Application Architecture and System Technology for LDP (Logistical Data Platform) and FASTCORS (Fast Transport of Containers outside Regular Services)

by

Henner Krabbe, Andreas Matheja, Claus Zimmermann

Contents

List of Figures

Abbreviations

ACE	Adaptive Communication Environment
AIM	Application Interconnectivity Manager
AIS	Automatic Identification System
ALSO-IRIS	ALSO Intermodal Routing and Information System
API	Application Programming Interface
ARA-Ports	Ports of Amsterdam-Rotterdam-Antwerp
BICS	Barge Information and Communication System
CoDaBa	Communication Database (component of AIM)
COPIT	Computers for Inland Navigation within Integrated Transport Chains
CORBA	Common Object Request Broker Architecture
COS	CORBA Object Service
CSL.DB	Common Source Logistics Database
DCS	Danube Combined Services Transportgesellschaft m.b.H.
DOM	Document Object Model
DoRIS	Donau River Information System
EAI	Enterprise Application Integration
EC	European Commission
ECDIS	Electronic Chart Display Information System
EDI	Electronic Data Interchange
EDIFACT	Electronic Data Interchange for Administration, Commerce and Transport
EDP	Electronic Document Processing
ELWIS	Electronic Water Information System
ETNA	European Transport Network Application
FASTCORS	Fast Transport of Containers outside Regular Services
GIS	Geographical Information System
GLONASS	Globalnaya Navigatsionnay Sputnikovaya Sistema
GPS	Global Positioning System
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HA	High Availability
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure Sockets
IFTDGN	International Forwarding and Transport Dangerous Goods Notification
ILL	Industrie-Logistik-Linz GmbH

IMO	International Maritime Organisation
INDRIS	Inland Navigation Demonstrator for River Information Services
IP	Internet Protocol
ISL	Institute of Shipping Economics and Logistics
ITT	Inter Terminal Traffic
IVS'90	Reporting and Information System in the Netherlands
JDK	Java Development Toolkit
JMS	Java Message Service
JNDI	Java Naming and Directory Service
JVM	Java Virtual Machine
LDP	Logistical Data Platform
LOMAX	Lock Management System
LSP	Logistic Service Provider
MeGa	Message Gateway
MIB	Reporting and Information System in Germany
MOM	Message-oriented Middleware
MOVES	Vessel Information System for the River Mosel (Moselverkehrserfassungssystem)
ODBC	Open Database Connectivity
OSIS	Open System
PC Navigo	Voyage Planner for Inland Navigation
QoS	Quality of Service
RDBMS	Relational Database Management System
RIS	River Information Services
SMS	Short Message Service
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
STI	Strategic Traffic Image
SWP	Sub Work Package
TEU	Twenty-Foot Equivalent Unit
TTI	Tactical Traffic Image
UML	Unified Modelling Language
URL	Uniform Resource Locator (World Wide Web Address)
VHF	Very High Frequency
VTS	Vessel Traffic Services

Summary

The Logistical Data Platform (LDP) was defined and developed in Work Package 5 of the European Joint Research Project COMPRIS (Consortium Operational

LDP delivers key elements of application architecture and system technology qualified for design, as defined by COMPRIS work package 5.

Herein, LDP is envisioned as a large-scale electronic business platform and information broker system for the logistics domain.

In order to provide the logistic user community with realtime/neartime service offerings, the platform must be continuously fuelled with data from RIS and other transport modes. Therefore, RIS systems have to communicate with open standards and implement supplier/consumer units to communicate with LDP. Legal aspects (e.g. dispatching of information to dedicated users) have to be fixed beforehand.

Therefore, the connectivity between LDP and RIS is treated as a contractual obligation at system level. This particular relationship is addressed by means of a messaging backbone architecture.

The application framework of LDP is, in itself, based on a model-driven approach since there is a variety of Use Cases to expect in the future. A generic process model defines the application layout and the platform-controlled elements that constitute the LDP core system.

In this framework, business process types denote sets of activities to be executed by the user (or the platform) in order to achieve a distinct business goal. The business processes, which in turn are runtime instances created from the respective business process types defined at buildtime, are hosted autonomously and on behalf of the user in a stateful, multi-threaded application server environment. This runtime environment can apply dedicated policies to take full control of server-side resources, e.g. by process lifecycle management. It will allow process execution while being disconnected, context sharing among processes, and asynchronous messaging, thereby providing non-blocking client/server communication.

The LDP client technology is designed to be stateful, asynchronous, and multi-channeled, thereby enabling clients to forward and/or distribute incoming messages while sending requests at the same time, for instance the user in synchronous interaction with one of his business processes.

Accordingly, and to draw a parallel, the technologies and paradigms that qualify for LDP can be compared to those of realtime market data and trading systems in the financial world, e.g. stocks exchange, derivative trade, etc..

Apart from information brokerage, the business processes to be hosted by LDP will be far more complex, especially with regard to planning and intermodal contracting scenarios. Therefore, the approach merges selected features of distributed business process systems with the capabilities of realtime/neartime information systems into a new class of responsive e-business systems.

The methodologies and techniques to apply over the course of LDP development will cover business process engineering, object-oriented analysis, design, and programming of concurrent event-based systems, the architecture and object services standards of message-oriented middleware (MOM) and either Common Object Request Broker Architecture (CORBA) or Java 2 Enterprise Edition (J2EE), depending on further decision making dedicated to the technology of the LDP core system.

Functionality, robustness and applicability of LDP were demonstrated for a first use case - a software application to establish fast transport of containers outside regular services (FASTCORS). Therefore, a consumer and a supplier unit were incorporated in a port information system sending containers to be transported to the market place. This market place, by means of LDP, dispatches incoming offers and sends them to dedicated users abroad. This was demonstrated by a LDP Ship Client, where the

skipper can pick containers from a list of offers (offered by LDP), showing all his information (weight, price, dimensions etc.) to the interested skipper. For acceptation, cancellation and updating of an offer, different events were implemented. Stability and applicability of the system were demonstrated on work package meetings and stakeholder sessions.

During LDP development and FASTCORS implementation no problems using CORBA standard (TAO implementation on top of ACE) or other components recommended for LDP development could be anticipated. The integration of existing applications to LDP environment means implementation of supplier/consumer units using CORBA libraries and runtime environment (open source). This task can be calculated as a two day job for an experienced Java or C++ programmer using the examples shown in this report.

Implementation of business processes on top of LDP is already possible and was demonstrated by FASTCORS.

In order to facilitate further steps towards practical software development, mandatory work items had to be derived - and have to be - from the current COMPRIS material and future projects. These items consider requirements, business abstractions, and the **Use Cases** in particular.

From an IT perspective, especially in terms of business systems engineering, future projects must perform a shift from the technical views to the business process view, hence focus on the actual business and information procurement goals, how to achieve them by means of processes, and how to further optimize these processes in the light of competitive capacity.

1 Introduction

1.1 Motivation

Inland waterway transport in Europe transport corridors is mainly used in port-to-port relations for bulk and regular (container) services on the rivers Rhine, Danube and Elbe.

To promote and use the strategic advantages of inland waterway transport in terms of loading capacity, safety, security and environmental advantages, the competitiveness shall be increased against land - based transport modes. This requires an aggressive expansion of actual available waterborne transport services for containers and Ro-Ro cargo for medium and long distances as well as a faster transport where transport and cargo information must be available to the end users, planning the “just-in-time” supply chain of their production.

As a logical consequence improving the quality of inland waterway transport services in terms of reliability, better route planning, tracking and tracing, online fleet management and ship monitoring is desirable.

Information flow of transport data for inland waterway transport is often diffuse, responsibilities are not clearly defined and data flow schemes differ from country and location. Mostly it has to be seen mainly as a disjunctive data housing of several, independent institutions.

Dispatching of data is not regulated so far. Even within former projects, reliability for data dispatching was not defined and Quality of Service (QoS) was not guaranteed - not in the security of the transmission (acknowledgements, cryptographic techniques) nor even in the completeness of necessary and/or wanted data.

Thus, the first step in producing “Value Added Services from RIS for Transport” is to define, implement and demonstrate an open platform with added value functionality – the “Logistical Data Platform” (LDP) – receiving, processing and dispatching RIS data and results/answers of initiated business processes related and based on this data.

1.2 Objectives

LDP should be a system capable to interact and communicate with different RIS systems, to interchange data, share resources and transfer results of its own process repository.

The user should be able to easily log-on to the platform by single-sign-on from their LDP Client to modify his user profile, download results and context dependent information or easily get in business with other parties.

Business processes of the user must be worked through, although he is logged-off from the platform. Results should be sent, if demanded, automatically to the user on-the-fly, after a specified event or as a regular service.

Focus of the platform is the logical combination of data with available RIS information and transport information.

Thus, the platform should give additional valued information generated out of RIS to the transport area.

LDP should not be compared with other initiatives, which can be described as database warehouse applications, where functionality is restricted to dispatching. The platform will use distributed reliable data storage systems, it is open to every user and not dependent on a special system, uses high security communication and data housing and provides generated added value information in addition to the collection of provided data.

Thus, the collection, processing and distribution of RIS information to the transport area with Added Value Information set-upon their business goals, objectives and requirements is the essential goal of this work package.

To stimulate waterborne transport within inter modal transport chains, Added Value Services from RIS will be offered by:

- Conception and development of an open interface for RIS and European transport information and supporting services for transport management through the introduction of a *Logistic Data Platform for RIS* ***(interconnectivity and dispatching).***
- Provide enhanced Value Added Information from RIS (speed, free loading capacity, stowage plan, traffic scenarios, estimated time of arrival, route planning, lock management and planning etc.), by processing RIS data, combining it with logistical data and initiating business processing ***(business processing).***
- Provide an environment (*FASTCORS – Fast Transport of Containers outside Regular Services*) using LDP on top of RIS information for individual and reliable transport of containers outside regular services. This platform will be used by single skippers, shipping companies, fleet managers, terminals, agents and end users ***(first use case for LDP).***

1.3 Accomplishment of the Project

This project was part of the research initiative of the European Community represented by the Commission of the European Communities within this 5th Framework Program (1998 - 2002) "Competitive and Sustainable Grouth Program under the Directorate General for Energy and Transport Contract No. GRD2/2000/30161

Consortium Operational Management

Platform River Information Services

COMPRIS

FRANZIUS-INSTITUT for Hydraulic, Waterways and Coastal Engineering of the Leibniz University of Hannover (Germany) was one of 44 partners in this consortium. Responsibility was for the following research activities (Work Package) between 2002 and 2004.

Work Package 5:

Value-added services from River Information System RIS for Transportation Information Subwork Package 5.3. Development and Implementation of a "Logistical Data Platform (LDP)" and the "Tranport of Containers Outside Regular Services (FASTCORS)".

Financial support form the European Community and technical and administrational support and cooperation from and between the numerous partners of the consortium is strongly acknowledged.

2 Requirements and Architectural Approach for LDP

Decision making in application and system architecture is based on a requirements foundation usually given by a Software Requirements Specification (SRS). Only a subset of requirements for LDP was available before system development started.

That is why in the following the architectural elements developed into a blueprint should be considered as a suggestion, a recommendation for further refinement - deduced from the available material, the questionnaire on user needs and enriched with the experience from the development of software systems that are comparable to LDP in terms of business process goals, scale, complexity class, and application model.

2.1 Application Context and Complexity Class

The goal was to develop a Logistical Data Platform (LDP), where logistical players can retrieve information for transport planning and the follow-up of an on-going transport, such as positioning of their goods.

The introduction to WP#5 gives a discussion on the business goals to be addressed by LDP. In process terminology this can be summarized as follows:

- Innovative ad-hoc transport services and business offerings,
- Process optimizations within and across participating business domains,
- Ad-hoc concatenation of business processes (intermodal chaining) and
- Extensive software system support for process execution and information processing.

Consequence:

In order to support processes by means of software systems, a *process view* has to be established in analysis and design. Technical and business-related processes need to be formally conceived before the transition to software is initiated. This precondition was not fully achieved.

Two fundamental aspects have been recognized already:

The integration of disjoint databases and applications will not help in order to achieve the process goals. By low-level integration, the completeness and the availability of data (that form the process resources) in terms of functional needs and application context cannot be guaranteed. Currently, the focus is on integration and harmonization, not on interoperability. Whether feasible or not, integration will definitely not along-produce the type definitions of business processes. These process types are mandatory: They define the characteristics and the information processing demands of the real business, hence influence the IT approach and provide starting points for gradual business optimisation.

There are specific needs for typical middleware services: Resource naming and trading, directory services to administer access control and security at organizational and technical level, and transactions in a global multi-user environment. A significant contribution to LDP will be the appropriate application architecture model on the one hand, and the re-use of middleware services from common standards on the other. A re-invention of services should be avoided.

Consequence:

At meta level, specify and agree on a common application architecture model for LDP. The application model should follow the process view and anticipate the transition of real business processes into a software representation. It should be technology-independent. Later on, a dedicated technology mapping will define the buildtime and the runtime environment for the proposed platform.

Hence it follows that LDP will be developed as an application family according to a given application architecture (its blueprint developed in chapter 3) and also as a dedicated mapping of the architectural model into a technology environment.

Consequence:

Narrow the search for middleware services to industry-proven standards, e.g. J2EE, CORBA, LDAP, etc. and eventually carefully selected XML technologies. Besides feasibility and the total frame of requirements, a large-scale software project should always consider aspects like

- maturity, long-term availability, and pervasiveness of technology and products in industry and academia,
- the freedom to operate LDP on different types of platforms, and especially on the leading environments for application server systems: UNIX and OS/390,
- the opportunity to use build-in hooks for system management instead of finally being forced to write the full API toolset required for system monitoring and
- the man power and skills required to develop and maintain the platform.

Furthermore, there is a vision and also a paradigm given for LDP: Users submit their requests to the platform and receive value-added information on-the-fly. Here, the platform acts as an information broker with an open interface but also as an intelligent user agent, capable of inherent information pooling. Besides the fact, that retrieval Use Cases are typical candidates for high volume/high frequency processes and that the (imaginary) agent's functional spectrum is well described by means of processes and their outcome respectively, the paradigm raises aspects with special concern:

(1) How in general is data described at meta level and across domains? What is the syntax and what are the semantics of a query language, if there is one at all? How is data processed for further use at application level, and especially with regard to the outcome of processes? Is there a system of common business objects (types) available in order to allow the definition of strongly-typed interfaces - in terms of interface signatures ? These questions directly affect the layout and the semantics of the open interface proposed for LDP.

(2) From a logical perspective, and taking the process/agent metaphor into account again, can the open interface proposed for LDP be seen as a common user portal instead? A portal that gives access to a context-aware environment, providing the process offering as a sort of dynamic catalogue, while LDP is the runtime and controlling environment for process execution - a process engine.

Further Conclusions:

- At present, a sufficient decision on type handling cannot be made. Even the questionnaire did not reveal if the object-oriented approach is practicable even at the level of the RIS/LDP transform.
- There is always a trade-off between strongly-typed and weakly-typed interfaces in application design. Experience shows that a component-oriented approach with a dedicated object-oriented type space is most favourable (strongly-typed). The information flow of processes and the application components will then share a common type set where each base type has a concise state, unique information1 and behaviour - the object's responsibilities. Below application component interfaces (dedicaded facades), and within specialized subprocesses, techniques like interface and implementation inheritance may be used to extend these types

[1] Primary keys, UUID in terms of its business origin/location, object lifecycle control, etc.

in a domain - or system-specific manner. By composition of subprocesses, the embedding and chaining of processes at dedicated nodes, cross-domain dependencies can be limited. Interfaces with high-value semantics (weakly-typed) are generic and will almost never change. At first sight, this is a benefit. Generic interfaces are applied when messages are passed around and interpreted in one turn, like in EDIfact. The problem with the approach is that there will never be a clear separation of concerns among application components. Messages have to be interpreted at any time throughout the process. It is also impossible to forward context onto entities referenced in message documents (e.g. EDIFACT documents), for instance transaction context. When referential integrity of resources becomes an issue for the success of the processes, the document messaging approach will not be sufficient.

