to solve the problem of finding services in a dynamically changing environment. SLP, the Service Location Protocol, is a simple protocol for finding services. We will use SLP in designing the middleware.

2 Proposed System

SLP can locate services and in a network by service type and by attributes. It returns a URL indicating the service's location to the client. SLP does not provide any additional mechanisms for configuring or communicating with a service — once the client has the service URL, it must contact the service itself, using the appropriate protocol. SLP is primarily designed to fit enterprise-sized networks with shared services.

- Quality of Service Selection

The Quality of Service Selection service is responsible for the selection of the different services registered in the service manager in CORBA. Because many different objects of same type can perform services of different quality in our system, the service with the best quality available is preferred. SLP will be used to implement this service. Service Location Protocol is an IETF standards track protocol that provides a framework to allow networking applications to discover the existence, location, and configuration of networked services in enterprise networks. In SLP an agent is a software entity that processes SLP protocol messages. There are three types of SLP agents:

- User Agent (UA)

The SLP User Agent is a software entity that is looking for the location of one or more services.

- Service Agent (SA)

The SLP Service Agent is a software entity that provides the location of one or more services.

- Directory Agent(DA)

The SLP Directory Agent is a software entity that acts as a centralized repository for service location information.

The SLP Directory Agent is capable of retrieving the attribute of different services, which have registers in the repository. With this feature, the Quality of Service Selection will be implemented.

- Network Roaming

This service enables the wearable to roam between the different access points. The wearable can "roam" from one access point to another, with the software and hardware maintaining a steady network connection by monitoring the signal strength from in-range access points and locking on to the one with the best quality. Usually this is completely transparent to the user; they are not aware that a different access point is being used from area to area. Network roaming is enable by Mobile IP, which provides a scalable, transparent, and secure solution.

Version 6 of the Internet Protocol (IPv6) fulfills future demands on address space, and also addresses other features such as privacy, encryption, and better support of mobile computing. It has 128-bit technology available for addressing. When mentioning mobile devices and IP, it's important to note that a special protocol is needed to support mobility, and implementing this protocol – called "Mobile IP" – is one of the requirements for every IPv6 stack. Thus, if you have IPv6 going, you have support for roaming between different networks, with global notification when you leave one network and enter the other one. Support for roaming is possible with IPv4 too, but there are a number of hoops that need to be jumped in order to get things working. With IPv6, there's no

2 Proposed System

need for this, as support for mobility was one of the design requirements for IPv6. Due to the limitation of the current developing environment, and the low-priority of this issue in the TRAMP project, it will be not implemented in the current version, rather in the future version.

Service Management The Service Manager provides the initial interface to the DWARF middleware and possess the following features:

- **Active Service Descriptions**
Internally, the Service Manager consists of Active Service Descriptions. An Active Service Description exists throughout the life cycle of a Service and represents the Service within the Service Manager. It has a state reflecting the state of the Service
- **Coordination**
The Service Manager coordinates the other subsystems. It creates Requests and Offers in the location subsystem to satisfy a Service's Needs and to make its Abilities available to other Services. It instructs the communication subsystem to connect the Services together that the location subsystem locates.
- **Description and Registration**
An administrator can describe a Service before the Service is started. Services are described by creating Service Descriptions, which describe a Service's attributes, its Needs and Abilities, and the communication protocols it supports. When a Service starts, it registers itself with the Service Manager, which then associates the running Service with its stored description.
- **Starting and Stopping**
The Service Manager can create connectors for a Service's Abilities before the Service is even started. When such a Connector is connected to, the Service Manager can start the Service, potentially even loading it across the network (or even purchasing it rst). When all other Services have disconnected themselves from the Service's Abilities, the Service Manager can shut down the Service. This start-on-demand and stop-on-no-use mechanism conserves resources.

The following interfaces of Service Management will support the services in the TRAMP project:

- **DescribeServices**
Services can either be described in an XML file, or dynamically, using the Service Manager's DescribeServices interface.
The main operations of this interface are:
 - newServiceDescription creates an empty new Service description with the given name.
 - getServiceDescription returns a reference to the Service description with the specified name.

2 Proposed System

- ServiceDescription Interface

This interface manages the attributes, Needs and Abilities of one Service description. The main operations of the ServiceDescription interface are:

- newNeed creates an empty new Need description with the specified name.
- newAbility analogously creates an Ability description with

- AbilityDescription Interface

An Ability description describes one Ability of a Service, i.e. a functionality that it provides.

The main operations of the AbilityDescription interface are:

- newConnector creates an empty Connector description with the specified communication protocol name.
- getConnector and deleteConnector retrieve and delete Connector descriptions.

- NeedDescription Interface

Need description describes one Need that a Service has, i.e. a functionality of another Service that it depends on.

The main operations of the NeedDescription interface are:

- newConnector creates an empty Connector description with the specified communication protocol name.
- getConnector and deleteConnector retrieve and delete Connector descriptions.

- RegisterServices Interface

Once a Service has been started, it must register itself with the Service Manager. It does that through the RegisterServices interface. This allows the Service Manager to give it the Connectors it needs to communicate with other Services.

The main operations of the RegisterServices interface are:

- registerService registers a Service's Service interface with the Service Manager and associates it with the Service description of the specified name.
- registerServiceAndNeedAndAbility is a helper function allowing simple Services to register themselves with one call.

The technical implementation of the different services are listed in the following subsections:

- Start On-demand

In order to reduce coupling between the individual services, the services should only access each other through well-defined interfaces. The middleware can maintain an abstract description of each Services, including the interfaces it supports and what kind of other services it depends on. Aside from formalizing the dependencies between services, this also allows the middleware to start services on demand, since it knows about services even when they are not running.

The technology used here for such a description of services will be XML. The Extensible Markup Language (XML) is the universal format for structured documents and

2 Proposed System

data on the Web. XML is a set of rules (you may also think of them as guidelines or conventions) for designing text formats that let you structure your data. XML makes it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous. XML avoids common pitfalls in language design: it is extensible, platform-independent, and it supports internationalization and localization.

Upon startup, a service registers itself with the service manager, from where it can gain access to all of the middleware's functions. The service manager must know the service descriptions for the services that register themselves with it. The service manager receives the service description in the form of an XML file, and starts the corresponding service automatically.

2.2.2.4 Services of Persistent Data Storage

The persistent data storage contains the following types of data:

- static data:
 - WDL files (workflow description language)
 - Document files: i.e. IETMs, HTML files, media files (see GUI-team)
 - World model descriptions (see Context-team)
 - Configuration data
- dynamic data:
 - Protocol data for a taskflow: Saves the path of visited tasks inside a taskflow with additional data objects like notes and accounting information. This protocol data can be used to resume a suspended workflow and to calculate and validate the payment.
 - Car history: summarizes the protocol data of terminated job for a specific car.
 - Customer registration data

These three items describe ascending levels in the following data hierarchy:

l1: customer 1: n l2: cars 1 : n l3: histories 1 : n l4: protocols

For the final scenario l1 and l2 probably will be hard coded.

In a distributed augmented reality application, the requirements for a persistent data storage system are very demanding. On one hand, we have to deal with network disconnections, so it would be desirable that every device carries around a complete copy of all necessary data. On the other hand, resources of wearable devices are very limited, so this is not feasible.

To conquer that problem, we use an architecture that consists of two separate services: A central server and a local proxy server. The proxy server retrieves in advance from the central server all the data the other services need and makes it available to them while the network is disconnected. This approach has the advantage that it is transparent to the services running on the wearable, so that they do not need to buffer the data by themselves.

2 Proposed System

PersistentDataStorage The central data storage server. It is the place where all the data is stored. An interface is provided through which other services can persistently store and retrieve their data. It will then be written to the servers hard disc. The use of a simple structured interface enables a high complexity data storage infra-structure. This may be realized by files located on a server and identified by a unique URI (uniform resource identifier), data which has to be retrieved from a database and or generated on the fly or a simple file system. Of course, a mixture of all of these opportunities at the same time can be employed as well. A DWARF service implementing the PersistentDataStorage interface then simply act as a parser and dispatcher of the requests and the genuine data source becomes transparent to the requesting service. Due to this loose coupling the data storage system can easily be furnished according to special demands and properties of a system or third party legacy systems. Because integration of IETF delivering systems was cut off, in TRAMP1 the PersistentDataStorage-subsystem will rely on a simple file system based solution.