- To be actively monitored by LDP in the context of the total offerings possible, RIS data should not enter the domain of LDP ("the added-value domain") in its native form. Logically, it should be pre-processed outside and introduced into LDP at object level2. The set of incoming aggregates (entities/objects in value semantics) will belong to a dedicated type space known by processes. The LDP process environment will act as an observer of this set3 and adjust its offerings dynamically.
- Near realtime information from RIS will provide activation data, startup events, boundary conditions, etc. for the processes offered by LDP. On the contrary: If basic decisive data is fed in from remote databases and has not already been evaluated until the actual execution of a process, the platform will not show the responsiveness and the ad-hoc capabilities desired.

Finally, the platform promises to be open and secure for every user, and to follow open standards. It will offer a certain degree of permanent RIS information that is maintained automatically and presented to the user - and to an inherent machine-hosted process runtime environment.

The latter is the interpretation of FRANZIUS-INSTITUT how to connect dynamic RIS information with the business processes and their activities: At event level, at any time, but decoupled from RIS at component level. The process environment to be designed will offer startable activities according to the current application context, and the actual data situation provided by RIS. **If disconnected operation is supported by means of an agent acting on behalf of the user (background/batch activities, etc.), the environment would also provide a user-centric view of processes still running or already completed.**

- The understanding of processes in LDP is more general than for instance in workflow applications. Some aspects allow a clear demarcation though: Although the modeling of users, their roles, and organizational hierarchies will play a significant role, especially with regard to access control, there are currently no indications for typical workflow features like delegation, suspend and resume, etc[4]. Instead, dynamic elements are prevailing: Fast reflection of changes in offerings, ad-hoc processes, and business transactions. This holds especially for the challenging objectives
  (a) to provide enriched information from RIS,
  (b) to establish an environment for FASTCORS (Fast Transport of Containers outside Regular Services) and

---

2 By a separate component, not necessarily in physical separation from LDP or RIS.

3 Actually, there are collections of objects, e.g. ships, trucks, loads, etc.

4 For a discussion on workflow system features see e.g. (Leymann & Roller, 2000)

(c) to realize transport planning processes.

In the model-based view, FASTCORS and other ad-hoc businesses can be regarded as top-level and mission-critical Use Cases to be realized by means of LDP. Planning processes are the most complex and demanding in terms of data availability, business rules, and likelihood estimates, especially when applied to questions of optimization.

Intermodal transport planning, and especially the respective contracting scenarios, will be a challenge for business process analysis and IT design. Very likely, new types of processes, business rules, and offerings will emerge in order to allow the combination of planning activities and real process execution. These phases will span from the initial transport task to the parties involved, across the bidding and agreement phase, to the commitments, and to the actual contracting phase. An IT-driven business process, for instance, must ensure that along the intermodal chain transshipment is negotiated precisely and that the contracting relationships, in their actual nestings and pairings, become adequately reflected in each business partner's backend system.

- There is a combination of the process paradigm and the ad-hoc paradigm foreseeable in the actual Use Cases of LDP meaning that a software framework for process application will have to support both context-sensitiveness and event-sensitiveness in order to allow dynamic selection and concatenation of processes at runtime. The following scenario provides a simple example: An operator is seeking a carrier to take over his load at location X and deliver it to location Y just in time. He would gather information from LDP and initiate a dedicated process within the platform, set the issue and conditions on offer, and disconnect. Interested parties in turn will become aware of the opportunity through LDP and bid on the offering[5]. If the operator is willing to hire one of the bidders he would certainly expect LDP to automatically launch a contracting process convenient for the respective mode (train, truck, ship, etc.) or, if necessary, one that is tailored to fit the selected bidder and his backend systems respectively. Likewise transparently, the operator would also expect the contracting process to take over application context from the preceding bidding process, e.g. the contracting parties, locations, etc. Scenarios like the one given above have different impacts. First of all, and as consequence of RIS dynamically connected in the background, the state of LDP is constantly changing in terms of the process offerings[6]. **Second, there are pending and floating processes sharing context, also in absence of direct user interaction.** Processes in turn will be subject to LDP automation according to a given parametrization, local constraints and global policies (user notification, deadline termination, etc.). Third, aside from global user components, the dialogs in business processes (the graphical user interfaces) are defined just for the actual activity, thus being isolated and maintained only within the respective activity class. This is one benefit of the process view: The decoupling of business activities and process state transitions (the user making business progress) leads to the well-known combination of process stereotypes (activities, transitions, pre- and post-conditions, context, etc.) and object-oriented application architecture. Over the course of a carrier process, decisions on the actual sub-processes launched and the implementation variants of activities employed will be made according to the current state of application context.

---

5 Either by transactional interaction with the skipper's process or by means of independent processes that extend a self-contained bidding context to which the skipper's process is listening.

6 In general at any time. In the perception of the user if and only if his credentials allow it to become visible.

- Due to the algorithmic, iterative nature of planning and optimization tasks[7], there is also one application type foreseeable in LDP that would normally not be modeled after the business process paradigm: planning systems. On the other hand, these rather monolithic components will have to share context and resources with processes in both directions and at the same time. Hence it follows that this type of application should be deployable as a process type containing one single activity - which is the application sharing the infrastructure of the process environment. It appears rather trivial, but the opportunity to have a complex functional application deployed as a trivial process for technical reasons is a vital migration path for legacy code and existing applications into LDP later on.

Summing up, mainly the following aspects contribute to the complexity of LDP and its development as an e-business software system:

- The common application architecture model on RIS side is not defined in detail. Standards, methods and technologies to apply in development and production are not defined yet.
- The set of basic common business process types is difficult or impossible to identify, because potential users are not willing/able to offer their internal knowledge about underlying business processes for economical reasons.
- The asynchronous messaging backbone for LDP/RIS-connectivity is not available on RIS side (RIS dispatcher). This is needed for semantic processing to perform in order to feed process offerings.
- The common business object model (common classes and responsibilities) is weakly defined on RIS side. The available set is not standardized.
- The buildtime framework for processes (software development environment) is not standardized.
- A process engine and an application runtime environment is needed, were metrics are not yet defined.
- A user portal must be customized for a dynamic process environment, working in RIS and transport context, without having credentials and an underlying public key system.
- The self-contained character of LDP as a dedicated user platform on the one hand, and the loose coupling8 with RIS and backend ERP/Supply Chain systems on the other hand – following the interface topology of a distributed system.
- For e-contracting, the need to provide transactional support between LDP processes and the enterprise backend systems. Transactions will be needed to reliably cover reservations, bookings, ordering, accounting, etc. The question of how to define and support transactions is a mission-critical issue. It involves the aspects of logical transactions, long-running transactional call stacks, locking strategies, exception handling, resource recovery, etc.
- With the exception of a few systems, it will be rather unlikely that the actual backend ERP systems involved will ever be able, for instance, to register as a XA resource9 with LDP, if

---

[7] Along with the Use Case Analysis, supply chain and resource planning will have to be addressed extensively. They promise cost-effectiveness and better profit.

[8] In distributed systems terminology, loosely-coupled means using dynamic invocation techniques, avoiding strong implementation dependencies, etc. It does not mean sporadic communication.

LDP would offer a TP monitor. Therefore, a general strategy for the deployment of transactions at application level will have to be investigated as soon as the Use Cases become available for further analysis.

- The lifecycle policies and object identity issues to address, and their appearance in business transactions. The real lifecycle of business objects is defined by remote systems (enterprise systems, ship register, etc.) and not by LDP. Most likely, LDP will apply proxy semantics on behalf of the actual object, hence the lifecycle of the proxy will be context-dependent. The object may vanish automatically from LDP if there is no further updating by RIS, e.g. through position messages etc. But obviously, it must not vanish, if it is either currently participating in a business process or it had been part of a process a short while before. In this event, it may be subject to further processes according to the user's intentions. There are value, reference, and immutable semantics to be considered for this particular family of proxy objects. In terms of LDP, there is lifecycle automation defined by the cache – but dynamically overwritten by LDP, thus requiring dedicated platform policies. Furthermore, since it is yet unclear how the (transponder) is delivered to and is delivered by RIS corresponds to the actual layout and capabilities of the related business object (e.g. a ship), there may be two proxies for each object necessary in LDP with a bi-directional association for mutual notification. For instance, in the event that a business process initially selects a dedicated object from a remote source instead from the RIS/LDP cache – simply because it is currently not listed there.

The expenditure for analysis, design, and conceptual prototyping should be included in further planning. For production in real business, the successful deployment of LDP in the logistics domain heavily depends on

- The quality of information retrieval and business processes (practical benefit),
- The ability to perform business just-in-time through e-contracting,
- RIS as a steady/continuous information supplier (messaging server) and
- The ability of LDP to understand the semantics of RIS data and automate the pre-processing into a variety of application contexts (context specification).

The logical communication network for LDP is sketched as follows:

In Figure 2-1, the arrowheads denote a uses-relationship. LDP can be accessed by all registered users through a common interface (portal), and by means of asynchronous client technology, hence allowing non-blocking bi-directional system communication. For simplicity, only the waterborne transport mode is sketched as a data messenger (RIS). All other transport modes can post data to LDP when acting in an intermodal many-to-many business scenario. This is the generalization to achieve by a mode-independent messaging backbone.

---

[9] XA defines the interface between a resource manager and the transaction manager. When a TP monitor and an ERP system (or a database) both support the XA interface they may coordinate a transaction between them; XA is a widely accepted standard of the X/Open family.

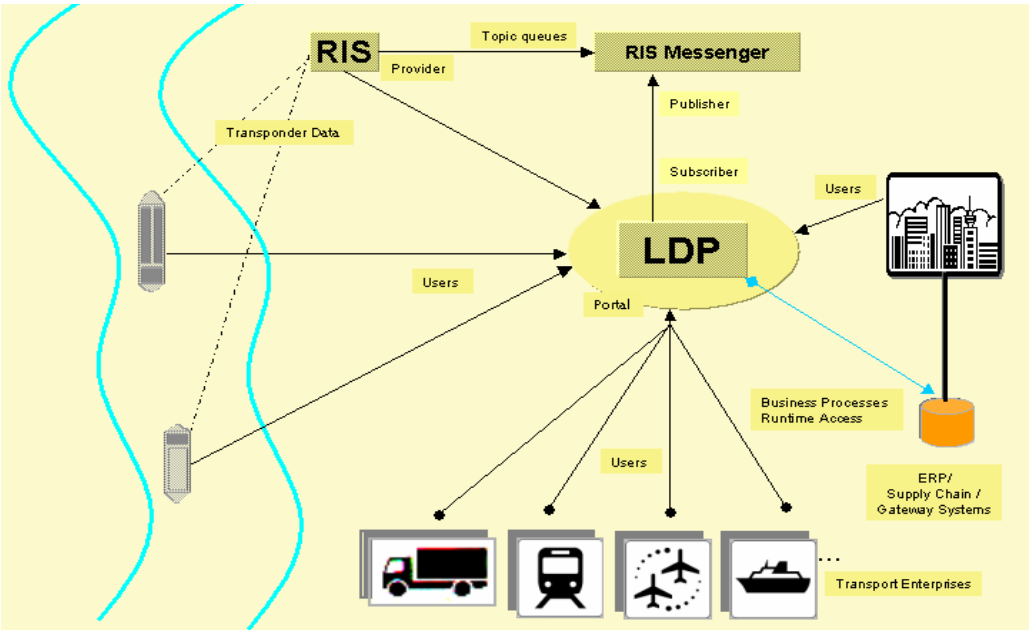

Figure 2-1: RIS/LDP Connectivity for registered Users through a common Interface (Portal)

2.2 Functional Requirements

The following is a terse pooling of requirements given as a system charter statement. Since there are currently no Use Case documents available, functional requirements combine with design and architectural outlook:

- Provide a modular and extensible process model for the mapping of forthcoming user processes to LDP; create a process runtime framework with dedicated state space, state transitions, and general process behaviour according to business needs.
- At user level, processes in LDP will be
  - offered in a sort of dynamic catalog (according to current resources)
  - accessible by means of specialized process interfaces
  - personalized and context-aware
  - logged and accounted
- Provide an application architecture framework based on the process model(s) to allow the assembling and deployment of business processes in LDP.
- Establish a continuous data communication network between RIS and LDP, acting as a backbone messaging system - providing context loading and a steady chat at system level.
- Design LDP as a stateful, asynchronous e-business application server environment
  - integrated with user directory services (admin, access control)
  - prepared for multi-channel client access to LDP
  - providing NLS-enabled offerings (national language support)
  - running processes concurrently in interactive and autonomous mode

- Provide a FIFO10 caching server configurable by application-dependent validation and refreshment policies; The server will act as a continuous-flow “data heater” that maintains incoming RIS data on behalf of LDP. The heater will be a messaging client to the (proposed) RIS messaging server and will act as a resource server to LDP at object level, hence serving objects of a lifetime t = f(Real time – T{parm1, ..., parmn}). In terms of LDP, the data objects delivered by RIS will exhibit distinct features:
    - reference semantics from the processes point of view (shared)
    - value semantics and primary keys for backend systems (most likely used in dynamic SQL for object localization and construction)
    - immutable, non-transactional
    - non-durable, lifecycle defined by the settings of the cache
    - carry access control information in order to associate with ACLs maintained in RIS or LDP

Further system analysis, especially the metrics to be collected, will reveal the stochastic queue characteristics (M/M/x) of the cache and the messaging/queueing interplay between RIS and LDP.

In alignment with the cache settings, history-related processes can become aware of the need to logon to background resources (RIS logging databases). In this event, a process would delegate the inquiry to LDP. LDP in turn will deliver the resources back to the calling process. Thereby, RIS databases will only have to grant access to instances of LDP, hence it follows the vital chance to optimize on RIS database performance by confinement to Static SQL and Stored Procedures.

- Provide a high-level user interface for the LDP clients by means of a process/activity catalog - instead of displaying raw data. Let process and activity outcomes (results) become part of the session context. The user interface will have to support the dynamic business characteristics of the platform, context exchange, etc.
- Establish Single-Sign-On (one pass authentification/authorization) and secure transmission. Process offerings and information should be classified according to the user’s credentials (access and execution rights).
- Support interface decoupling and connectivity across distributed systems by means of application services based on open standards. Provide encapsulation at component level by means of high level facades for the activities employed in processes.
- In order to keep process state under control, enforce that access to external systems (Supply Chain and ERP backend systems of parties involved) is isolated at implementation level of activities. There may be different implementations of the same activity to be chosen at runtime depending on the context of the process.
- Prepare LDP to become a TP monitor that manages all process-related transactions.
- Provide a business-related context exchange and automation facility between ECDIS/AIS and LDP according to the Use Cases (to be defined). This holds especially for skippers and monitoring centres running bridge systems and realtime ECDIS/AIS displays. They will expect the STI/TTI11 to share context with LDP, for instance in terms of activation events and context initialization.

---

[10] First In First Out (a queue of arrivals and departures)

[11] Strategical Traffic Image / Tactical Traffic Image

- Testing, i.e. the specification and execution of test cases/regression tests at unit, component, business, and platform level should be supported throughout development and within the deployment framework. Thereby, quality standards for the releasing of new processes into the LDP runtime will be defined gradually.

Remark: Till today, concise descriptions of requirements are underexposed. Future projects should seize the opportunity and recollect them in one turn with a functional prioritization and specification of accompanying test cases.

2.3 Non-functional Requirements

Currently, there are no distinct quality characteristics. Metrics and other constraints on performance, usability, safety, reliability, business rules, etc. have not yet been specified. Before heading into future platform development, this should be inspected again. At present, the nonfunctional requirements list is holding these items:

- Apply common middleware standards, software engineering methodology, and best practices from industry.
- Apply products, tools, etc. that are commonly accessible and affordable for partners. Where suitable, open source products may be employed for development and production.
- LDP must be scalable in terms of software and hardware, be configurable, and should run on major operating systems and hardware.
- LDP should support single systems setups (basic availability) as well as dedicated operating with HA features (High Availability) and system management.
- LDP documentation should be unified (e.g. by UML), to be applied to deliverables from analysis and design. Expert forums, a general information base for software development, would benefit from the use of collaborative web tools, e.g. the widely-used twiki (www.twiki.org).

3 Application Architecture

3.1 General Concept

There are four architectural domains to consider for LDP:

(1) The RIS/LDP Bridge (Connectivity Architecture),

(2) The LDP Business Process and Information Broker Architecture (LDP),

(3) The LDP Ship Client onboard a ship and

(4) The LDP Port Client.

If possible, concepts for these different components are presented now. At the moment of defining the concept, having in mind that details of the final RIS architecture will be available at the end of COMPRIS, some prerequisites are under development and requirement analysis has to be seen as a topic continuously changing.

Domain (3) and (4) were not proposed in the description of work, but are mandatory for FASTCORS development, testing and demonstration.

3.2 RIS/LDP Bridge

3.2.1 MOM Approach

Continuous feeding by RIS is a vital requirement for the operation and the business success of LDP, hence data delivery from RIS to LDP must be understood and treated as a contractual obligation. The paradigm to address this type of application is messaging.

3.2.1.1 Layout

Messaging is a system of loosely-coupled asynchronous events, requests, streams, etc. that are used by distributed objects to communicate with each other.

A message is a unit of information that is sent from one node to one or many other nodes in a system.

It is important to understand that in messaging, the destination is not, in itself, an endpoint of communication but either a queue or a topic. And a domain is a logical grouping of related topics and queues.

Messaging servers act as intermediaries between the requestor and the receiver of the messages. Messaging clients are objects that make use of the messaging server through a dedicated API. Message-oriented middleware (MOM) provides a standard and a common reliable way for objects to send and receive messages in a distributed enterprise system. Especially in terms of LDP/RIS connectivity, MOM distinguishes above other communication technologies, e.g. Remote Procedure Calls , CORBA method invocation, Java-RMI, etc. by the following features:

- Guaranteed message delivery, time independence and one-time-in-order delivery
- Fully asynchronous, non-blocking communication
- Latency hiding and transaction support (if TX services are available)
- Interface decoupling, location independence, and routing services
- Configurable QoS features and practically unlimited scalability

There are different communication models available in MOM: Request-Reply, Point-To-Point, and Publish-Subscribe[12]. The latter is the most versatile and scalable[13] since it allows publishing clients to produce messages for an unknown/unlimited number of subscribers (1:n), and leaves subscribing clients the choice between durable and non-durable connections, Figure 3-1.

Topics are a concept of classification that can be applied to individual addressees, thematic mapping, access areas, etc. There are also two options where to deploy the application logic for the selection of topics and queues. Either RIS, as the publisher, knows about the topic and where to send the message, or the messaging server determines the appropriate slot by examining the message itself (filtering/semantic processing). This will lead to an increase of server work load but scalability is not affected since semantic processing still corresponds 1:1 to the message.

Being asynchronous, MOM in general frees up the client application. But in instances where the sender wants to be notified on delivery, MOM can also easily provide notification services - outside transactions. Therefore, handling over responses to other parts of an application and journaling is realized easily. **A messaging server will also offer durable subscriptions - meaning that it will keep messages in persistent queues while the subscriber is disconnected.**

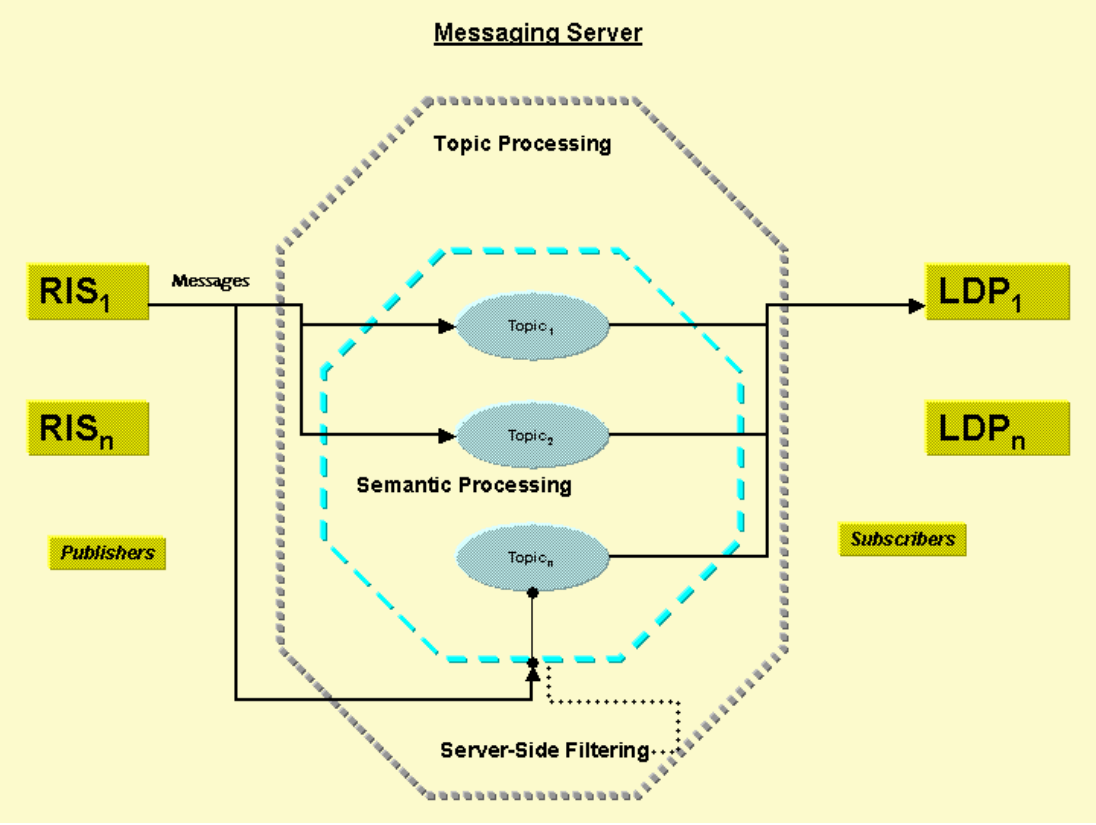

Figure 3-1: Messaging Server

The actual design pattern and techniques used for the implementation of the RIS/LDP bridge will be left open until metrics and message requirements have been further investigated. The decision can be made ad hoc and in one turn with the implementation of the messaging server since these patterns are readily available through all major MOM products.

---

12 For protocol samples see appendix

13 By adding more machines that look at the same queue

3.2.1.2 Message Models

Message Models define the unified interface needed (API) for creating and sending, and receiving and interpreting messages throughout the system. A unified interface supports messages that

- span across operating systems and machines (different code sets and byte order)
- can store common LDP objects in their content, e.g. ships, freight, trucks, etc.

The actual message model to deploy will be subject to further discussion. Basically, there is the well-formed LDP object type space on the one hand and a generic message model on the other hand to choose from, e.g. defined by XML. In some cases, for instance when messages are used for configuring the client application dynamically (class loading), a hybrid approach can be interesting.

Native Messages:

Using native messages means to either rely on the message formats and preparation techniques provided by the MOM product or to create one own's application-specific API - depending on the features of the product and the actual programming language environment.

Currently, there is one message model standard available on all major platforms - the Java Message Service (JMS). For the benefit of having text, map, stream and object messages right at hand, JMS can be applied if there is no specific need to process RIS messages outside a Java runtime environment (in the event that LDP itself is hosted in Java).

Then, object messages would be constructed by means of the Java Serialization API which in turn is directly supported by the JMS message model.

In the event that, for instance, performance is mission-critical and Java/JMS is not the technology of choice - due to limited machine resources, etc. other frameworks can be applied to realize a publish/subscribe messaging system, e.g ACE[14]. Here, a CORBA interface would be deployed to load data dynamically into LDP, thus being accessible at object level and for all major language environments.

XML Messages

In order to support further decoupling of message transport and message content, and with regard to future investigations, e.g.

- the responsibilities of RIS data centres in terms of data distribution
- dynamic graphical user interfaces, thin client technology
- wireless access to LDP / mobile devices
- the processing needs in heterogeneous environments (outside LDP)

XML may provide interesting features for the messaging approach: XML is human and machine understandable (editable), it is a standard with a powerful validation model, uses Unicode, and allows object bindings to native language environments, e.g. Java, C++, Python, etc. via the Document Object Model (www.w3c.org/DOM). By means of style sheet transformations (via XSLT) XML can be processed into different representations, e.g. database records.

The trade-off between native messaging and XML must be considered carefully. XML is tempting in terms of data handling and web connectivity, but when it comes to business logic, a mapping has to

---

[14] Adaptive Communication Environment. An asynchronous C++ framework for high-performance network computing [Ace02].

be defined and maintained, and complex business processing while searching DOM trees can be time-consuming.

3.2.1.3 Naming, Trading and Directory Interfaces

All clients willing to participate in a messaging system (send/receive) will have to know where to find the messaging server, the topics and the queues according to their names respectively. Especially for administration reasons and to guarantee separation of concerns, this is accomplished by the combination of a naming service and a directory service. Initially, the client is given a URL and a standard interface, e.g. JNDI15, COS Naming16, etc. where to find the naming service. On request, the naming service will deliver the initial context factory of the messaging system and the directory reference holding the named topics and queues for the given context, e.g. hosted by LDAP17 servers.

If required, RIS data centres and the LDP platform can be dynamically bound together on a many-to-many base by means of a trading service - an extension of the naming service.

Given certain user criteria, e.g. region, waterway, etc., the trading service would for instance procure the remote reference(s) of a matching RIS environment, hence allowing further connection into other domains.

3.2.1.4 Organisational Issues

The messaging blueprint supports a variety of communication options for the application architecture of the RIS/LDP bridge. The actual choice depends on the tasks and future technical alignments of RIS centres.

In the first place, responsibilities need to be clarified:

- Who takes responsibility for the distribution of RIS data?
- Who transforms RIS data for logistical use and how is it processed?
- How is access control managed at organizational and technical level?
    - Which primary key(s) define access control features?
    - Will these keys be coded into transponder data?
    - Who will administer ACLs and the accompanying service processes?

In a likely production scenario, RIS centres will operate a central messaging server. They will decipher ECDIS (e.g. water depth) and AIS data (e.g. positionsand identifications), join it with access control information, and post it to the individual topics accordingly. Given one or more topic names, authorisation and the address of the RIS system(s) maintaining the topics, LDP will start to listen to the topics on behalf of the user. Continuously, LDP will bind the message content to the user's process environment - enabling dynamic offerings and the use of these resources in business processes.

Eventually, LDP may have to retrieve RIS data in raw, thereby managing the business transform itself. In this event, LDP should adhere to the messaging approach since it provides the necessary separation of concerns at platform level. The messaging server will then provide a server-side method implementation such that the data is frequently downloaded from RIS and pushed internally, instead of being pushed onto the topics by RIS messaging clients.

---

[15] Java Naming and Directory Interface

[16] The correspondig CORBA object service

[17] Light Directory Access Protocol

There will be also different set - ups of the messaging server and the clients possible - according to the actual policies deployed in RIS/LDP communication.

3.2.2 Discussion on Variants to MOM Approach

Given the contractual obligation between RIS and LDP and production conditions, there are currently no adequate alternatives to messaging imaginable. It is a mature concept and it is also easy to understand for application developers seeking a connection to the backbone at business level. The level of decoupling provides a maximum of flexibility - eventually needed to reorganize different aspects of RIS and LDP later on.

Regarding classical solutions, and even if the volume and the frequency of RIS data would turn out to be rather low in comparison to typical enterprise class messaging systems, the combination of the requirements

- loose coupling at application level,
- permanent and timeout-free data distribution from RIS to LDP, on a 1:1 or n:m base and
- high-frequency, short-lived publishing sessions

will prohibit an application design on top of database systems or by means of a synchronous communication bus.

In a database solution, especially near realtime, each publishing event would correspond to a forced transaction, thus competing with SELECT statements issued by LDP processes and other applications.

In a synchronous system approach (blocking calls), exception handling at application level will become a major issue. Service and connection time - out exceptions will need to be handled excessively since there is only a limited number of sockets and threads available.

In terms of access control management and secure transmission (SSL, etc.), there is one fundamental alternative available:

In a Public Key Infrastructure (PKI), on top of the messaging layer, RIS data will be encrypted by the sender (a corporate signed key), broadcasted, and deciphered by only those recipients who are holding the respective private key, eventually further secured by means of key splitting. One application scenario would be to let the user submit his list of keys along with his session authentification, thereby enabling LDP to decipher the data on behalf of his identity.

This is a subject for further investigation. At present, neither the technical nor the organizational implications in a transponder-based environment can be assessed adequately.

3.3 Logistical Data Platform

3.3.1 General Concepts

3.3.1.1 Server-side Object Semantics

In the preceding chapters, LDP was introduced as a dynamic e-logistics B2B[18] infrastructure environment. Besides the general use of distributed systems technology for object semantics, communica-

[18] Business-to-Business. B2C processes (Business-to-Consumer) may be offered through LDP alike B2B. Currently, they are not in the focus of interest.

tion and backend integration, LDP qualifies for a hosted, application server-based design - implicitely central, and in contrast to a physically distributed system, in particular a distributed database solution.

On the other hand, for technology-related and organizational reasons, LDP will avoid to make the number, the size and the production setup of LDP host instances a compelling condition. In this way, there may be one or many LDP instances listening to one or more RIS messengers, either independent of each other or working collaboratively, and no matter how the actual production setup will look alike, e.g.:

- LDP on the big iron. A pan-european logistics turntable, hosted by IBM mainframe or dedicated UNIX server technology,
- At a regional 1:1 relationship with a local RIS authority,
- As a self-hosted server system, e.g. Linux-based, acting as a B2B/B2C business process front-end to ERP legacies and Supply Chain Systems.

Likewise, no concise assumptions on the location and type of the backend systems can be made in advance. The embedding of remote backend systems into a transactional context created by LDP will always be a matter of implementation details since one can usually not expect these systems to register with LDP as, for instance, XA resources - if LPD would provide the TP monitor framework. Instead, the application model will enforce the details to be enclosed by either the respective business activities or the business objects themselves, thus hiding stubs, protocols, and system-specific business logic from LDP. Therefore, a single logical business activity in LDP may also map onto a set of implementation variants.

By evaluation of the process context, frequently depicted and coded as a hierarchically growing tree, Figure 3-2, the activity may switch dynamically from its defaults to one of its code variants, thereby providing specific server-side logic and specific GUI controls that fit the activity in its actual process context.

The trigger can be a certain node/value pair detected in the context space, e.g. a specific transport mode, a distinct contracting party, etc.

The technique of context-awareness, combined with dynamic object bindings, is easily generalized in upward direction, from activity level to process level, and further on to session level, thus defining a common binding model for LDP:

At process level, activities can be switched entirely. At session level, the user-specific process offerings can be configured just-in-time according to the information held in the context, e.g. the outcome of previous processes, the signalling of resources coming in from RIS, etc.