- Needs:
 - *None*
- Abilities:
 - **StoreDataObject**
 - **GetDataObject**
 - **RemoveDataObject**

PersistentDataStorageProxy A local proxy server. This service is located on each wearable device, acts on behalf of the central server and therefore provides the same interface to the rest of the application. It has an internal cache, so that repeated queries for the same resource are only carried out locally after checking the correct file version. This check is done according to an observer pattern (i.e. the server publishes filechanges to all subscribed proxies, who will then have the opportunity to update the files themselves). For a distributed application that needs to deal with network disconnections, an additional interface will be provided (in subsequent tramp-projects) that allows the application to specify a number of files that might be needed later. These files would be transmitted immediately and stored in the local cache to make them available to the application when the central server is not reachable (prefetching).

Disconnected data transmission is also possible in the other direction. When a component needs to persistently save its internal status while the network is disconnected, the proxy service buffers it until the central server becomes available.

As this prefetching requires sophisticated mechanisms to determine the data to prefetch in relation to a given context, this will be employed in future iterations on TRAMP and not for the current proxy realization.

- Needs:
 - *None*
- Abilities:

- **StoreDataObject**
- **GetDataObject**
- **RemoveDataObject**

2.2.2.5 Services of the Task Flow Engine

The taskflow engine controls the sequence of tasks - it offers successor states and checks transit conditions - and provides links to documents which are available inside a task.

setup Taskflow This is a description of a generalized cycle of events between the systems ui, application and session management for controlling the work of actors including the distribution of tasks to several actors and the aggregation of these tasks.

- actor x confirms to take over an offered task y, this is done by the user interface (see the paragraph below for information about the initial service providing task y to actor x)
- notification to the application subsystem that actor x is working on task y
- application calls the Expert System with FindTaskflow providing the task identifier y
- the Expert System translates the task identifier to a WDL-URI and returns it to the calling application subsystem which opens the corresponding taskflow
- the taskflow engine provides initial data (Taskflow Action) of the newly opened taskflow y to the main user interface of an actor (mechanic). This initial data contains a) a list of document URIs with arguments which are available in the current state (those arguments describe i.e. the document type as a requirement for interface abilities) and b) a list of possible successor states - including states in the taskflow of the actor itself and taskflow numbers of new tasks which can be distributed to new actors.
- in case a document should be retrieved the persistent data storage is requested to return the file by the provided URI

Referring to step 1 there must be an initial service providing task y to actor x. In the TRAMP scenario actor x is the customer representative known by the application team at system startup (after the registration of the garage server and the interface of the customer representative). Task y accords to an universal, initial taskflow with a static address which is the starting point for the control of all the following subtaskflows.

TaskFlowController This service creates a taskflow tree by parsing specific taskflow files described in WDL format (workflow description language) retrieved from the *Persistent Data Storage*. The nodes contain the links (in form of URI, Unified Resource Identifier) to the necessary documents (i.e. the information available for working on the task). With these links, the *User Interface* subsystem can request all necessary information from the *Persistent Data Storage*. A path in the taskflow tree represents the sequential steps, which a user can perform during the work he has to accomplish.

Each node represents a state of the *TaskFlow*. The selected node represents the current state.

2 Proposed System

- Needs:
 - **GetDataObject**
 - **TaskflowMessage**
 - **TaskflowAction**
- Abilities:
 - **TriggerTaskflowTransition**
 - **TaskflowMessage**

2.2.2.6 Services of the Expert System

The Expert System actually consists of one service, also called Expert System, which parses string queries to Taskflow-identifiers that are dispatched to TaskflowEngines.

Expert System

- Needs:
 - **GetDataObject**
- Abilities:
 - **FindTaskflow**

2.2.2.7 Services of the User Interface team

The UI's task is to interact with the user, by displaying information provided by other subsystems and by fetching user input and altering the data, if necessary. The User Interface Subsystem will provide a set of user-friendly interfaces, which will handle display and input of data in the process of interaction between the TRAMP system and end users. It is necessary for a correct display to know the position of the user or of certain objects in the real world; therefore the tracking subsystem provides the necessary data.

In turn, the UI subsystem is responsible for getting user input from the Application subsystem and delegating those events to the mediator subsystems. The mediator has to deliver the events to the viewers whose state is changed by the user input. The mediator can also send events to the output subsystem, to change the state of the user interface.

We should render to a markup language (HTML, VRML) that can be displayed in an appropriate browser (VRML- or web browser). For every GUI there is a controller that deals with incoming events. For almost all of those devices there already exist mark-up languages like HTML, WML.

2 Proposed System

Internal structure The UI subsystem is divided in the following infrastructure.

- i. the UIE (User Interface Engine) is the core of the layer, which controls configuration of the UI-Controller.
- ii. the UIController: is a DFA which synchronizes the different views and keeps track of them. Each view has an Adapter which is initialized by the UIE in the very beginning. The adapter can change views by using DOM, EAI (e.g. for VRML) in the runtime. Without it, the view changing would cause much time in transferring a new UIML description from UIE to views. You can imagin an Adapter as an Applet which runs in the background and can manipulate the views. For example, there are two buttons (one is 'add' and the other one is 'delete') in HTML file, if 'add' is clicked, a message will be sent to Adapter (through HTTP), and the adapter registers the state of the view in UIcontroller, then modifies the view with the permission of controller.
- iii. the adapter: can communicate with other services. The channel from views to adapter/manipulator can be any protocol, it is view specific. HTTP is just one simple example. The relations between the UIE, the UIController and the Viewer with its specific adapter are illustrated in figure 2.6..
- iv. the Viewers: a viewer consists of a manipulator (in form of an Adaptor) and the view itself. We have the following viewers:
 - a) IPAQ viewer
 - b) VRML viewer
 - c) Flash viewer
 - d) HTTP viewer

Services In figure 2.5 is the general User Interface subsystem structure and the interaction with other subsystems synchronously or asynchronously through event calls or method calls.

i. Display Repair Instructions

This service enables the user to get a display of Taskflow in an appropriate form. It needs to receive this taskflow data in CUIML from the TFE (Taskflow engine). This data will be used in turn from the UIE, to control configuration data of the UI-Controller in form of a method call. The UI-Controller has the ability to trigger taskflow transitions (event).

- ii. Navigation Display This service enables the user to see his current position within his surrounding in form of 3D VRML display inorder to simplify the navigation task. It needs to get the position data which is an event driven process, from the Context WorldModel service (2.2.2.2).

2 Proposed System

Design Goals

i. I/O devices mapping

Actor	Platform	IO
customer representative	IPAQ	touchpad, pen
customer	IPAQ	—
mechanic	laptop	HMD, tracker, gesture recognition, mouse
mechanic	IPAQ	touchpad, pen

ii. Performance characteristics

The performance of the user interface mainly is depending on the performance of TRAMP: The user interface is integrated into TRAMP, which is a very large program. The user interface itself does not slow down the system that much.

iii. Error Handling and Extreme Conditions

Errors such as data inconsistency or any extreme or unusual condition (e.g. network congestion, server unavailability or unreachability) should be detected by the system and displayed to the user.

The system should always display clear error messages and be able to make suggestions to the user as to what measures should be taken in order to fix the problem.

iv. System interfacing

The user interface subsystem primarily communicates with the user input devices such as a keyboard, mouse, a touch-pad, and possibly a microphone.

Data will be displayed or printed, and sound may be used to inform the user. The system should be able to display information clearly, even on small screens and, in the case of the handheld computers, be easy to read outside, or in unusual environments.

v. Quality Issues

The user interface must be absolutely reliable and trustworthy.

2.2.3 Interaction of Subsystem Services

This section shows the interaction between the subsystems resulting from the user tasks described in the RAD.

2.3 Hardware/Software Mapping

Based on performance considerations and client feedback, our target deployment platform consists of a wearable Compaq iPAQ with limited resources, running Linux, and laptop(s) for processing computation-intensive tasks, both running Mac OS X. Development is done on both operating systems, mainly on Mac OS X.

To get a general overview on the four main hardware components and their relationship to the TRAMP subsystems you may refer to figure 2.19.

2 Proposed System

The software bus of the TRAMP system is completely based on DWARF, making extensive use of CORBA which is mapped to TCP/IP and the subsequent layers of the operating system. To achieve platform independency, Java is used as main programming language, which should allow easy migration to the client's SPOT hardware system in the future.

The target environment's hardware consists of the following parts:

- i. Wearable: Compaq iPAQ 3630 series, ARM Strong CPU (206MHz), 32 MB RAM, running Familiar Linux, allowing output/input through its 320 x 240 pixels built-in display
- ii. Laptop: Apple PowerBook, Motorola G4 processor, running Mac OS X
- iii. GPS receiver: Garmin eTrex Summit, connected to the iPAQ via PCMCIA. The GPS outputs location data in strings according to the NMEA 183 standard.
- iv. Head Mounted Display: Sony Glasstron, using a 1.55 million dot LCD to provide the user with see-through functionality, plugged into laptop via VGA output
- v. Wheelpointer (availability unsure), plugged into laptop, technical data not available yet
- vi. Inertial Tracker: Intersense Intertrax 2, connected to the laptop via its serial interface
- vii. Camera: iBot, plugged into laptop using FireWire

Parts of the TRAMP scenario require an external computer system as server for interactive electronic technical manuals (IETMs) and the expert help system. The exact specifications for the server have not been set. The server will need to support DWARF, database subsystems as well as the IETM database.

Figure 2.3 is a summary of the mapping between hardware components and machines/environment.

It should not be left unmentioned that the current hardware allocation renders the whole system useless for practical tasks, as no real mechanic could work on a car with two laptops mounted on his back. This limitation is currently implied by the lacking computational power of the available wearable computers and will vanish as soon as small portable computers with CPU power comparable to the two laptops are available. A visionary wearable computer has been designed in the early phase of the project and maybe one day it will become reality. Besides its sufficient CPU power and other resources this device includes the GPS receiver and has a small built-in printer for bills. Its extremely robust design makes it suitable for the conditions found in real garages and the additional input devices like the wheelpointer and a virtual keyboard, as well as the HMD are connected to it via high speed WaveLAN.

2.3.1 General System Performance

Due to the nature of augmented reality, TRAMP aims at real-time response times concerning the information displayed on the HMD.

For any network requests, TRAMP's aim is a maximum response time of one minute, not including the time to connect to the network. However, TRAMP can not guarantee network response time, due to circumstances beyond our control like heavy network traffic.

2 Proposed System

The general performance also depends on the implementation of the DWARF infrastructure, since all interaction between subsystems as well as between the system and the user is based on DWARF services.

2.3.2 Input/Output Performance

At first glance, the combination of iPAQ and laptop appears to have a considerable amount of resources. However, given the expected running times from the various subsystems, the potentially large amount of data transferred between the subsystems of the overall system, and taking into consideration that the system bandwidth may not always be stable, especially over wireless connections, real-time responses are quite possibly stretching the system's limits. To achieve that goal, I/O buffers and pre-processing are probably necessary for most, if not all of the subsystems. If these buffers are not provided in hardware, software buffers should be considered.

2.3.3 Processor and Memory Allocation

Image recognition and computing the output for the head mounted display are very CPU and memory intensive and require the fast CPUs of the laptops as the iPAQ's computational power is by far not sufficient for this task. Memory allocation is assumed to be unconstrained as the laptops can be equipped with as much RAM as is necessary.

The wearable computer's memory is limited to 32 MB, so the iPAQ will be used as additional input/output device only. 512 MB of RAM are expected to be enough for the laptop, however tests will have to be done on a real system to determine the exact memory allocation.

2.3.4 Connectivity and Network Infrastructure

The wearable computer is connected to the laptop over an 10Mbps wireless Ethernet network (WaveLAN) which is also used for communication between the laptop and the stationary server. Multiple wearable systems can connect to the server simultaneously. WaveLAN is also used for position tracking based on measuring the signal strength from various WaveLAN access points.

The connectivity between subsystems is done via DWARF based on CORBA, which is based upon TCP/IP. TCP/IP is a reliable protocol as long as the wireless connection remains active. A wireless connection can fail because of component failure, because the components move out of range or because extensive EMF interference consistently weakens or garbles the signal.

The anticipated bandwidth requirements should be analogous to video streams, so several hundred kb/sec would result in an acceptable user-system interaction performance. However, future plans to integrate speech recognition and text-to-speech conversion may require higher bandwidth or might delay the overall response.

2.4 Persistent Data Management

The persistent data will be stored (at least in the first build) in the file system. The used file types include

- WDL files (workflow description language)
- IETM's, HTML and media files (for UI)
- XML files, for example for the world model or for configuration files

Furthermore, the protocol data for a taskflow as well as the car history has to be stored. The file formats for those still have to be determined.

More detailed information about the persistent data can be found in the service-description "Persistent data storage" (2.2.2.4), which also provides a few methods to easily access those files.

2.5 Access control and security

Due to the demo purpose of the TRAMP project, the Access Control and Security issue is considered a low-level issue for TRAMP. Only a brief introduction will be given as followed:

A secure access control mechanism that prevents unauthorised or even malicious users from using the network must be incorporated. The access control mechanisms must operate fully transparent to any correspondent node in the network. End users of the access network should, apart from the initial authentication at login time, not notice any changes when using the network (i.e., standard applications must be fully supported).

Network security is a broad topic that can be addressed at the data link, or media, level (where packet snooping and encryption problems can occur), at the network, or protocol, layer (the point at which Internet Protocol (IP) packets and routing updates are controlled), and at the application layer (where, for example, host-level bugs become issues). As a result, IPv6 protocol stacks are required to include IPsec. IPsec allows authentication, encryption, and compression of IP traffic. Except for application-level protocols like SSL or SSH, all IP traffic between two nodes can be handled without adjusting any applications. The benefit of this is that all applications on a machine can benefit from encryption and authentication, and that policies can be set on a per-host (or even per-network) basis, not per application/service.

Due to the limitation of the current working environment, the IPv6 will not be used in the TRAMP project, rather in the future version.

2.6 Global software control

The following section will describe the software control implementation. Since TRAMP is using the DWARF Middleware and Service Manager, the TRAMP global software control won't differ much from DWARF global software control.

2.6.1 External control flow (between subsystems)

- Control flow is distributed within the TRAMP system. This means that there is no central control instance, but
- each service has its own control flow.
- Services request input ("needs"), wait for it and resume control when it arrives.
- There is a dispatcher ("Service Manager") coordinating communication between processes and balancing needs and abilities between single services.
- The services use asynchronous callbacks to communicate with each other.

2.6.2 Concurrent control

- **Service Manager**

The TRAMP service manager handles communication between single services and balances abilities and needs of the services. Failure of one service must not affect the service manager and the other services.

- **Multithreading**

The service manager uses threads, so that a large number of services is able to use the service manager simultaneously. The service manager also handles asynchronous events within the system.

- **Callbacks Between Subsystems**

The TRAMP service manager uses asynchronous callbacks for interprocess communication.

2.6.3 Internal control (within a single process)

- **Callbacks Between Subsystems**

Like the TRAMP service manager, the services use asynchronous callbacks for interprocess communication. Every single service uses internal methods to notify the other subsystems of its own status, its needs and abilities.

- **Avoiding Deadlocks**

Callbacks must not block the caller. So only status information is updated in the callee. All other internal work within a single service must be done by other worker threads.

- **Worker Threads for Each Service**

Each service has an own thread for communication, which communicates with the service manager and other services.

2.7 Boundary conditions

One of the main proposals of Boundary Conditions is to describe how the whole system can be started, what services have to be initialized first. Following figures show the system start. It is important to stress that Garage-Server has to be started first because Client (Mechanic-Service) depends on it and data provided by this server.

As one can see with help of the following figures there are at least two actors: Administrator who starts the Garage-Server and another administrator who starts the Mechanic-Service. After both servers have started the administrator of the Garage-Server has to test the communication to detect possible errors during the start process. For carrying out this test he/she has to ping (2.7) all existed system.