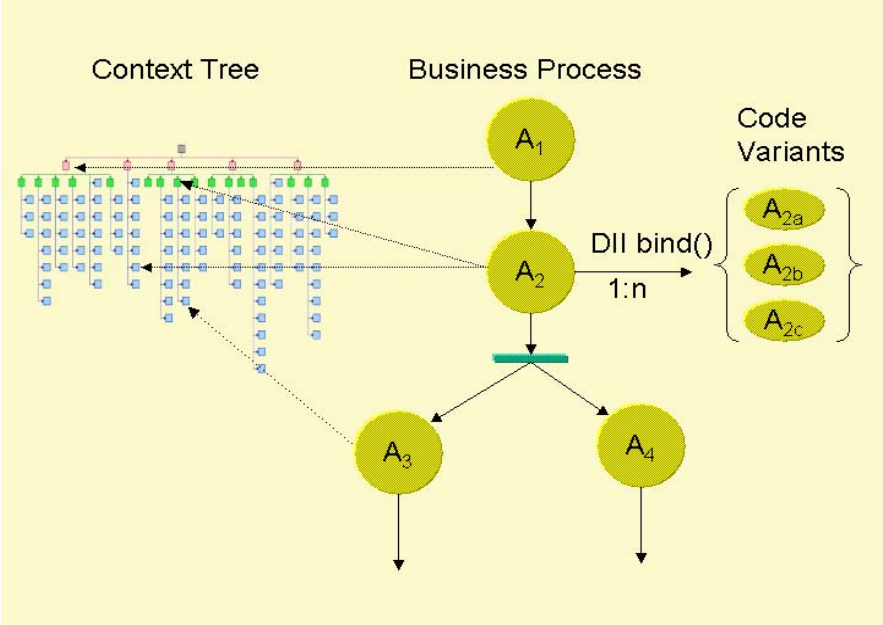

Figure 3-2: Context-Aware Binding

As introduced by the discussion on object lifecycle and proxy semantics, the situation becomes more complex at business object level. Typically, these are the process resources to be consumed, to be produced and to be affected by transactions, e.g. ships, contracts, contracting parties, etc. Since LDP will never have the true business object on the handle, with the exception of those events when a transaction with a particular backend system[19] is performed, relaxed policies, dedicated locking strategies and refreshment techniques will be needed to cope with the following issues:

- The actual state of the resource may differ from the state currently known by LDP, hence uncertainty will increase over time. Moreover, existing application software outside LDP will interfere with the backend systems, thus changing resource state concurrently and independently.
- LDP processes may show different transactional characteristics. Depending on the actual Use Cases, there will be short-lived, non-transactional and almost stateless operations leading to immediate results on the one hand, and rather long-running processes on the other hand. The latter will, in themselves, define a logical transaction context (LTX) that becomes nested further down the process. When two or more processes compete for one resource, e.g. a vessel to charter, one of them may commit its LTX at a certain point of time, thereby winning the race. In this event, the other LTX competitors will have to be signaled, at least. All further action may depend on the application context and the platform policies, but it does not necessarily mean to enforce the termination of the competing processes.
- One must be aware that user frustration may be high, and LDP acceptance low, when these event types are not marshalled onto other processes. A user will not experience true benefit from LDP if he is left unknowingly while working through his own business process, and then finally rejected by a write lock exception of a remote database (attempting to commit a transaction that has recently been superseded by another user's transaction).
- As mentioned before, the business objects (the process resources) must be treated as single logical instances within LDP since many user processes may compete for the resource in

[19] Usually the ERP systems of contracting parties manage object identity and state

concurrent transactions. That implies the need for state management and multi-threading features of the business objects. Although it would be much easier to maintain isolated copies of the resource in each user process, the complicated approach is likely necessary since reference counting of the business object's proxy, a preliminary for lifecycle control, can be tracked by LDP only.

- In a classical LTX situation, a growing number of database locks on the table items of resources would be acquired and retained by the process over a considerable time span - until it comes to either a COMMIT or a ROLLBACK of the enclosing transaction bracket. Naturally, a nested transaction at stage x cannot commit until its superior transaction at stage x-1 has received its COMMIT message.
- In order to guarantee resource availability, most enterprise systems will definitely not allow the non - deterministic blocking times that arise from LTX scenarios, especially when locks are claimed by alien software (such as LDP). Therefore, LTX cannot be supported in LDP unless its own policies and strategies for dealing with concurrent transactions have been defined, e.g. a general optimistic concurrency control scheme.
- For technology reasons and according to experience of FRANZIUS-INSTITUT, the chance to maintain a transaction bracket across two (or more) remote backend systems is almost near zero. At a future level, when LDP is in the role of a LTX monitor, will have to apply policies and TX workarounds compliant with the ERP systems of contracting parties. Besides common business types, transactional interfaces are a topic for bilateral investigation among project partners, especially logistics enterprises. It is also imaginable that activities responsible for e-contracting are dynamically chosen according to known pairings of the contracting parties appearing in the process context.
- Example:
  An order has been committed to a remote ERP system over the course of a business process. Later on, the governing LDP process is aborted by the user (for any reason). Then, in a homogeneous 2PC environment, the order transaction would be just rolled back by all TX resources affected. In most cases, this will not be feasible in LDP due to the heterogenity at interface and TX technology level, hence special organizational action has to be taken on both sides of the transaction context, by LDP and the backend system affected. ***But even if there was a timestamp policy available in order to guarantee that no other transaction has changed the state of a resource in between, most enterprise systems will usually not allow the inverse (undo) transaction later on, as a simple reverse of an entry.*** *This is always a subject to log redo processing, data cleansing by DB administrators and business experts.*
- The state of a resource object (e.g. vessel position, fuel stock/consumption) and other information possibly delivered by AIS transponders, as well as the data originating from other messaging devices, such as GPS/GSM etc. used by other modes, may or may not be sufficient in terms of the business process willing to proceed (at runtime).
- This in turn directly affects the pool of regular expressions to be able to choose from when specifying the preconditions and basic state transitions of each activitiy in a business process.

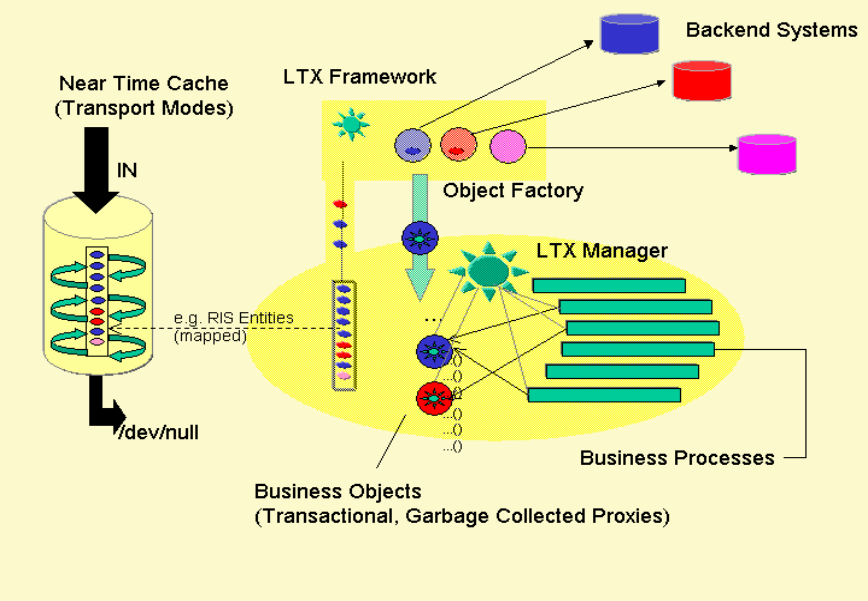

Figure 3-3: Proxy Objects as TX Resources in LDP

LDP will define its own patterns to overcome proxy-related problems. The actual approach will be influenced by the RIS/LDP transform, the period of object validity (the neartime range), and the level of state completeness needed in terms of the process demands. At this time, and due to the lack of Use Case evidence, only starting points can be given:

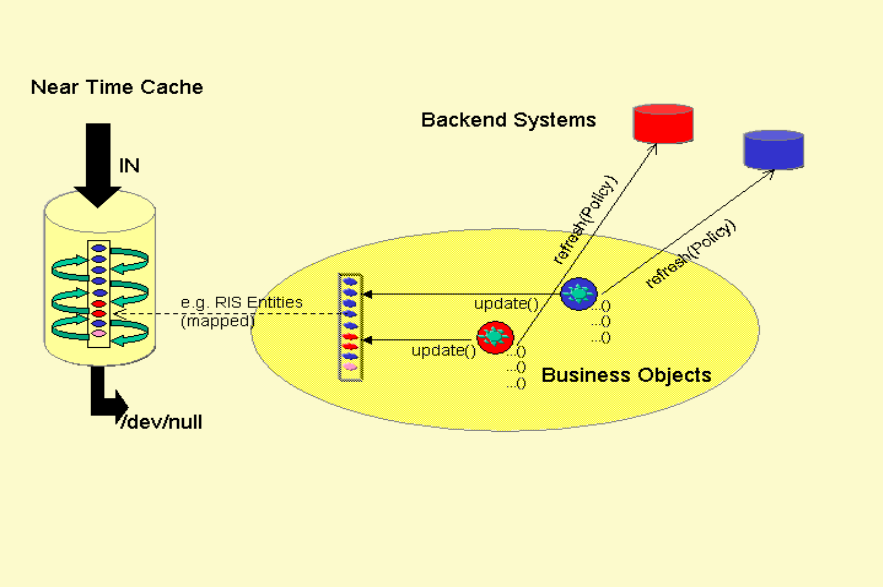

Figure 3-4: Proxy Refreshment

If the objects delivered by RIS turn out to be sufficient, they can be morphed and introduced as business objects. When a process claims an object the first time, it will acquire the features for instance by a factory transform, by connecting to its home base ERP, etc., Figure 3-3.

It will then carry the policies and the code to refresh itself. Along the transform, it may have inherited the ability to participate in concurrent LDP transactions without affecting the backend system. It will also start to listen for messages arriving in the cache (position updates), hence being reliable for the use in LDP processes, Figure 3-4.

In balance with the cache settings, one policy could be to limit the business process lifecycle a priori. The process will either have to finish within a given time t or become subject to garbage collection. The full-blown business proxy will die if there is no further process referencing it. Its flat ancestor may remain in the cache until either flushed or picked again by another process instance.

In general, LDP operation policies will be adjusted according to the Use Cases. But it is quite clear, due to the characteristics of neartime processing, that LDP will not allow to persist business processes with the meaning of suspend/resume in the first place. This is an option for the future, when the need arises and sufficient heuristics can be given. Basically, LDP will provide a stateful, long-running, and transient process environment including active lifecycle (resource) management, hence not a typical workflow environment. Of course, in a HA environment, session migration and process serialization will be means to establish safe-failover and to support load balancing.

A stateful and multi-threaded LTX scenario is mandatory from the LDP business perspective. On the other hand, it requires quite complicated technology and an extensive, therefore expensive design. Restrictive access and usage patterns, e.g. by avoiding multi-valued associations, will facilitate the stateful approach. Other design restrictions may be investigated and adopted for the benefit of LTX support.

A functional approach would favour a stateless environment, keep business data in value semantics, and isolate user processes strictly from each other. Likewise, any database or EDIfact approach will do so. The responsibility, e.g. to take care for context-dependent refreshment calls, validity and consistency checks, etc. will be assigned to the application developer. Thereby, a business process type and its entire set of activities must be designed and coded in one single application context. This in turn will prevent the activities from being re-used in other process types, and interfere with the dynamic features of the platform, e.g. ad-hoc process chaining and dynamic subprocess extensions. Of course, there are ways in between, e.g. the association of entities with services (facades) that access the object's home system on behalf of the business object. Nevertheless, referential and state integrity, as well the chance for an extension with LTX services will then be lost.

3.3.1.2 Client Side

Regarding the client part of LDP, it is emphasized that for the proposed LDP process environment, a stateless client technology, e.g browser-based, cannot meet the requirements, especially with regard to the realtime and neartime features, hence the on time responsiveness expected by the user. Therefore, besides neartime data procurement, asynchronous messaging technology will be employed at both ends, the LDP host environment and the distributed clients. This does neither mean to start off with a fat client approach nor to break with the application server paradigm (server-side business logic). Instead, the client design will aim at a lightweight but stateful, message-aware client with desktop features that address the visualization and the interactivity needed in a dynamic process control environment.

For remote clients, concordant with the server side's ad-hoc features, dynamic capabilities will play an important role. Therefore, starting with a minimal footprint, GUI controls and business rules may be loaded dynamically and asynchronously in the background according to the current LDP usage context. Since the LDP server environment, in itself, manages the user's processes, a client's configuration can be scheduled preemptively instead of the client synchronously downloading classes on demand.

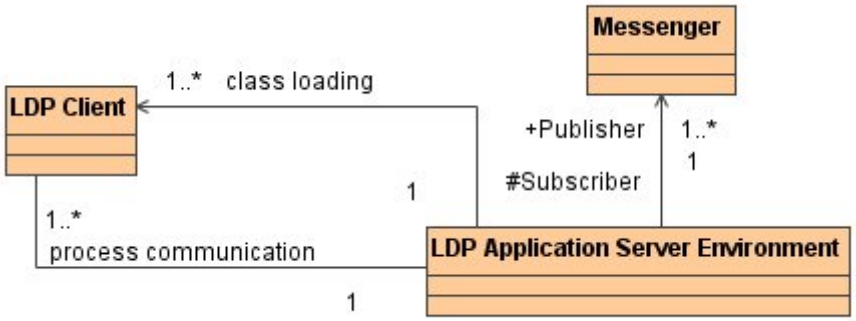

Figure 3-5: Asynch 2-way C/S Communication

The set of dynamic techniques for software configuration, in the light of client technology and bandwidth, and the LDP strategies to limit software distribution, in the light of frequent changes and continuous growth of LDP process offerings, will depend on the platform usage patterns and the metrics, yet to be estimated.

Furthermore, there is another important aspect to consider for a balanced client/server design: in any distributed system, and for each interactive part of a business process, it is mandatory to externalize a certain amount of the activity's business logic for preemptive client-side evaluation. Plausibility checks on user input, according to common type and business rules, avoid user frustration through server-side rejections of inacceptable input values.

3.3.1.3 Integration Aspects

When LDP is ready for operation (technically), and the business analysis and design of e-contracting processes is placed into the foreground, the interface variety of backend systems will become a major issue. Usually, any of the business partners using LDP as a process turntable must have his backend systems somehow tied up to LDP.

Therefore, LDP must be shielded from interference with backend system particularities (the heterogeneity), e.g. from the foreseeable variety in

- low-level networking and business communication protocols
- programming language APIs
- transactional behaviour
- organizational and security-related standards

Any such particularity of a backend system, e.g. EDIfact over X.25 instead of IP, must be hidden at activity or object level, thereby not affecting LDP process state transitions.

The application architecture framework will ensure that any such coding will be placed within elements of this type.

3.3.1.4 Building Blocks

Covering development and production systems, the main buildings blocks contributing to LDP can be depicted as follows:

The blocks located at a lower level of the LDP baseline architecture, e.g. NLS enabling and Common Object Services (like server dedicated to neartime/realtime messages issued by transport modes. It has been introduced already through the design of the RIS/LDP bridge. The actual core of LDP is the process environment consisting of a buildtime framework, a runtime framework for process execution (engine), the LDP standard client technology and a common server access interface (the portal). It will

provide hooks for non-default clients and devices. A deployment workbench, test automation, monitoring and administration components as well as a user help system and issue/problem tracking tools will be needed for production level operating. The portfolio and interplay of all building blocks will be subject of forthcoming discussions. The first steps towards system design will address the runtime framework (core). In parallel, a strategy for the Logical Transaction Manager, the LDP-specifc LifeCycle Management and Directory Services will be developed.

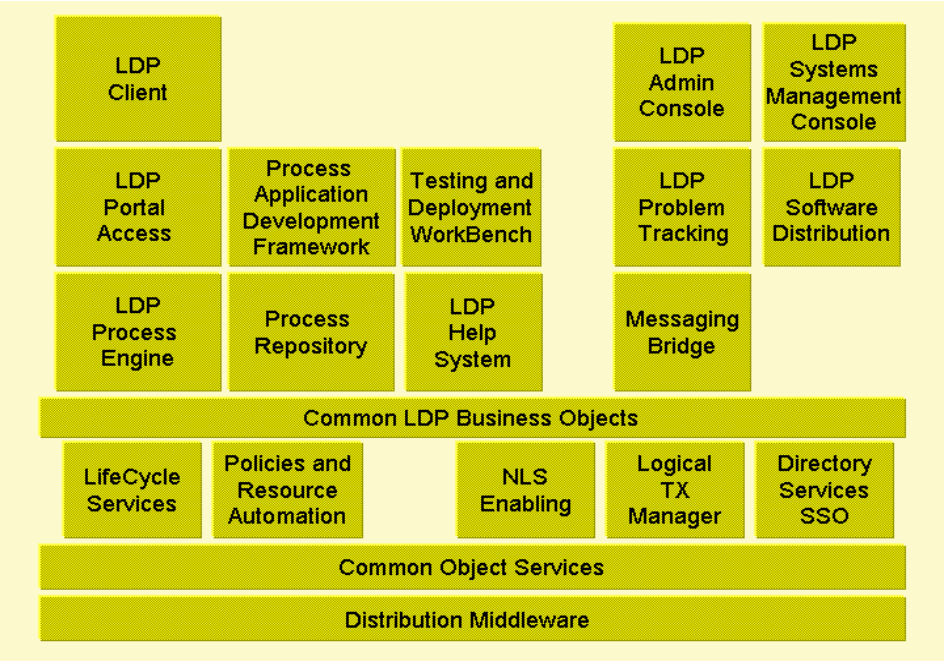

Figure 3-6: LDP Building Blocks

Most important, and at the heart of LDP, there is a generic, transferable and technology-independent process model that rules the application architecture. It defines the semantic foundation of the application family, thereby providing the patterns for all forthcoming information broking and business processes within LDP.

3.3.2 Generic Process Model

Basically, there are two conceptual models eligible for the business and information broking processes in LDP. Both are related in terms of steering logic - but quite different in terms of steering technology:

There is the graph-based model on the hand, usually a directed acyclic graph (dag) acting as a state/transition net (Petri net), and consisting of activities (predicates, nodes), transitions (edges), and technical elements like start/end nodes, splits, joins, options, etc. The dag represents the flow of control to be executed by the process engine. It is accompanied by a second graph which is given by the flow of resources to be consumed and produced by activities. Both graphs are disjoint, and usually, the graph for the resource flow must be treated as a graph with local cycles in lateral extent. The graph-based approach qualifies for complex, workflow-type processes with a large number of activities involved.

On the other hand, there is the agent-based model. Here, no global interpreter is applied in order to execute process state transitions. Instead, each activity within a process will act autonomously according to a given precondition - its current parameter. When the process is started, the activities belonging to the process will observe a dedicated resource space by means of regular expressions and change their state accordingly, e.g. become visible, executable, terminable, etc. As part of the proc-

ess, there may be also batch and background activities with no user intervention. All activities will consume and produce resources by means of the resource space.

This model qualifies for dynamic and non-sequential processes, in contrast to deterministic production workflow and processes. It has the potential to deploy rare but powerful techniques, e.g dynamic rewriting of preconditions, on demand loading of activities, thereby enabling ad-hoc process concatenation and process extensions, etc.

Nevertheless, both models require precise and comprehensive business logic specifications in order to avoid deadlocks and ambiguities while executing. In general, the graph-based model will be more challenging in terms of the interpreter and the tools needed for application development. The agent-based model will be less complex in terms of process types but challenging in terms of event communication and threading.

In the first place, and since graph-based steering may be introduced as an add-on when the need arises, we recommend the agent-based approach for LDP. The coarse meta model, depicted as a UML class diagram, is read as follows, Figure 3-7:

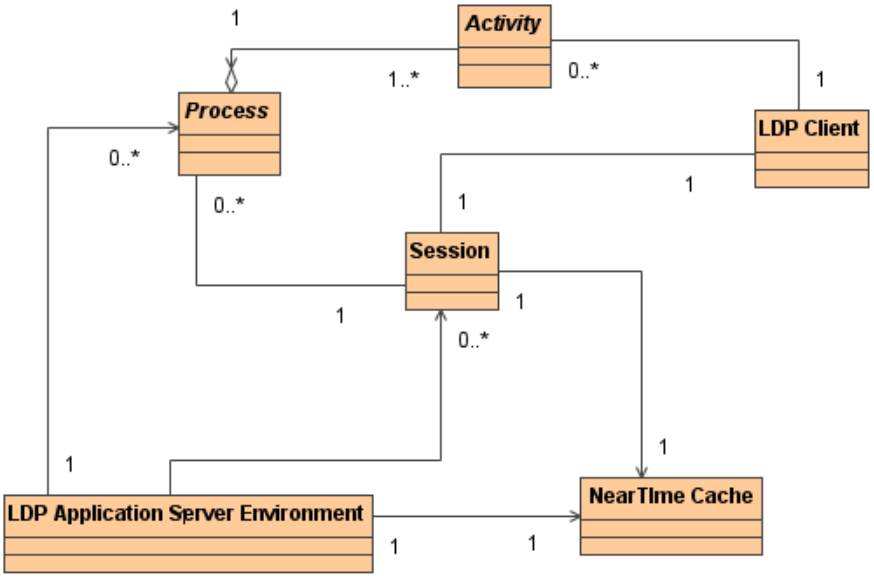

Figure 3-7: Application Meta Model

One or more activities (a set) constitute the process. Each process is under of control of LDP and likewise associated with the user's current session.

The session maintains the communication channel to the remote LDP User Client (the user) and is also connected to the neartime cache. Thereby, realtime objects can be introduced as process resources, process activation events, etc. into the LPD user space, either automatically or when selected interactively. While working through the processes, the user will be in dialog with the respective server-side activities, accessible through the LDP client desktop.

At a conceptual level, the processes exist and run under full control of LDP without the need for a session or a user (user disconnected, processes running in the LDP background).

While the session is at most a technical element to manage user/client interaction with LDP, a process defines the high-level abstraction to express application functionality (a realization of Use Cases). Processes may be also assembled from other processes (by composition) and use subprocess relationships. A subprocess is a process like any other process but has runtime call stack semantics: it will run in synchronous and modal mode with respect to the carrier process (the creator). Therefore, the

respective activity that launches a subprocess will remain blocked until the subprocess has returned. Subprocesses can be embedded recursively. Naturally, activities can also start other processes asynchronously (fork, oneway), hence returning to their own thread of control immediately.

3.3.2.1 Process/Process Type Relationship

A process is a business process instance derived from a process type. A process type in turn is a scheme defined at buildtime. The scheme holds the activities to be employed in the process, and their respective parameters, hence the preconditions to apply to this particular process context. Process types will be defined as logical single instances and actively used by LDP, e.g.

- to determine the dynamic set of processes that are startable according to the neartime resources available (e.g. from RIS) and the user's ACL, constituting a process catalog.
- Any LDP process type will have its global precondition evaluated continuously. It will be offered to the user for process construction if, and only if, a successful execution of a process instance is possible - fundamentally.
- to provide factory/construction services for the actual business process instances when the respective process type is requested by the user.

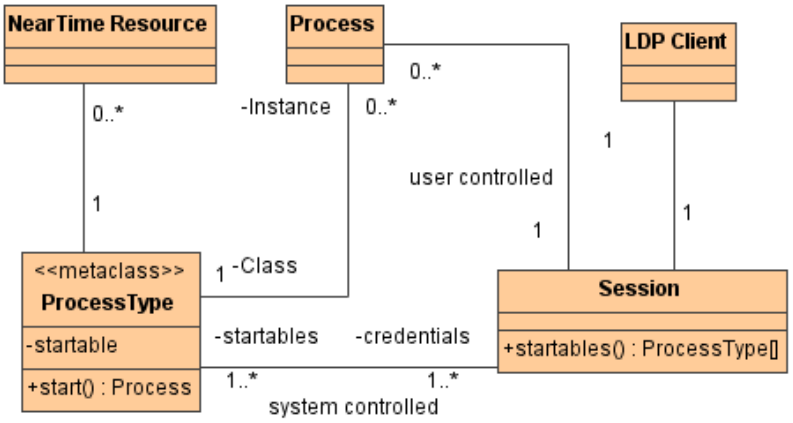

Figure 3-8: Process Type / Process Relation

Figure 3-8 shows the basic relationship between process types and processes. If the boundary conditions allow for a process type to be started by the user, which is automatically determined by LDP, one or many instances may be created over the course of the session.

3.3.2.2 Activities

Any process in LDP is composed from a non-trivial set of activities. The activities denote a decomposition of the process at business level - by no means at algorithmic level.

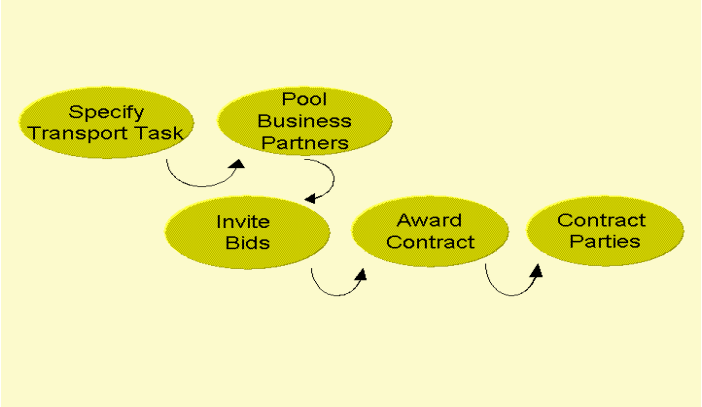

Figure 3-9: Logical Units of Work

Therefore, each activity defines a logical unit of work, a business-related abstraction that gradually contributes to the success of the process. Typically, business activities will be defined at the following level of granularity, Figure 3-9:

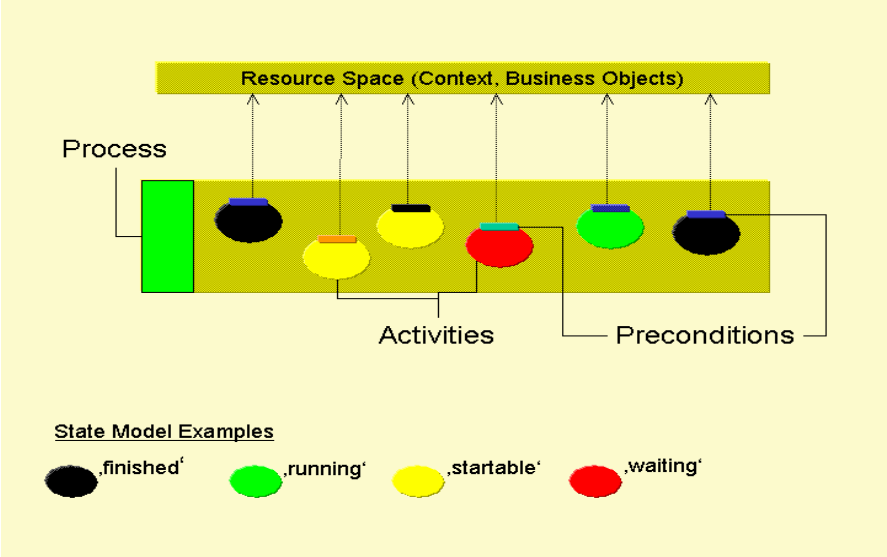

Figure 3-10: Activities and State Model

From the outside, and according to their role as active model elements (agents) in application architecture, activities will define a common state space, a common behaviour, and generic services for the process environment. They will, for instance, signal readyness, work state, etc. according to their state model automatically, Figure 3-10. By combination of state information derived from:

- scheduled, triggered or event-based autonomous evaluation of regular expressions (the pre-conditions), especially while not in direct interaction with the user, and
- inner, business-related state transitions

the activities, by themselves, will entirely manage and post the information needed by LDP for process steering. The inner coding (the implementation) is dedicated to specific business logic and will perform the transactions that affect the business objects and other resources in context.

With the exception of predefined batch/background units of work, most activities will be designed for interactive use, and thereby associate with a set of dialogs, usually a graphical user interface (GUI). Since the instance of an activity encloses its business logic completely, a GUI is defined only for the scope of each activity. Therefore, activities and GUIs will be designed and deployed pair wise.

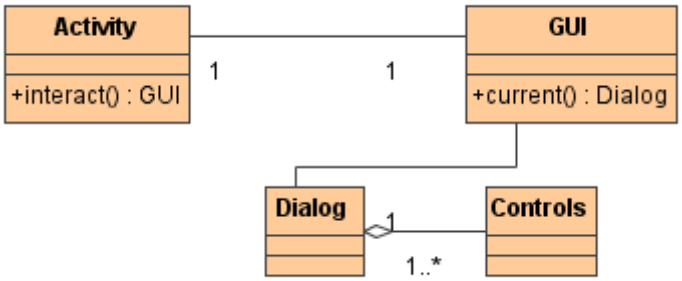

Figure 3-11: Activity/GUI Pairing

Thereby, and if well-designed, activities may be re-used across process types without modifications.

Furthermore, in contrast to true agent-oriented systems, activity objects in LDP will not move around or migrate to remote destinations. Like the governing business process, they will be hosted in the address space of the LDP server system. Thereby collocated, the overhead of a distribution software bus can be avoided for inner process communication (home interface). Depending on the client technology they will eventually provide a stub for remote client communication (remote interface).

Apart from starting processes directly from within activities, either by passing context to subprocesses (modal) or to standalone processes (fork), activities may be also loaded on demand into a running process - although not belonging to the process type by default.

The combination of processes, process embedding, subprocesses, and on demand loading of activities, will contribute to the powerful ad-hoc capabilities of LDP.

Therefore, besides common business activities, there will be a limited set of technical activities, e.g. subprocess activities and composite activities, etc..

3.3.2.3 Preconditions

In order to become active, agent-like components in LDP, process types and activities will support dynamic preconditions. In the given context, preconditions are regular expressions to be evaluated at any point of time in runtime. The results will be used by the instances to monitor the resource space and the surrounding context in order to adjust their state and behaviour themselves accordingly.

In the example given above, Figure 3-9, the activity ‘Invite Bids’ would change its state to ‘startable’ if, and only if, there is a collection of business partners visible in the resource flow, which in turn appears when the activity ‘Pool Business Partners’ is finished and releases the particular collection into the process context. Therefore, the ‘Invite Bids’ precondition to become ‘startable’ may be specified as:

ResourceSpace->exists(BusinessPartners) and

BusinessPartners->notEmpty

While the condition is not met, the activity will not present itself as ‘startable’ to the user. Similarly, other state transitions can be defined until a process state space is fully specified.

There is also an important side effect: since the resource space is easily shared by concurrent processes, the transactions committed in one process and the resources produced, will become visible to activities in other processes immediately, hence leading to a self-sychronizing process (application) environment20.

---

[20] Usually, enterprise systems do not provide means of active synchronization. It is simply enforced at database level, thereby leaving write lock exception handling to the application.

Typically, the preconditions will be deployed with the process type in the process repository and loaded at runtime. The expressions will be formed according to standard grammars, e.g. OCL, the Object Constraint Language (Warmer & Kleppe, 1999), by means of native code, by means of scripting languages like Python etc., depending on the language bindings available in the actual technology environment.

It is obvious, that the semantic multiplicity of preconditions depends on the build-in object type space of LDP, the number of types, their sets of states and operations (methods) respectively. For many reasons, e.g. performance and system robustness, this type space should be as limited as possible on the one hand, and as complete as possible in terms of business logic on the other hand. These are conflicting aims that have to be addressed by careful and concise OOA/OOD. The particular types are specific for the logistics domain and will constitute an important part of the resource space. Instead of the Use Cases, and the proceses that may grow and evolve over time, these objects will become a pervasive application element in LDP right from the beginning.

3.3.2.4 Flow of Resources

As explained, the agent-based model proposed for LDP does not require a graph for process steering.

Instead, the activities co-ordinate the full set of state transitions by themselves, and in combination with a stateful process instance that is aware of the respective activities posting their outside state.

Nevertheless, a graph of resource flow is needed, especially in order to avoid ambigious assignments. Simply speaking, if there are many instances of one type listed in the resource space, the activity must either know which particular instance to claim or must consider additional information for decision making. The latter is not recommended since the knowledge to apply is highly context-dependent and must be hard-coded, thereby disqualifiying the activity for use across process types (re-use).

Furthermore, the resource graph is a powerful option to boost producer/consumer perfomance and to optimize event channel communication:

- Given at process startup time, the graph tells each activity, in advance, from where exactly to expect incoming resources, of which particular type, and where to deliver resources produced - if not to the resource space, e.g. to the session context, a file, a database, etc.
- If there is, at event processing level, a generic map of event channels available for resource flow, the activities can tell by the graph where to register and listen for events of a certain type (the resources to be consumed), and which signals to emit (the resources produced).