2 Proposed System

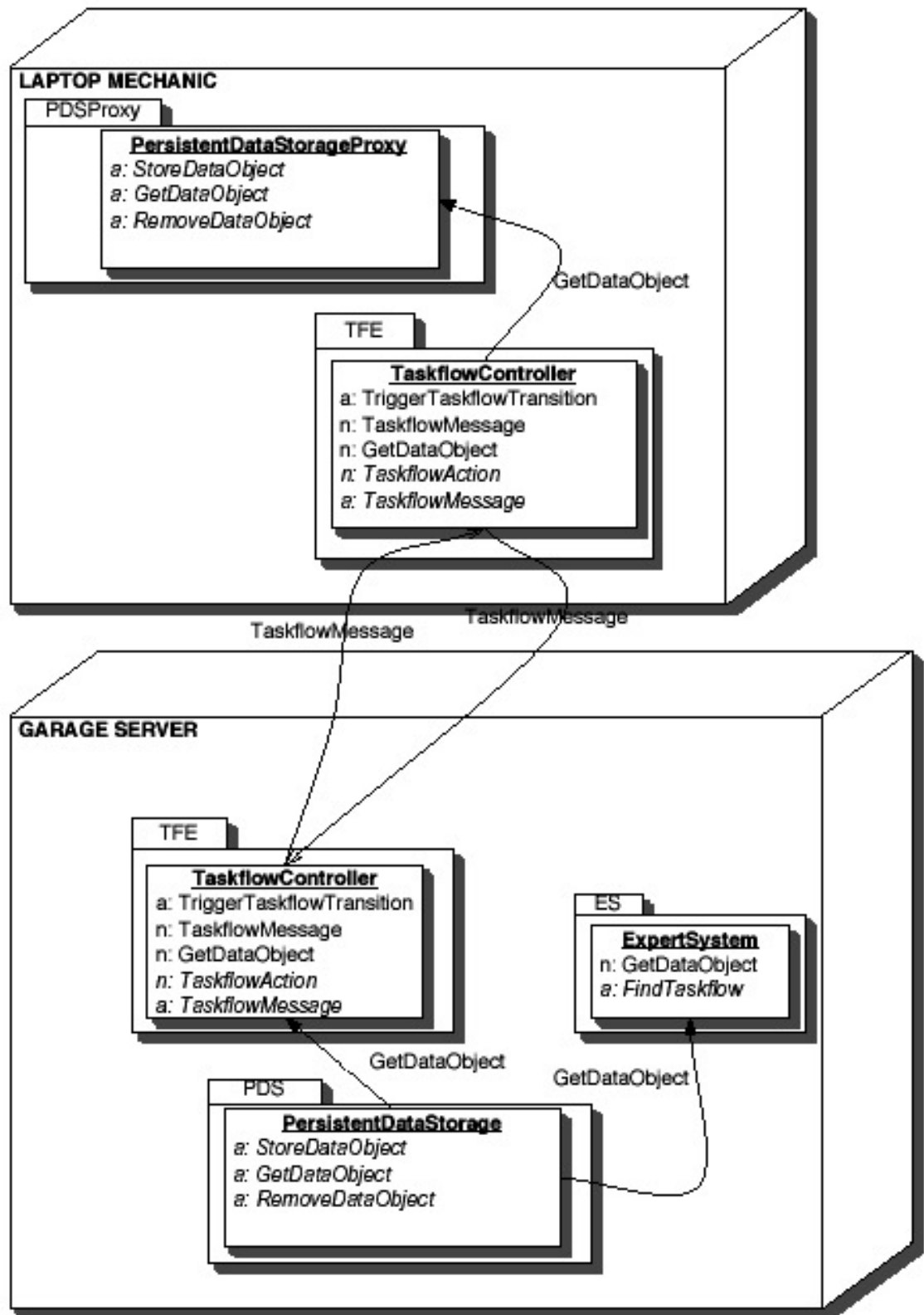


Figure 2.6: needs and abilities of the session subsystem

2 Proposed System

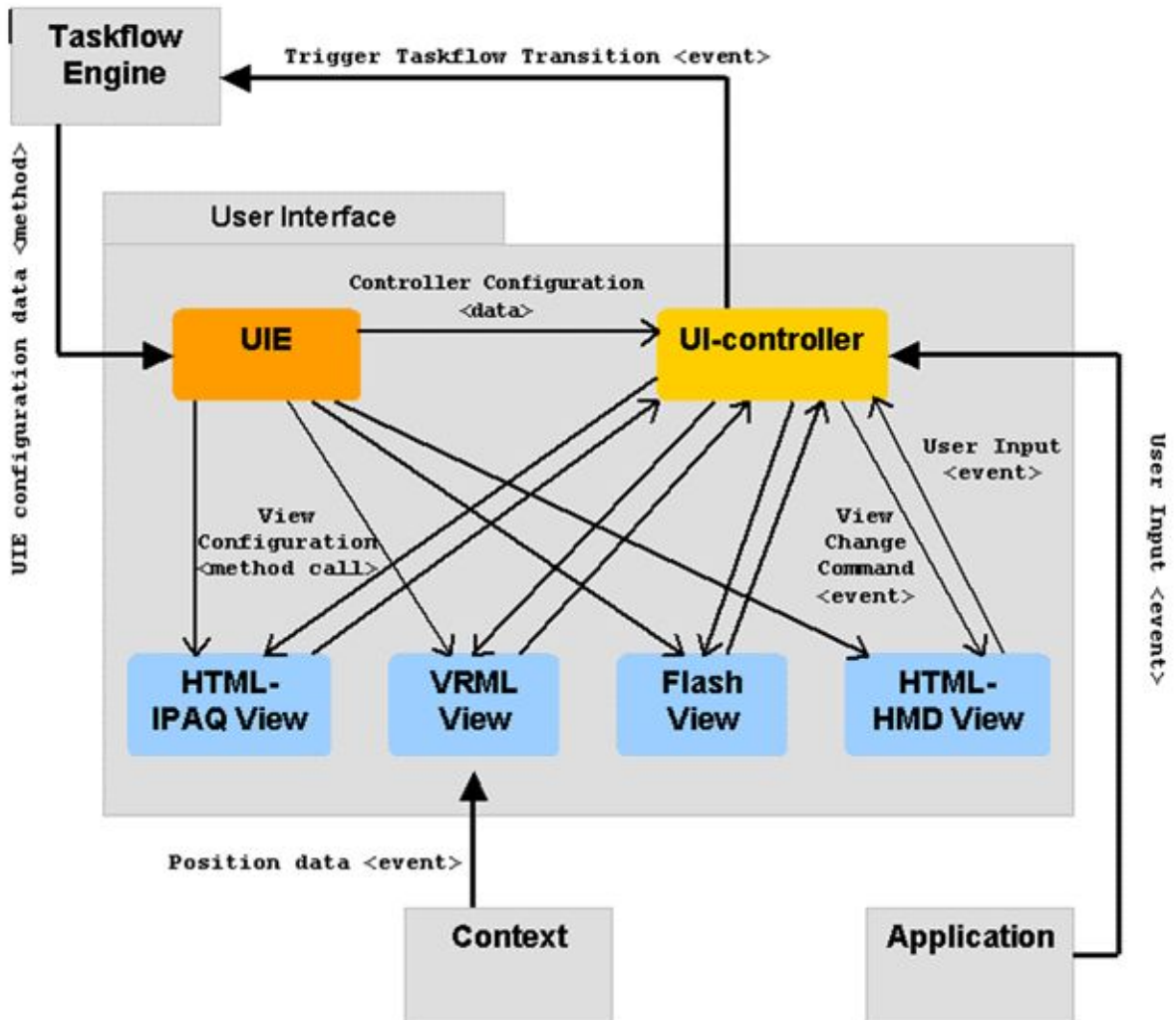


Figure 2.7: User Interface subsystem diagram

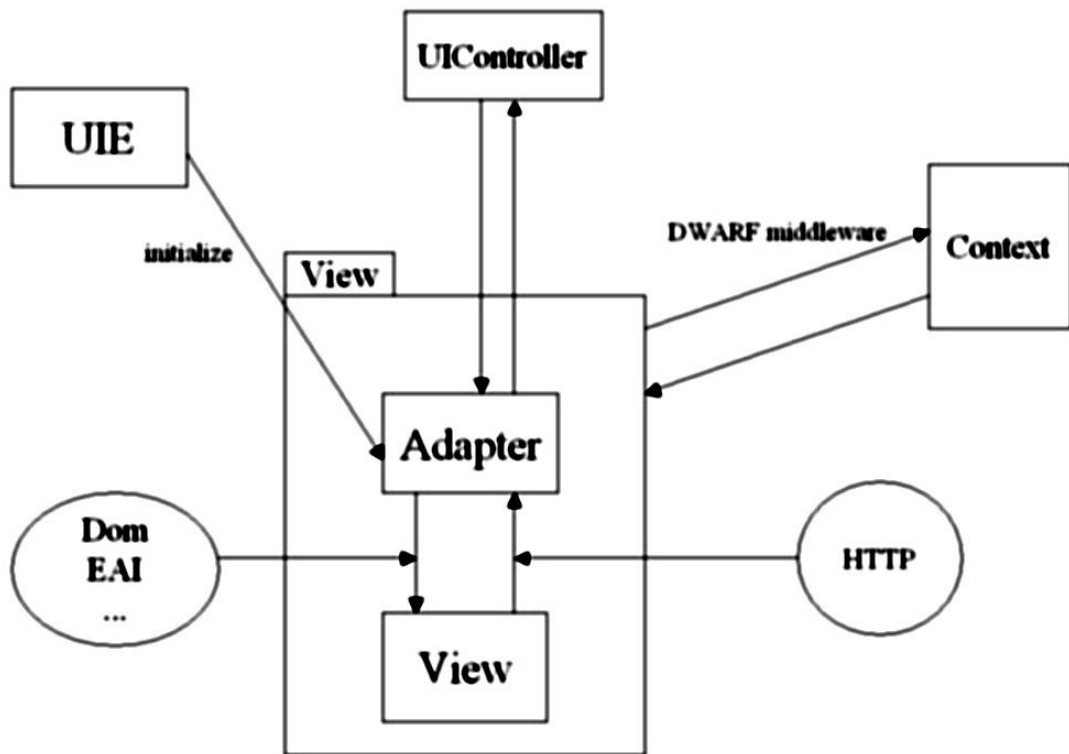


Figure 2.8: UI subsystem diagram: break-down to adapter-viewer level

2 Proposed System

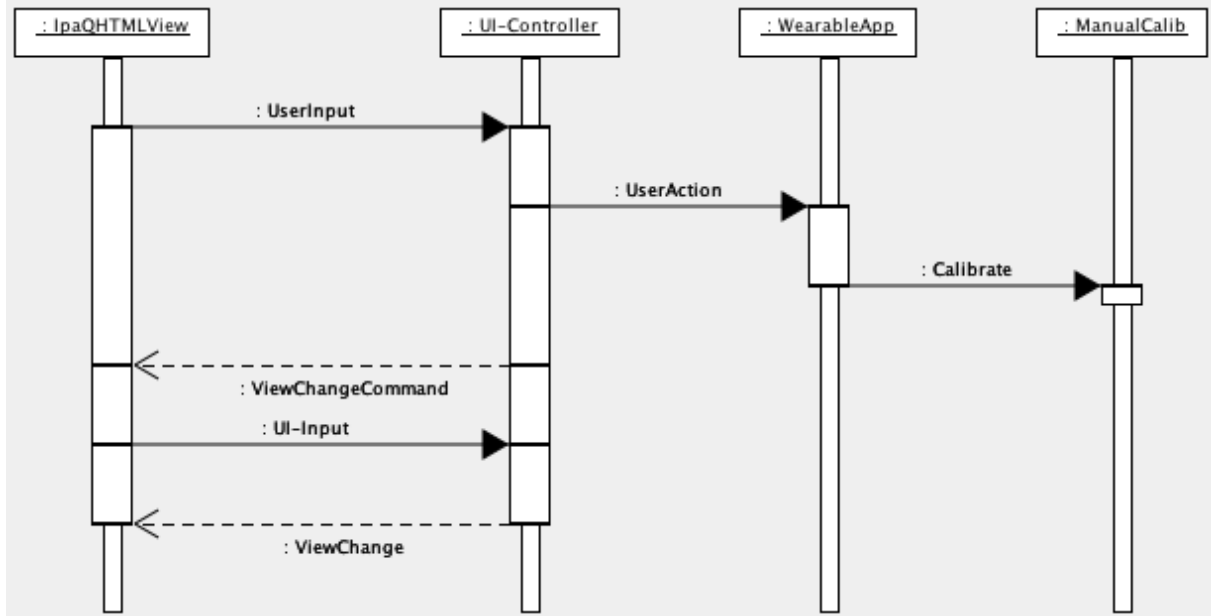


Figure 2.9: Calibrate System

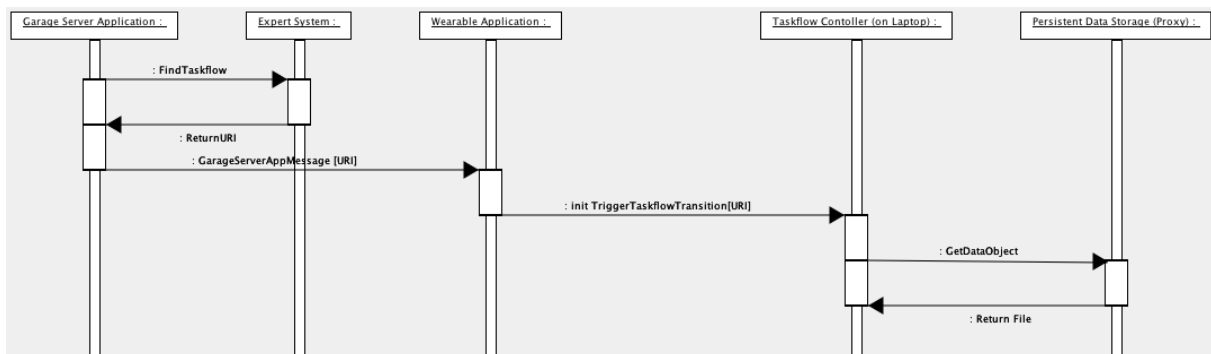


Figure 2.10: Handover Maintenance Task

2 Proposed System

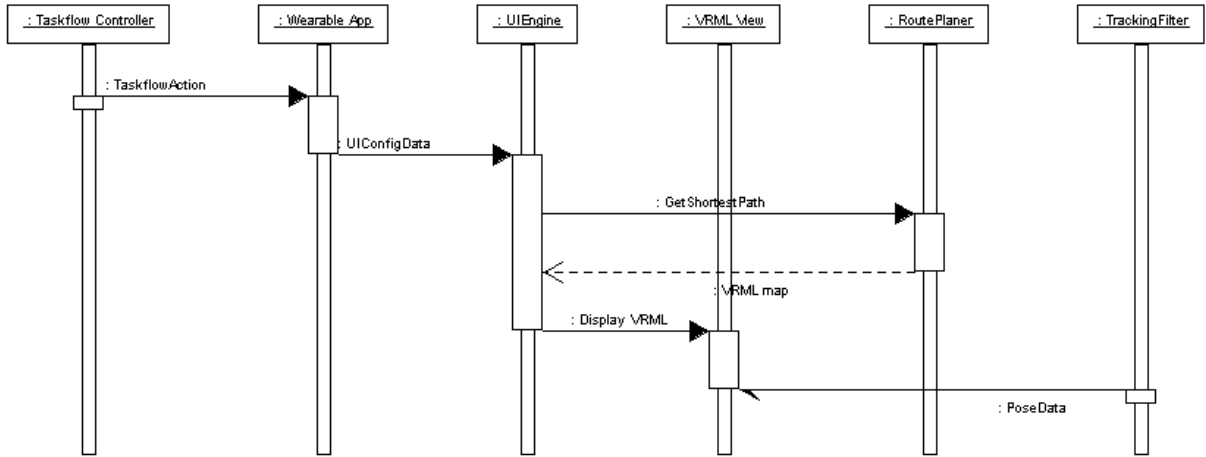


Figure 2.11: Navigation

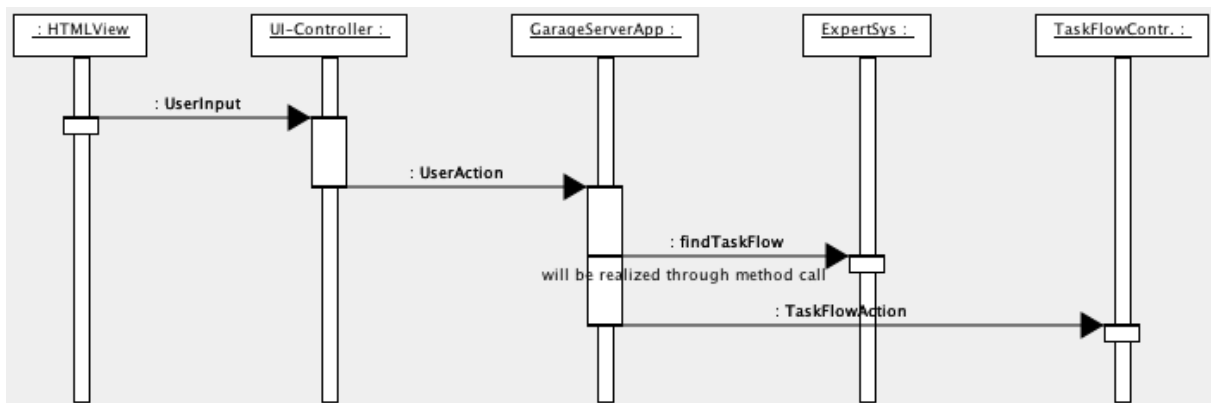


Figure 2.12: Request Maintenance

2 Proposed System

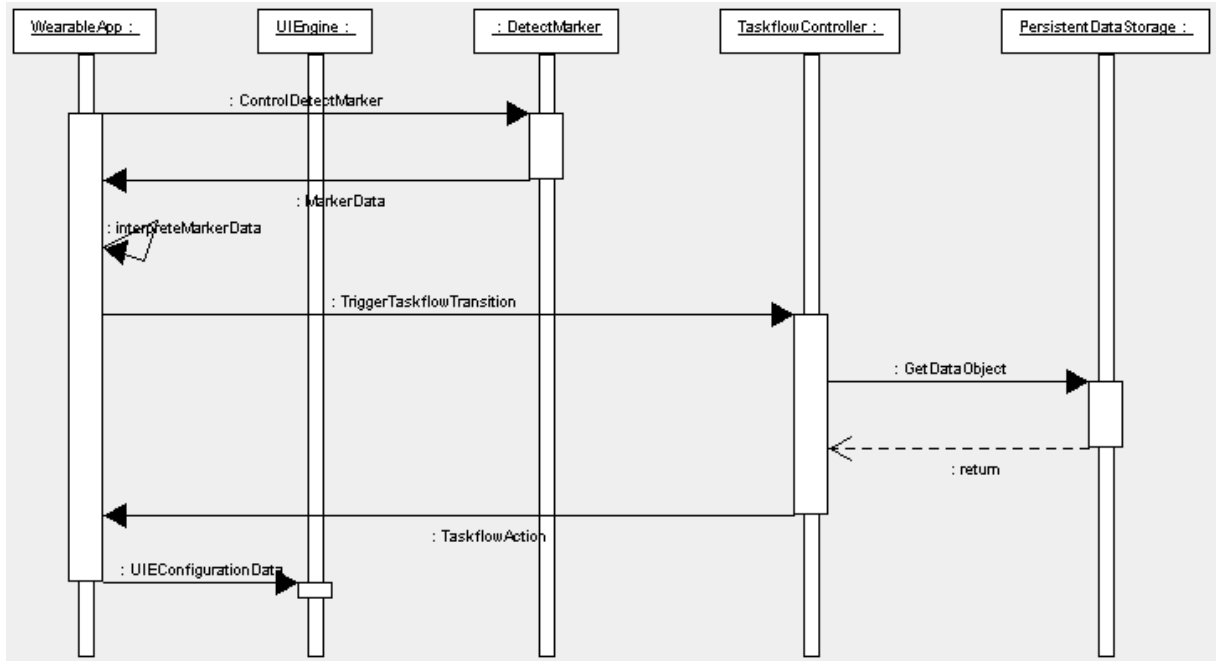


Figure 2.13: Detect Marker

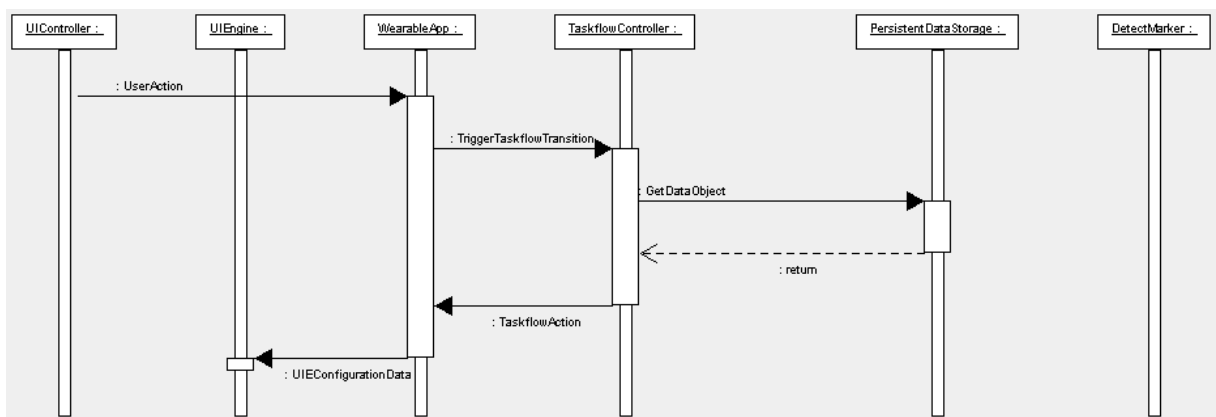