To describe the flow of resources, the main activity types are introduced, and their receptor elements dedicated to resources - the slots. The definitions are given as follows:

- The common activity is a pattern for business-related activities,
- The common activity has input and output slots (north/south face),
- Each slot accepts one collection of one distinct type,
- The collection is treated as being of the same type as its elements,
- A collection guarantees disjoint elements (in terms of object identity),
- A collection behaves unsorted (the order of elements is neglectable),
- There are one or more slots for *subprocess launching* (on the east face),
- There one or more slots for *subprocess results* (on the west face),
- The *subprocess activity* has no east/west face (to denote embedding),

- The *composite activity* handles the embedding of a full process type,
- The *composite activity* has no east/west face (is in itself a convex hull),
- The *asynch composite activity* starts a process in non-blocking, oneway mode,
- The *asynch composite activity* has no east/west and no south face and
- The *I/O activities (usually further subtyped)* provide external import/export interfaces for the process, e.g. to release resources to superior context, to acess files, databases, etc..

The multi-valued, typed slots of activities can be depicted as nodes of the resource graph. The simplified picture, Figure 3-12, shows how activities can relate to the resources produced and consumed by other activities. Hence the arrows display event channels and read as “Input slot A listens to resource availability at slot B”.

When event signalizing occurs, the activities may adjust their state in alignment with their preconditions and signal it to the process hull (and the user). It does not imply to start execution and consume the respective resources immediately. Since startable activities usually denote the next step(s) of workflow, the user will have to invoke a ‘startable’ activity explicitely unless it is marked for asynchronous automation by LDP.

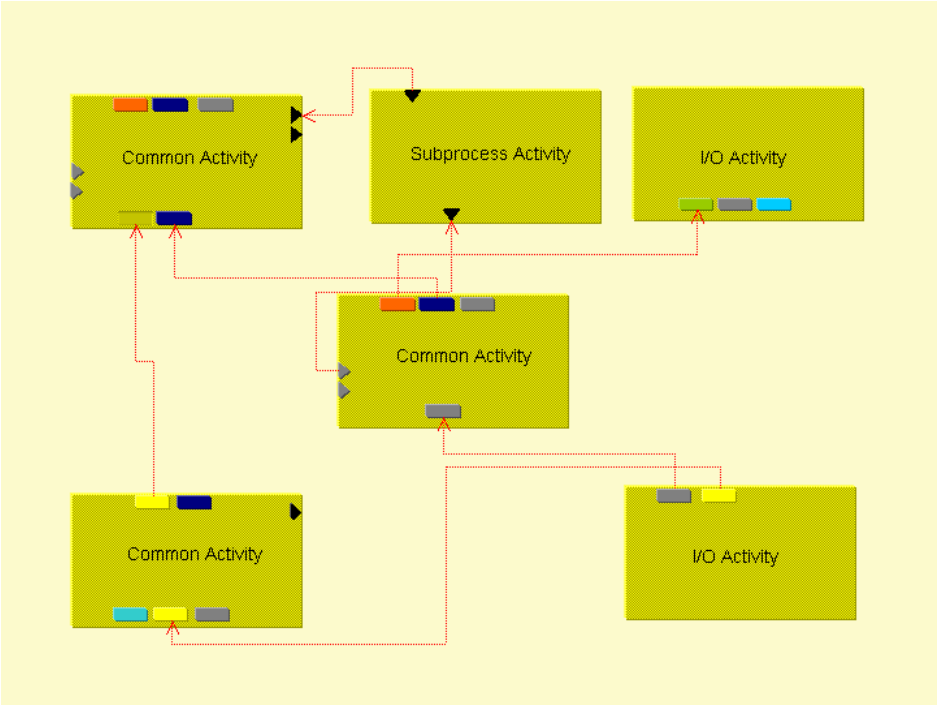

Figure 3-12: Graph of Resource Flow

The behaviour of activities and the tableau of preconditions must guarantee at least one correct and deadlock-free passage through the process. There may be many passages, and in no particular order, but likewise deadlock-free, thus leading to one of the standard process outcomes (successfully finished, aborted).

More formally, there is at least one topological sorting of activities (an order) that implicitly defines a direct acyclic graph in terms of steering. Likewise, in the terminology of the agent-based process model, the topological sorting says that it is guaranteed that an activity A will not become ‘startable’ before all other activities that provide resources for A have ‘finished’, thereby releasing their resources in committed state.

3.3.2.5 Context Hierarchy

Context is manifold and has been mentioned throughout the document, e.g. application context, transaction context, context-dependent bindings, etc. In general, context is a hierarchy of information and state that, as a means of decoupling, is used by system and application components to adjust their behaviour according to the current process situation (context-awareness). Context interposes key components and provides the runtime glue at different scales.

For example:

- At session level, context is introduced through
    - the user characteristics, e.g. ID and credentials, native language (locale), current location/position, etc.
    - the processes running on behalf of the user and managed by LDP
    - the resources released for re-use by other processes (results)
    - the real - time link to resources outside LDP
- At process level, context is introduced through
    - the history of a process (not to mix up with the call stack), e.g. depicted as a tree of business progress and used for logging and online help systems
    - information that is bound to the process life cycle and does not belong to the resource flow but to the general business type/system type space, e.g. a legitimation code requested by a backend systems (not administered in LDP).
- At business object level, context is introduced through
    - logical transactions, hence the state managed by the object for each of the transactions affecting it.

The Use Case analysis will reveal the items, the metrics, and the usage patterns of context. Most likely, and for the benefit to have an organigram of the run - time processes in LDP, context will be designed as a managed tree, Figure 3-2.

Since processes may run on behalf of the user while being disconnected from LDP, the context tree will be linked dynamically into the session. Furthermore, it will be assembled from those parts that are cumulative and immutable (according to process history) and those parts that are modifiable by application components and subject to garbage collection (according to the respective process and object life cycle polices).

3.4 Software Architecture on Client Side

The proposed LDP framework guarantees a maximum of flexibility and functionality for software development for clients.

In waterborne transport, these clients will be located at ports, logistic service providers, waterway authorities, ships and trucks etc..

Client software can be either developed or adopted using Java and/or C++ in an adequate environment (J2EE, ACE).

Existing software applications can easily be adopted, if one of the mentioned programming language is used. It will be shown later how communication with LDP can be achieved very easily.

4 System Technology Suggestions

The top level use cases for LDP have not yet been worked out in detail. Therefore, and because the system was build from scratch, the system technology favours aspects of sustainability, open standards and long-term platform evolution.

Nevertheless, as a consequence of the generic process model chosen for application architecture, unforeseen requirements will be treated by means process modeling and will not affect the core of the platform.

For the proposed products and technologies, the requirements derived are combined with uncertain belief on further evolution (estimations), architectural experience, and the general goals of an R&D project on budget, e.g. to achieve a maximum level of scientific, business and IT innovation, and to keep software product costs at acceptable levels.

It is already clear that only application server technology, service add-ons and common open middleware standards will provide the baseline technology needed for LDP. In addition, a variety of production scenarios need to be supported. The range covers hosting on PCs, UNIX server systems and, as a future migration path for dedicated operating with HA features, the mainfraime (IBM z-Series).

Basically, there are only two middleware and language environments that qualify for the core of LDP server systems accordingly. And there is no alternative on the horizon, in terms of pervasiveness in business, industry and academia:

- J2EE, the Java Enterprise Edition (Sun Microsystems, 2003). Along with the build-in features of the Java language and the Java Virtual Machine, it brings with it the middleware services and the application patterns that can be applied to LDP, e.g. transactions, threads, stateful session and message-driven beans, messaging services, etc. J2EE will also provide a smooth migration path from the PC and UNIX environments to the mainframe.
- CORBA, the Common Object Request Broker Architecture (OMG, 2003). It provides the architecture and the services of a distributed object bus, a component model, and can be married with J2EE for interoperability at certain levels, e.g. notification services, remote object invocation, etc.

For both standards, production quality open source software is generally available, e.g. TAO (Object Computing, 2002) in the CORBA world and jBoss (Jboss, 2003) in the J2EE world. Nevertheless, the actual pool to choose from is very limited due to the need for asynchronous object communication and fully multi-threaded (reentrant) servers.

In terms of CORBA, there is no need to consider a commercial product. In terms of J2EE, the application server market is still volatile. There are currently two mature products that qualify for LDP instead of jBoss or the Sun J2EE reference implementation: BEA Weblogic and IBM WebSphere. But their licensing is costly. Therefore, in the J2EE case, it is recommended to start off with a jBoss environment and eventually switch to IBM or BEA for production later on. This is easily achieved by the J2EE compliance and frequently practiced in small to mid-scale industry projects.

4.1 RIS/LDP Bridge

The choice is already given by application architecture: messaging technology, furthermore, only those products that are compatible with a J2EE- or CORBA-based approach for the LDP server system (the core). We recommend either

- JMS (the Java Messaging Service) or a solution based on
- Cross-platform asynchronous C++ networking

JMS is available from the jBoss project and also included in the SUN J2EE reference implementation. The latter can be used for non mission-critical purposes. In addition, there is a number of commercial products available like IBM's MQSeries/JMS, FioranoMQ, etc., usually at rather moderate costs.

For a high-performance C++ messaging solution, the open source framework ACE (Adaptive Communication Environment, see Schmidt, 2002) is recommended. It is generic, very robust and qualified for real-time applications. On the other hand, since it is framework, it will require more expenses dedicated to system infrastructure programming - in the first place. At least in comparison to JMS that comes with the full set of APIs. For decision making, the IT strategies in the technical-driven areas of RIS and AIS may be further investigated.

According to these options, clients posting realtime messages (not only RIS but also from other modes) will be either import the respective JMS package or be linked with the ACE client library (C++). In both cases, small memory footprints can be achieved.

4.2 Selected Building Blocks

The LDP core components hosting the processes, the event channeling, the threading, etc. will either rely on a J2EE (Java) or a CORBA (C++ or Java) server environment. Therefore, the design of most of the building blocks and the common object services will resolve according to the middleware. Other services are available in the open source domain and can be plugged in, e.g. encryption (SSL), or adopted later on with moderate efforts, e.g. stream compression.

At this time, the discussion will be limited to the issues of client technology, user management and life cycle control.

4.2.1 LDP Client Applications

For development and LDP product placement reasons, Franzius-Institut recommend to define one single standard for LDP client technology. Regarding the dynamic requirements, there is no benefit when using non-messaging, stateless client technology, e.g. the web browser. Furthermore, to achieve almost unlimited scalability and a maximum of robustness, the client may either use

- one physical MOM channel for each logical channel
    - one for neartime/realtime resource and state signaling or
    - one for LDP process communication hence subscribing to two physical messaging servers or, depending on the business needs to be fulfilled by activities
- one ore more phyiscal MOM channels and a client stub for activities, e.g. with a CORBA interface.

If desirable, a synchronous call interface can be added to the LDP server systems portal later on, thereby providing hooks for other client technology.

In order to set the LDP client standard, a decision on device support is required. This in turn will lead to a choice of GUI toolkits and programming environments.

In our perception, only a sufficiently large graphical desktop will reveal the power of the asynchronous application interface (presentation layer). Full mouse and keyboard support are mandatory, the latter especially for the power user. Therefore, notebooks and common PC desktop systems will be the client devices of choice for LDP.

Depending on the language environment, and the messaging and middleware approach on the server side, there is either the

- Java Foundation Classes (Swing) or

- Qt by Trolltech (C++) (free for non-commercial development)

recommended for technology reasons. If handhelds come into question (PDAs) or special visualization and optimization needs arise, e.g. on memory, ergonomy, performance, etc. stripped-down variants of JFC or other toolkits may be of interest:

- swt, the toolkit delivered with the Java SDE Eclipse or
- Qtopia and Qt Embedded, a variant of Qt that supports one single code base, from the desktop down to the handheld systems (C++).

4.2.2 Directory Services and Single Sign On

A very important part of LDP is user and access control management. Therefore, the appropriate modeling of organization and user structures will be a key task in the beginning of system analysis and design. This business-driven topic should neither be delayed nor treated generically. For example, the role objects that decorate LDP base types for legal and natural persons likewise, e.g. roles like ‘customer’, ‘contractor’, ‘carrier’, etc. require that access control information is mapped onto process context in order to resolve the credentials of a user according to his particular role.

User managements affects the following areas:

- Organization structure and user identity, primary key/UUID management,
- The roles of users in terms of LDP and in terms of the business processes,
- The credentials, the access and execution rights in LDP, hence the process catalogue and

The administration processes and the management interfaces.

While there are, at present, no further organizational details on RIS and LDP available, we recommend role-based modeling by means of LDAP (lightweight directory access protocol) and the use of the OpenLDAP directory server in the first place. There are C++, Java (JNDI) and other language interfaces available.

Single Sign On is a feature to strive for strategically, although seldomly achieved: the user is authenticated to the platform once (and only once), thereby gaining access to the offerings and other (remote) systems transparently, according to his credentials. Moreover, the user will neither be guided to areas nor become aware of offerings he is not authorized to. This requires means of active configuration, e.g. by a dynamic process catalog. When it comes to integration of external systems, single sign on usually fails because of security-related issues (password distribution) or technical problems (no context passing at interfaces, etc.), hence legitimation keys will also become part of a process context by direct insertion (through activities).

4.2.2 LifeCycle Services

Since business objects exhibit proxy semantics (on behalf of their respective ERP home systems) and will be treated as single logical instances (SLI), there is reference counting and active life cycle management requested by LDP. To some extent, this is related to system garbage collection, e.g. as performed by the JVM, but here instead requires name space isolation and full application control. Never, neither in the case of SLIs, naturally, nor for the process-bound resources, can lifecycle control be surrendered to client code - the client may be killed abnormally thereby leaving the respective server-side resources uncollected.

In addition, and according to the process policies, server-side resource automation will be established in order to manage system resources, e.g. by object eviction patterns, and to keep performance at a maximum level, e.g. by instance pre-loading, thread pools, connection pooling, etc. Although the base

technology in place, specific extensions will have to be developed since neither CORBA nor Java provide the features as needed for LDP.

4.3 Software Development Environment

In the case of a J2EE /Java development, we recommend the following main software tool set:

- Eclipse, the Java SDE, Open Source
- MagicDraw UML, a moderate priced commercial CASE-Tool
- JBoss (J2EE Application Server)
- JMS (JBoss, or SUN native)
- OpenORB, in the case of a Java-based ORB
- Javadoc (if C++ is not involved) or Doxygen for Documentation
- CVS, Subversion (enhanced CVS) for versioning
- Suse Linux 8.x or Windows 2000Pro

In the case of CORBA/C++ development, either standalone or in combination with J2EE, we recommend the following tools in addition:

- GNU Pro SDE (RedHat) or Sniff+ (Commercial), or Emacs (usually sufficient)
- ACE (for asynchronous networking)
- TAO (The CORBA implementation on top of ACE)
- Qt (C++ GUI Toolkit)
- Doxygen
- Suse Linux 8.x using g++, Windows 2000Pro (requires Visual C++ 6.0)

For rapid protopying, scripting, and for the development of pre-production admin tools, Franzius-Institut recommend the interpreter language Python (www.python.org). Python is a full-featured OO language available on major operating systems and has a rich class set for common system programming tasks.

4.4 Production Environment

The combination of messaging and application server technology will reveal a maximum of flexibility and scalability. Cost-effectiveness is at best achieved by Intel PCs running Linux (out of the box). They will scale for most of the scenarios that will arise in the near future, until memory limits for PC mainboards become an issue. Since the software will be entirely portable, the next stages in terms of MIPS power and total costs of ownership/operating are:

- load-balanced clustering using Intel hardware (running Linux),
- MP RISC systems running Unix OS that support HA and dynamic partioning, e.g. AIX, Solaris and
- IBM z-Series, parallel sysplex technology running in OS/390 mode or providing any number of virtual Linux machines (Linux/390). This is a setup for dedicated operating at a global business level.