Figure 2.14: Taskflow Transition

2 Proposed System

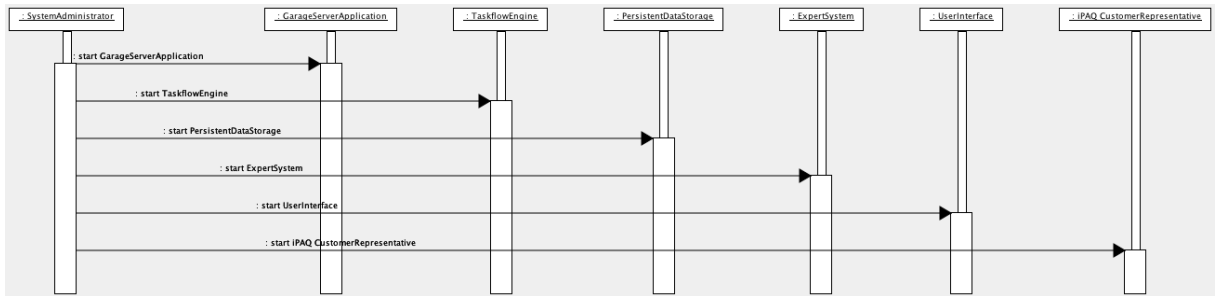


Figure 2.15: Start Garage Server Application

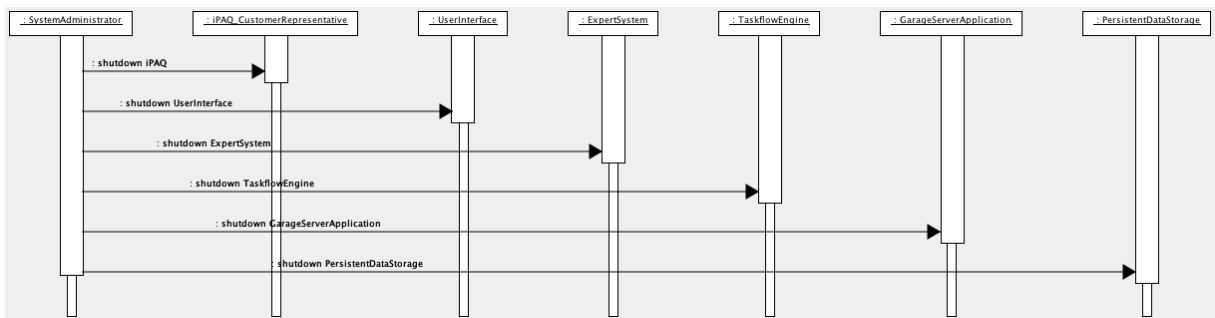


Figure 2.16: Shutdown Garage Server Application

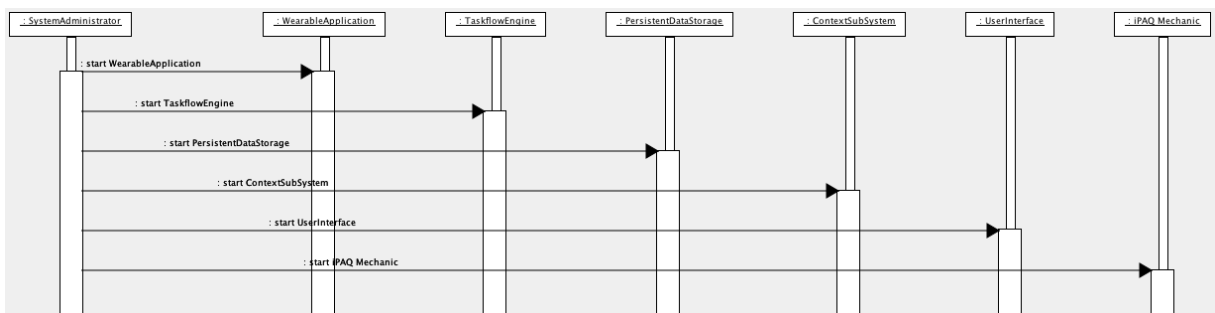


Figure 2.17: Start Latop Mechanic Application

2 Proposed System

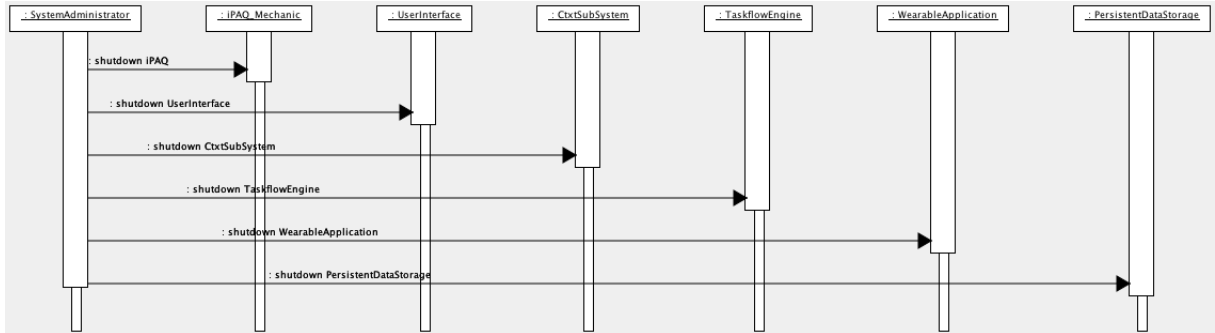


Figure 2.18: Shutdown Laptop Mechanic Application

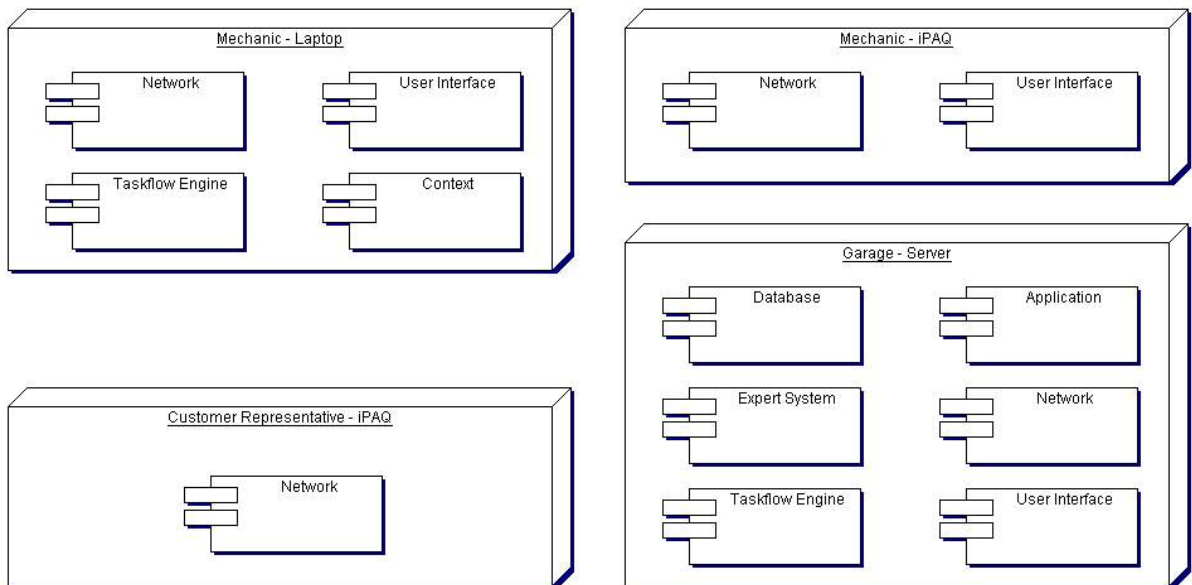


Figure 2.19: Allocation of TRAMP subsystems to hardware

Component	Interface	Operating System	Connected to
Compaq iPAQ	Serial, USB, PCMCIA	Linux	—
Laptop	Serial, USB, PCMCIA, FireWire	Mac OS X	—
GPS receiver	PCMCIA	Mac OS X	laptop
HMD	VGA	Mac OS X	laptop
Wheelpointer	PS2	Mac OS X	laptop
Inertial Tracker	Serial	Mac OS X	laptop
Camera	FireWire	Mac OS X	laptop