5 FASTCORS: An example for communication between port information systems and ships for the fast transport of containers outside regular services using LDP

5.1 Introduction

This chapter explains how to integrate LDP, a port information system and an onboard application of a ship at CORBA level using the Common Object Service CosNotification (OMG, 2002) for the first LDP use case: FASTCORS.

FASTCORS, as an example of a LDP use case, is an open IT platform for using RIS information for individual and reliable transport of containers outside regular services. The platform can be used by single skippers (by using their onboard application like LDP Ship Client), shipping companies, fleet managers, terminals (using BIDIS or by adding a supplier/consumer unit to their ERP system), agents and end users to combine RIS data and their in house transport information to offer and/or seek for containers to be transported outside the regular line services.

Different event types (OFFER; CANCEL, UPDATE etc.) were implemented for this business case, to bring the platform in a pre-usable state.

Above that FASTCORS can serve as an example for any Java application seeking to interface with LDP by means of event messaging.

For the code examples, the open source implementation JacORB is applied. It provides the CORBA runtime and the IDL compiler to be used for Java code. In order to publish or subscribe events from LDP, an application will only have to provide an implementation of standard interfaces. These interfaces are StructuredPushConsumer and StructuredPushSupplier as defined in CosNotification. Accordingly, an object willing to receive events from LDP will implement StructuredPushConsumer while an object willing to publish will implement StructuredPushSupplier 21. An object may also act concurrently in both roles, consumer and supplier. The necessary Java programming and the use of the CORBA interfaces is explained.

For further details on CORBA refer to Henning & Vinoski (1999) and Brose et al. (2001) in particular for Java programming with JacORB.

5.2 Installing the Java ORP

JacORB (www.jacorb.org) was chosen for its robustness and compatibility with TAO. There are several other ORBs available, both commercial and open source. Generally, the examples should work with any other ORB. Download the JacORB distribution and follow the installation instructions.

Please note that the ORB does not have to be rebuild. It is sufficient to generate the startup and IDL compiler scripts by running 'ant jaco' and 'ant idlcmd' as stated there.

Then, to make the ORB work with the JDK it is necessary to tell the Java runtime environment (JRE) to use JacORB instead of its own built-in ORB. This is achieved by copying the file orb.properties from the JacORB distribution to the directory where the JRE looks for property files. The actual location depends on how the JDK is installed. On UNIX it can be the user's home $HOME. orb.properties can also point to jacorb.properties which is installed in $HOME/etc/jacorb.properties. Add the JacoORB

---

[21] There are also batch interfaces for structured events available (*SequencePushConsumer* and *SequencePushSupplier*). These have not been used in LDP as yet.

class libraries (jar files) to your Java CLASSPATH environment. Then adjust the properties that affect the interplay with TAO and LDP:

In the file jacorb.properties locate the variable

ORBInitRef.NameService = corbaloc:iiop:w.x.y.z:2809/NameService

and replace w.x.y.z:2809 with the IP and port number of the TAO Naming Service instance running on the network. This tells JacORB to use by default the TAO Naming Service where objects from LDP have been registered.

In order to start a JacORB application, the 'jaco' script can be used instead of the classic ‘java’ command. It supplies the JVM with the appropriate parameters. Alternatively, for instance when starting from within Eclipse, the parameters used in the 'jaco' script can be used as is for the settings of the java execution command line arguments in Eclipse.

5.3 Building a Supplier for Client Applications

The following example demonstrates necessary CORBA and Java programming to receive (consume) events from LDP.

First of all, a typical set of CORBA and Java packages is imported:

```
import org.omg.CosNotification.*;
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyFilter.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.PortableServer.*;
import org.omg.CORBA.Any;

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

Then a Java class is declared, e.g. PushConsumer, to extend the standard interface from CosNotification. Using the extends relationship, the class will implement or override the implementation given by StructuredPushConsumerPOA. This particular object implementation has already inherited the interface from CosNotifcation and uses the Portable Object Adapter (POA) to register itself with the CORBA runtime:

```
public class PushConsumer extends StructuredPushConsumerPOA
 {

        // To stay aware of connections, the consumer holds some object references in the class field.
        // This is not necessary for the most simple implementation but of course needed
        // for active connection management ( connect/disconnect, suspend/resusme)

        protected static org.omg.CORBA.ORB orb;
        protected EventChannel channel=null;
        protected ConsumerAdmin consumerAdmin=null;
        protected StructuredProxyPushSupplier proxyPushSupplier=null;
        // just to demonstrate output of events to graphical components

        protected static TextArea a = null;
```

A sample init method is defined to create the consumer object. This can also be done elsewhere or in other ways, preferably in a constructor. The method takes command arguments for the ORB, the channel's name (topic) and a string that may hold an ETCL expression for event filtering. To give an implicit example, a filter object is created along and registered with the event channel.

```
public void init( String[] args, String cn, String tcl_expr)
 {
   try{
        // First of all, start the ORB runtime

        orb = org.omg.CORBA.ORB.init(args, null);

        // Get the Naming Service from the environment. Here, the IOR of the TAO
        // Naming Service is found and returned

        NamingContextExt nc =
        NamingContextEx-
        tHelper.narrow(orb.resolve_initial_references("NameService"));

        // Next, look up the event channel's object reference

        channel = EventChannelHelper.narrow(nc.resolve(nc.to_name(cn)));

        // Then ask the channel for a consumer administration object

        InterFilterGroupOperator ifgop  = InterFilterGroupOperator.AND_OP;
        org.omg.CORBA.IntHolder adminId = new org.omg.CORBA.IntHolder(0);

        consumerAdmin = channel.new_for_consumers(ifgop, adminId);

        // If a filter expression is supplied, create a filter

        if ( tcl_expr.trim().length() != 0)
             add_filter(tcl_expr);// see add_filter method below

        // followed by the initial subscription (may be changed later by calling
             // change_subscription (..)

        EventType added[]   = new EventType[1];
        EventType removed[] = new EventType [0];
        added[0] = new EventType ("*", "*");

        try{
             consumerAdmin.subscription_change(added, removed);
        }
        catch (Exception e){
             System.err.println( "Error: " + e );
             e.printStackTrace( System.err );
        }


        // Then, get the root object adapter (default POA)

        POA poa = org.omg.PortableServer.POAHelper.narrow (
             orb.resolve_initial_references("RootPOA"));
```

```
// Create a servant for the consumer object

StructuredPushConsumer structuredPushConsumer =
(StructuredPushConsumer)new PushConsumer()._this(orb);

// Get the Proxy Supplier for the consumer

ClientType clientType = ClientType.STRUCTURED_EVENT;
org.omg.CORBA.IntHolder proxyId = new org.omg.CORBA.IntHolder (0);
ProxySupplier proxySupplier = null;
try{
proxySupplier =
     consumerAdmin.obtain_notification_push_supplier (clientType,
           proxyId);
}
catch( AdminLimitExceeded e ){
     System.err.println ( "Error: " + e);
     e.printStackTrace( System.err );
}

proxyPushSupplier =
     StructuredProxyPushSupplierHelper.narrow(proxySupplier);

// Finally, the consumer is connected to its Proxy Supplier

proxyPushSupplier.connect_structured_push_consumer(
     structuredPushConsumer);

// Activate object
poa.the_POAManager().activate();

}
catch ( Exception e ) {
     System.err.println( "ERROR: " + e );
     e.printStackTrace( System.err );
}
System.out.println("Normal Termination...");
}
```

So far, event consumer object has been created and activated. This is where CORBA programming is practically done.

Next comes the operation ***push_structured_event*** to be implemented by the application programmer.

The operation ***push_structured_event*** is invoked automatically and asynchronously by the ORB whenever an event is delivered from the channel to the consumer. Therefore, the developer will only have to fill in the code that handles the event after being actually received.

In the example, some information from the event header is displayed and inspected. If the event holds a certain named type (“OFFER”), the appropriate object is constructed with the respective values held by the event.

```
public void push_structured_event (StructuredEvent event){

     try {
```

```
        // Access the event header …

        System.out.println ("Event Domain: " +
             event.header.fixed_header.event_type.domain_name);
        System.out.println ("Event Type  : " +
             event.header.fixed_header.event_type.type_name);

        // Look at the body of the event …

        Property properties [] = event.filterable_data;
        if (event.header.fixed_header.event_type.type_name.equals("OFFER")
             &&  properties[3].name.equals("Transport")){

             Fastcors.TransportHelper th = new Fastcors.TransportHelper();

             Fastcors.Transport t =  th.extract(properties[3].value);

             System.out.println("*** Transport Data  " + "\n"
                         + "\tContainer Nr.: " + t.ct_number + "\n"
                         + "\t        Länge: " + t.ct_length + "\n"
                         + "\t      Gewicht: " + t.ct_weight + "\n"
                         + "\t        Preis: " + t.price     + "\n"
                         + "\t Pickup@Hafen: " + t.port_pickup + "\n"
                         + "\t   Dest@Hafen: " + t.port_dest + \n\n");

             String display= new String("Container: " + t.ct_weight +"\n");
             a.append(display);
             }
        }
        catch ( Exception e ) {
             System.err.println( "ERROR: " + e );
             e.printStackTrace( System.err );
        }
}
```

It is worth noting that the classes Fastcors.Transport and Fastcors.TransportHelper were compiler-generated from IDL definitions of struct Transport in the Fastcors name space (module). No further coding is required.

At this point, a Java application is ready to receive and process events from LDP. Other operations can be left untouched or unused in the first place, in particular those defined for connection management.

Active connection management requires the following operations:

```
public void suspend (){
        try{
             proxyPushSupplier.suspend_connection();
        }
        catch  (Exception e)
        {
             System.err.println("ERROR: " + e);
             e.printStackTrace(System.err);
        }
}
public void resume  (){
        try{
```

```
        proxyPushSupplier.resume_connection();
        }
        catch  (Exception e)
        {
             System.err.println("ERROR: " + e);
             e.printStackTrace(System.err);
        }
 }
```

The *suspend* operation still holds the line but makes the event channel start buffering. Upon *resume*, the channel will deliver all buffered events to the consumer until its internal queue is empty again.

In a production system, this feature is usually exploited only for special channels, e.g. logging services, and restricted by high-water marks to avoid excessive resource consumption. Typically, event channels define certain QoS properties at start up time.

The *subscription_change* operation was already shown during the initialization of the consumer object. It allows to update the subscription of events (family of events to receive or exclude). A subscription is a high-level sorting definition that only affects the event header, in particular the fields event domain and event type.

```
public void subscription_change (EventType added[], EventType removed[]){}
```

To give a simple demonstration of how the consumer object can be used, a main method is provided:

```
public static void main( String[] args ){

      try{
             BufferedReader br=
             new BufferedReader( new InputStreamReader(System.in));

             System.out.println("Enter channel name as known in
             LDP Naming Space:”);
             String cn= br.readLine();

             System.out.println("Enter optional TCL expression
             (event filtering):");
             String tcl= br.readLine();
             tcl.trim();

             // Create and init the consumer object
             PushConsumer c= new PushConsumer();
             c.init(args, cn, tcl);

             // Create a simple Text Widget for displaying event content
             Frame f= new Frame();
             a = new TextArea(1,1);
             f.add(a, BorderLayout.CENTER);
             f.pack();
             f.show();

     }catch (IOException e){
             System.err.println("Error reading Console Input.");
     }
  }
}
```

The *add_filter* method is a helper that registers the filter object . It may serve as a starting point for advanced applications that dynamically add and remove filter objects from the channel. The group operater used before *init* defines how the channel treats a combination of filter objects. In this case, a logical AND of expressions is performed.

```
InterFilterGroupOperator  ifgop  =
              InterFilterGroupOperator.AND_OP;
```

Here is the code of the helper function that plugs a filter object into the channel:

```
public void add_filter(String expr) {
     // Get the default filter factory
     FilterFactory filterFactory = channel.default_filter_factory();
     Filter filter = null;

     try {
             filter = filterFactory.create_filter("TCL");
     }
     catch (Exception e){
             System.err.println( "Error: " + e );
             e.printStackTrace( System.err );
     }

     // Setup filter object
     ConstraintExp exp[]   = new ConstraintExp[1];
     EventType eventType[] = new EventType [0];
     exp[0] = new ConstraintExp (eventType, expr);
     try {
             ConstraintInfo info[] = filter.add_constraints (exp);
             consumerAdmin.add_filter(filter);
     }
     catch (InvalidConstraint ex) {
             System.err.println( "Error: " + ex );
             ex.printStackTrace( System.err );
     }
}
```

5.4 Building a Consumer for Client Applications

Programming a supplier is almost the same procedure as for the consumer and thus straight forward. First of all, canonical Java packages are imported:

```
import org.omg.CosNotification.*;
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyFilter.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.PortableServer.*;

import org.omg.CORBA.Any;

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

Like the consumer extends *StructuredPushConsumerPOA*, the supplier extends a dedicated object with name of *StructuredPushSupplierPOA*.

```
public class PushSupplier extends StructuredPushSupplierPOA
{
     protected static PushSupplier s= null;
     protected static org.omg.CORBA.ORB orb = null;
     protected EventChannel channel = null ;

     protected StructuredProxyPushConsumer pushConsumer = null;

public void init (String[] args, String cn)
       {
     try{
             orb = org.omg.CORBA.ORB.init(args, null);
             NamingContextExt nc = NamingContextExtHelper.narrow(
             orb.resolve_initial_references("NameService"));

             channel =  EventChannelHelper.narrow(nc.resolve(nc.to_name(cn)));

             // Get an admin object
             SupplierAdmin supplierAdmin = channel.default_supplier_admin();

             // Create the supplier servant
             StructuredPushSupplier structuredPushSupplier =
                    (StructuredPushSupplier)new PushSupplier()._this(orb);
             ClientType ctype = ClientType.STRUCTURED_EVENT;
             org.omg.CORBA.IntHolder proxyIdHolder = new
                    org.omg.CORBA.IntHolder();

     try{
             pushConsumer = StructuredProxyPushConsumerHelper.narrow(
             supplierAdmin.obtain_notification_push_consumer(ctype,
                    proxyIdHolder));
     }
     catch (AdminLimitExceeded ex)
             {
                    System.err.println("Could not get consumer proxy,
                    maximum number of proxies exceeded!");
                    System.exit(1);
             }
     // Connect push supplier
     try{
                    pushConsumer.connect_structured_push_supplier(
                    StructuredPushSupplierHelper.narrow( structuredPushSupplier ));
     }
     catch( Exception e ){
                    e.printStackTrace();
     }

     // Run the ORB
     POA poa = org.omg.PortableServer.POAHelper.narrow
     (orb.resolve_initial_references("RootPOA"));
     poa.the_POAManager().activate();
}
catch ( Exception e ){
```

```
     System.err.println( "ERROR: " + e );
     e.printStackTrace( System.err );
 }
System.out.println("Normal Termination...");
}
```

Now the supplier object is ready to deliver events into the channel. A *send* method can be defined as a wrapper for the standard operation *push_structured_event*. The wrapper takes a StructuredEvent and pushes it to the channel.

```
public void send( StructuredEvent ev)
{
     try{
     if( ! pushConsumer._non_existent()){
             System.out.println("pushConsumer " + pushConsumer + " existing.");
             pushConsumer.push_structured_event(ev);
     }
     else
             System.err.println("pushConsumer " + pushConsumer + " missing.");
     }
     catch( org.omg.CosEventComm.Disconnected d ){
             // can be ignored ..
     }
}
```

Before pushing events, an application should check that the consumer proxy is connected. Otherwise, the exception *CosEventComm.Disconnected* is thrown. Experience shows that exceptions from connection management usually do not affect the robustness and stability on either side. Therefore, depending on the application, they can eventually be left unhandled any further.