Figure 2.20: Interfaces and Operating Systems used by TRAMP hardware

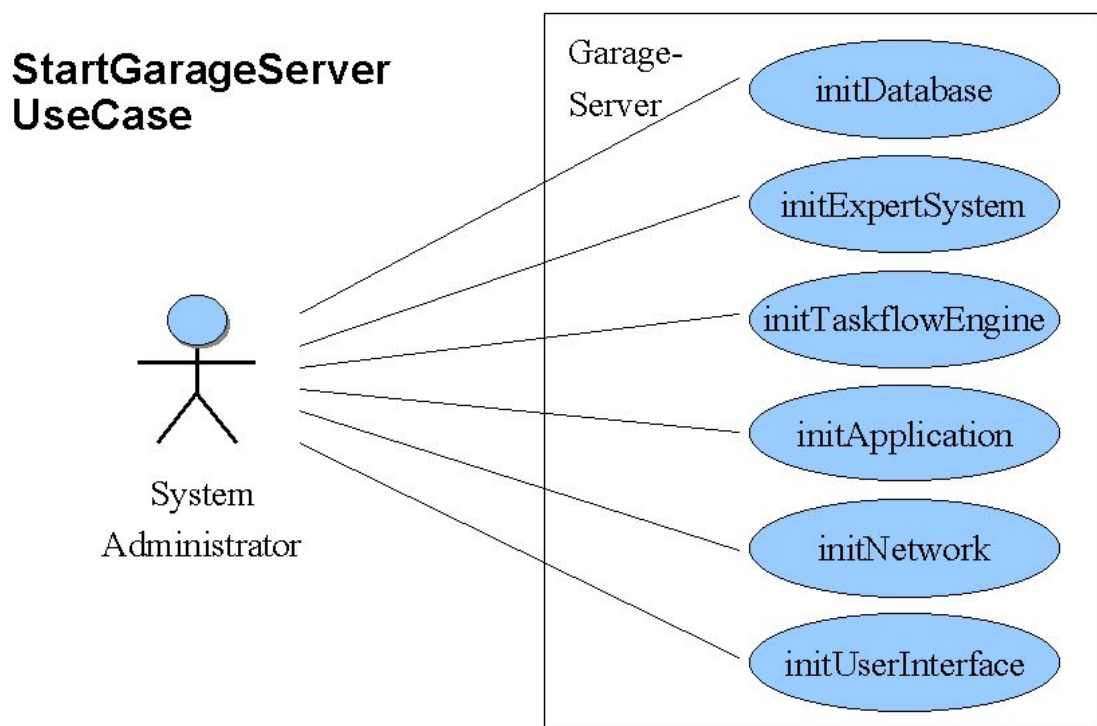


Figure 2.21: Start Garage-Server Use Case

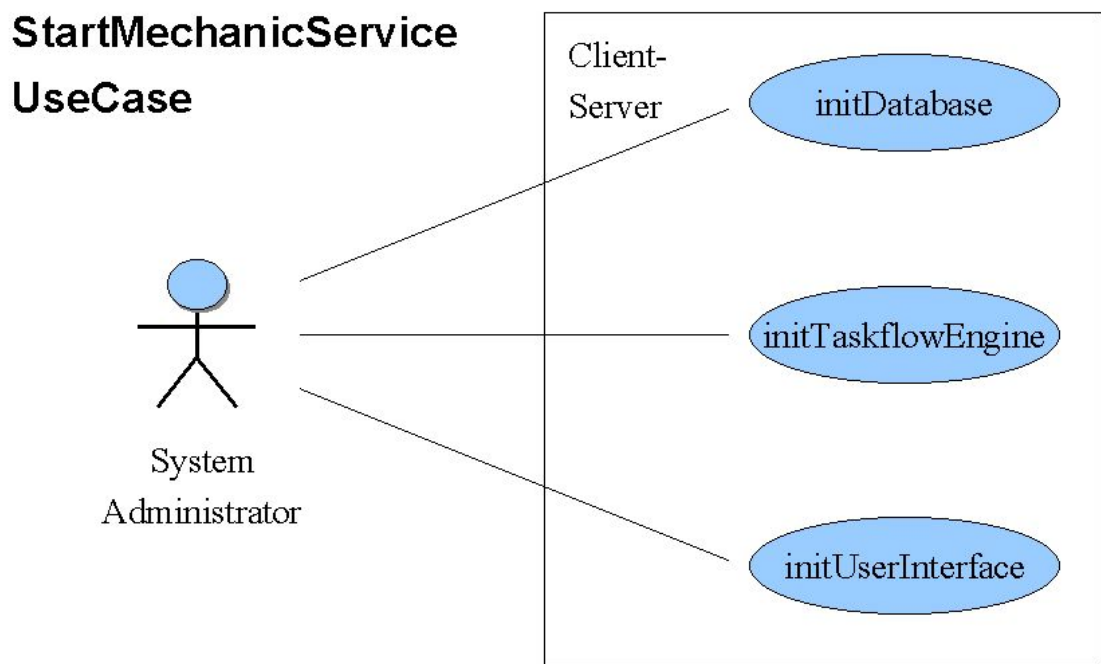


Figure 2.22: Start Mechanic-Service Use Case

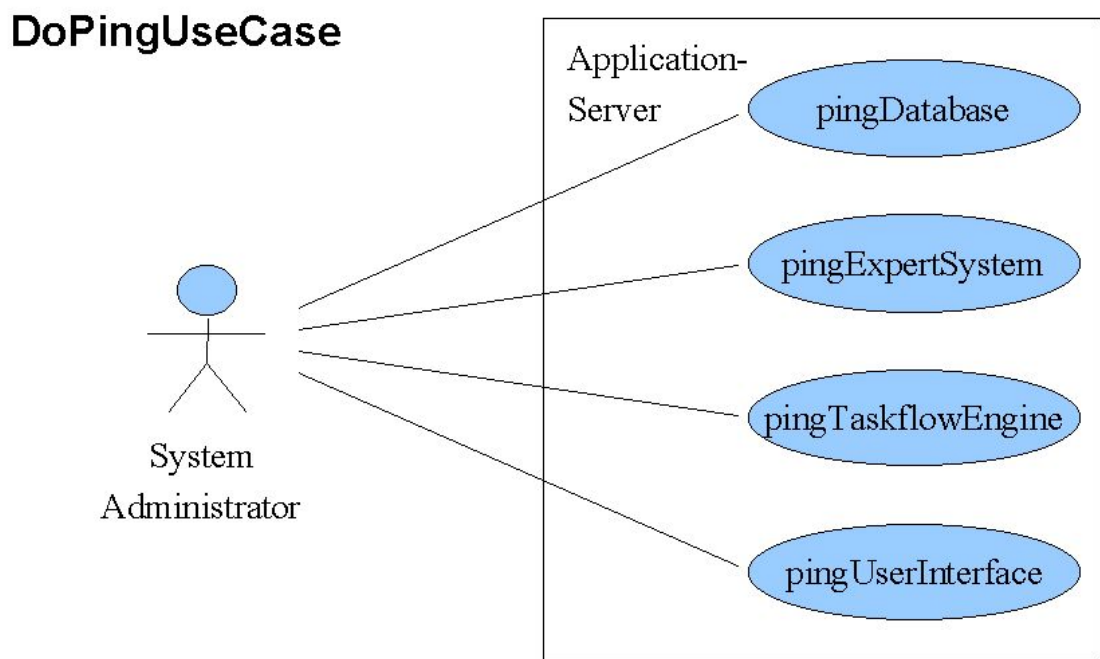


Figure 2.23: Do Ping Use Case

A Glossary

Actor

external entity that needs to exchange information with the system; an actor can represent either a user role or another system

Analysis

model of the desired system containing: object model, dynamic model

Domain Constraints

describe user visible domain facts that are not directly related with the functionality of the system (nonfunctional requirements)

Dynamic Model

sequence diagrams - describe interactions between the components of a system

Examples

illustrate use of the system with scenarios - instance of a use case

Glossary

see glossary :-)

GPS

Global Positioning System

HMD

Head Mounted Display

Index

alphabetically list of all requirement elements for accessing specific elements

IETM

interactive electronic technical manual

Object Model

describes structure of a system in terms of objects, attributes and associations

ODD

Object Design Document

Problem Statement

brief introduction to the problem including purpose and scope of the desired system

Quality Constraints

nonfunctional requirements

RAD

Requirements Analysis Document

Requirements

define the problem domain as relevant to the system including following elements: Actors, User Tasks, Domain Constraints, Quality Constraints on User Tasks

SDD

System Design Document

Services

A Glossary

functionalities that must be provided by the system (=functional requirements)

Specification

defines the functionality provided by the system including: Use Cases, Services, Global Functional Constraints, Quality Constraints on Use Cases, Quality Constraints on Services

Subsystem

division of the system into subsystems

Subsystem Services

functionalities that must be provided by the subsystem

TRAMP

Travelling Repair and Maintenance Platform

Use Cases

a general sequence of interactions between one or more actors and the system

User Tasks

typical tasks supported by the system

WDL

The Workflow Description Language describes tasks in XML syntax. Tasks contain links to successor tasks and referenced external documents.

Bibliography

- [1] *Proceedings of the IEEE and ACM International Symposium on Augmented Reality – ISAR 2000*, Munich, 2000.
- [2] M. BAUER, A. MACWILLIAMS, F. MICHAHELLES, C. SANDOR, S. RISS, M. WAGNER, B. ZAUN, C. VILSMEIER, T. REICHER, B. BRÜGGE, and G. KLINKER, *DWARF: Requirements Analysis Document*. Unpublished, 2000.
- [3] B. BRÜGGE and A. H. DUTOIT, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.
- [4] Y. CHO, J. LEE, and U. NEUMANN, *A Multi-Ring Color Fiducial System and a Rule-Based Detection Method for Scalable Fiducial-Tracking Augmented Reality*, 1st International Workshop on Augmented Reality (IWAR), (1998).
- [5] T. JEBARA, B. SCHIELE, N. OLIVER, and A. PENTLAND, *DyPERS: Dynamic Personal Enhanced Reality System*, Tech. Rep. 463, MIT Media Laboratory, Cambridge, MA, 1997.
- [6] H. KATO, M. BILLINGHURST, R. BLANDING, and R. MAY, *ARToolKit PC version 2.11*, December 1999. available at http://www.hitl.washington.edu/research/shared_space/download.
- [7] A. MACWILLIAMS, *Using Ad-Hoc Services for Mobile Augmented Reality Systems*, Master's thesis, Technische Universität München, 2001.
- [8] W. H. PRESS et al., *Numerical Recipes in C: the art of scientific computing*, Cambridge University Press, 2nd ed., 1992.
- [9] S. RISS, *An XML based Task Flow Description Language for Augmented Reality Applications*, Master's thesis, Technische Universität München, 2001.
- [10] J. SCHMIDT, *Aufnahmegeometrie*, in *Erweiterte Realität: Bildbasierte Modellierung und Tragbare Computer*, B. Brügge, H. Niemann, and T. Reicher, eds., Technische Universität München, October 1999, pp. 152 – 174.
- [11] J. VALLINO, *Augmenting Reality with Minimal Calibration*. <http://www.cs.rit.edu/~jrv/research/ar/thesis.html>.
- [12] M. WAGNER, *Design, prototypical implementation and testing of a real-time optical feature tracker*, Master's thesis, Technische Universität München, 2001.

Bibliography

- [13] R. WHITAKER, C. CRAMPTON, D. BREEN, M. TUCERYAN, and E. ROSE, *Object Calibration for Augmented Reality*, tech. rep., ECRC, 1995.
- [14] A. ZISSERMAN, A. W. FITZGIBBON, and G. CROSS, *VHS to VRML: 3D Graphical Models from Video Sequences*, in *Confluence of Computer Vision and Computer Graphics*, NATO ARW, August 1999.