The following *main* method demonstrates the use of the supplier object:

```
public static void main( String[] args )
 {

     try{
             BufferedReader br= new BufferedReader( new
                    InputStreamReader(System.in));
     System.out.println("Enter channel name as known in LDP Naming Space:”);

     String cn= br.readLine();

     s= new PushSupplier();
     s.init( args, cn);
```

Next, we create a StructuredEvent for demonstration purposes.

```
     StructuredEvent ev = new StructuredEvent();
```

As an example, we assign this event the (imaginary) “FASTCORS” vertical industry domain and the event type “OFFER”

```
     EventType type= new EventType("FASTCORS","OFFER");
     FixedEventHeader fixed = new FixedEventHeader(type,"");

       Property variable[]= new Property[0];
```

```
ev.header= new EventHeader(fixed, variable);
```

Then, we can load the desired information into the event body using name/value pairs as given by CORBA::Any. In the example, the event is loaded with 4 entries.

```
// Allocate body section
ev.filterable_data = new Property[4];

// First entry
Any a0 = orb.create_any();
a0.insert_string("*");
ev.filterable_data[0] = new Property("Receiver", a0);

// Second entry
Any a1 = orb.create_any();
a1.insert_string("BIDIS Hannover");
ev.filterable_data[1] = new Property("Sender", a1);

  // Third entry
Any a2 = orb.create_any();
a2.insert_long(1234567);
ev.filterable_data[2] = new Property("LDP_ID", a2);

  // Fourth entry (here an object is loaded )
Any a3 = orb.create_any();
Fastcors.TransportHelper th= new Fastcors.TransportHelper();
Fastcors.Transport t= new Fastcors.Transport(
        "CT9987XXX","14.5m",16000,"5000 EUR",
        "Rotterdam","Antwerpen");
th.insert(a3, t);
ev.filterable_data[3] = new Property("Transport",a3);

// End of loading

// Create empty body to finish event definition (important !)
ev.remainder_of_body = orb.create_any();

// Now send the event to the channel
s.send(ev);

orb.run();
}catch ( IOException e ) {
        System.err.println(  e );
        e.printStackTrace( System.err);
}
 }
}
```

Please note that instead of creating many suppliers for one particular channel, a supplier server can be used to free the channel from frequent connection management.

By means of consumer and supplier units, the technical issues of application integration have been solved, as can be seen by CSL.DB integration. It is up to the business experts now to define the business processes, business events (data resources) and state transitions.

5.5 Business Process Modeling for FASTCORS

To demonstrate communication and integration of client technology (ship application onboard – LDP Ship Client) and port information systems (here BIDIS as an example of a ERP system running on an inland terminal) with LDP, a real-time ad hoc contracting scenario was specified for FASTCORS, Figure 5-1.

The BIDIS PortClient and the LDP Ship Client are used by the contracting parties while **LDP acts as real-time middleware and concurrent broker of business transactions**.

The FASTCORS process realizes a simple 2-phase-commit proctocol at business level. To keep it simple for demonstration purposes, alternative process outcomes and certain business exceptions have been omitted from the example.

First of all, a PortClient posts a transport offering to LDP which in turn can be received by consumers belonging to the transport domain. In order to receive offerings, the respective consumers subscribe to a dedicated topic, the FASTCORS event channel.

A skipper running the ShipClient application will then receive transport offerings in real-time and eventually select one or more containers he is willing to pick up and deliver for the indicated price. By drag 'n' drop-selecting the respective boxes in the ShipClient application, the skipper confirms interest into the transport and can launch commit events that are transferred to the respective port by LDP. The PortClient may accept one of the incoming bids and return a COMMIT event by which a peer-to-peer contract is established on the fly. Right after, a CLOSED event is submitted to the channel indicating that the particular offering has been closed as a public business case.

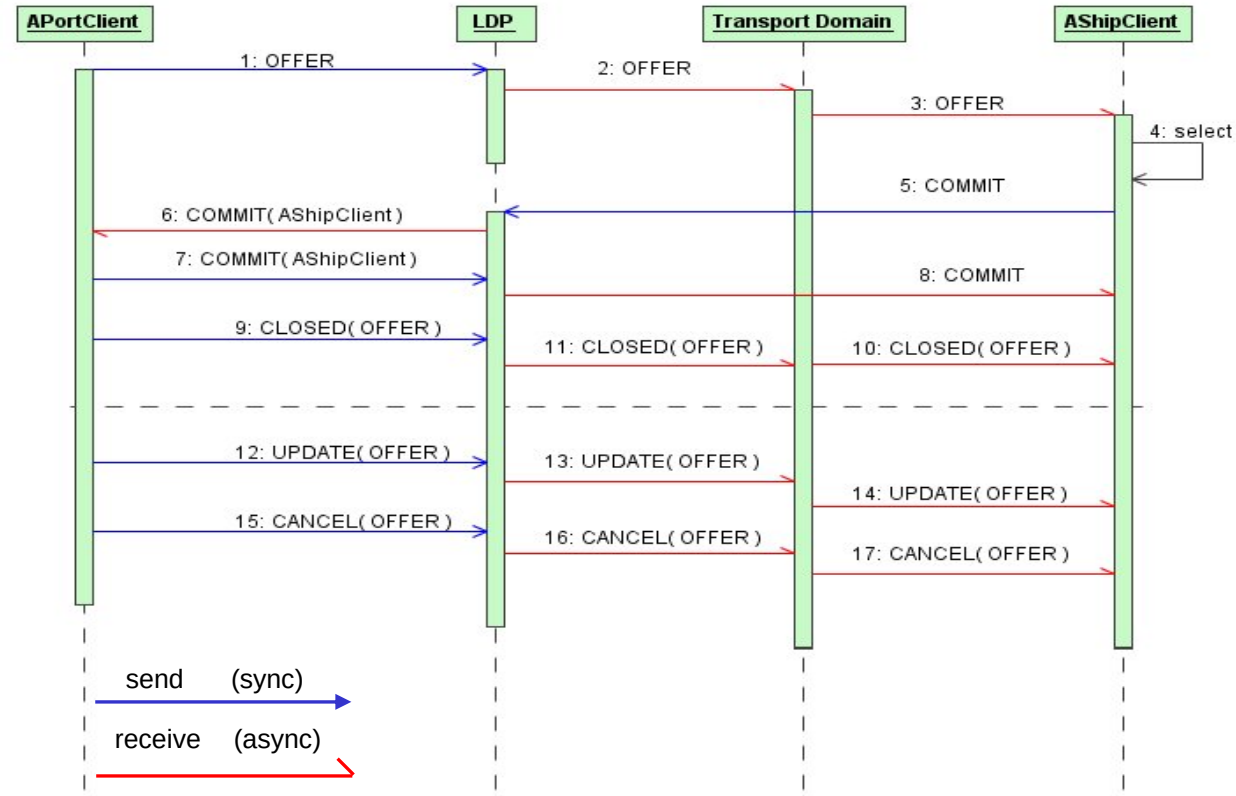

Figure 5-1: FASTCORS Business Process

In the scenario, the port also has the option to publish UPDATE events for prior transport offerings, e.g. to make pricings more attractive to skippers. A recall of offerings is supported through the CANCEL event.

Business rules that affect dispatching, secure routing to dedicated consumers, and selection of events according to their content, etc. can be defined by means of event filtering expressions.

Depending on the application scenario and organizational constraints, filter objects can be either centrally managed by LDP or deployed with the remote application (here PortClient or ShipClient). Nevertheless, event filtering as defined by filter objects, whether they are local to LDP or created remote on the network, is performed server-side in collocation with LDP event channel and thereby ensuring high-performance networking.

5.6 LDP Integration in the Port Information System

To demonstrate a first example of LDP integration to existing business processes and make the real-time capabilities of LDP visible to users and stakeholders, FASTCORS was selected as a first use case covering most event types commonly used in offering and contracting scenarios between different parties offering and performing transports.

To provide a run time environment for FASTCORS demonstration, a consumer und supplier unit was added to the Port and Information System BIDIS22 as described above.

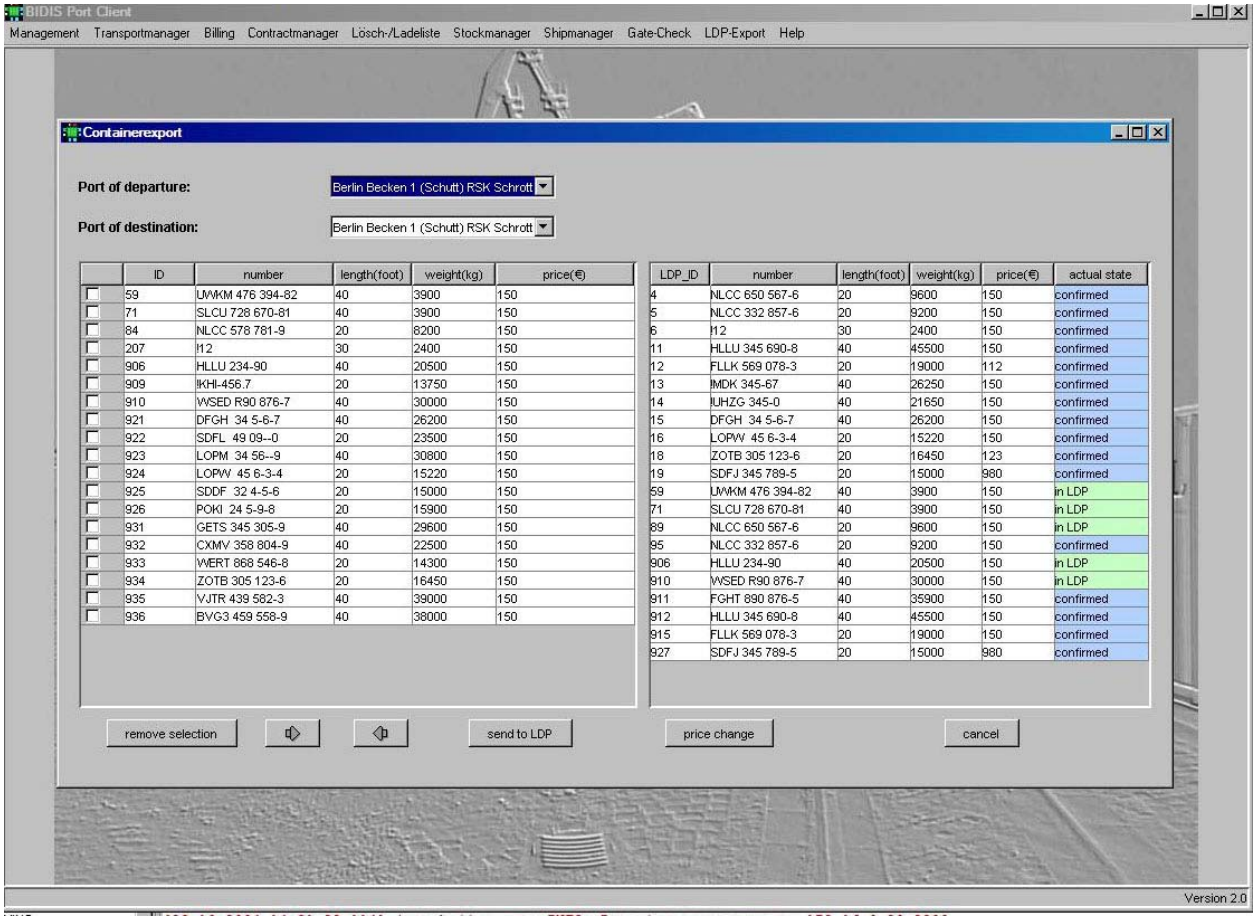

Figure 5-2: LDP Integration in the Port Information System BIDIS

22 BIDIS: „Binnenhafen Informations- und Distributionssystem“. Prototype version of a port information system. Developed by Franzius-Institut for Hydraulics, Waterways and Coastal Engineering, All Rights reserved.

Additionally a LDP connectivity tool, Figure 5-2, was added in BIDIS to allow the port offerings (OFFER), changings (UPDATE), closings (CLOSE) and cancellation (CANCEL) of container transports outside regular services.

The central BIDIS database of all container transports is shown to the transport manager. He selects containers to be offered and sends them to LDP. In real-time the skipper registered to the event channel “OFFER” gets a notification and can answer on this.

The transport manager in the port can change, cancel or close his offer. Having full control all the time, contracted container transport are marked in the BIDIS database and foreseen as a special transport outside the known services. Their schedule is handled different. Loading time and working hours of staff is arranged separate.

In this way, the inland terminal gets a powerful instrument to handle transport without known services and bring selected transports to the free market.

5.7 LDP Integration in the onboard Application of the Ship

The integration of port information systems via LDP with onboard applications is also a great opportunity for independent skippers seeking for freight.

In such a market outside regular services, dedicated to containers to be delivered in short time and/or other difficult conditions, independent skippers (so called “Partikuliere”) have a chance to come back in business again – also in the container market, which is in fact a business of big ship companies so far.

To demonstrate the possibilities of such a contract situation in practice, Franzius-Institut has developed a simple, but efficient onboard application called LDP Ship Client, Figure 5-3.

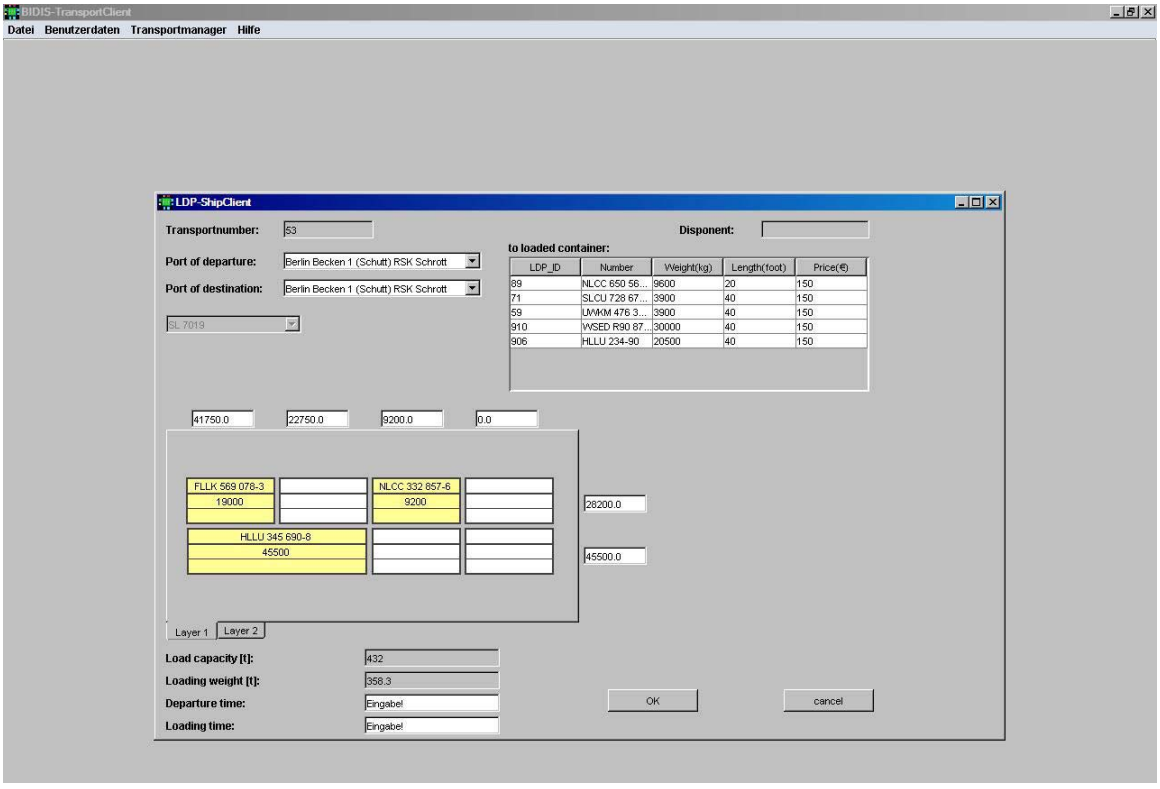

Figure 5-3: LDP Ship Client Application

On the right, the skipper gets all offers dedicated to the event channel he is registered to. In this specific case that are offers of container transports outside regular services.

He picks the transport (container) he wants to do by putting it into his stowage plan and checks for loading capacity (if several containers are loaded etc.) and stability. If everything is okay, he makes the contract with the offering port by saying “OK”. If another skipper did the contract before, he gets a CLOSING event before. The container is deleted from his stowage plan.

If the port takes the transport from LDP to put it in again to a regular service, this is shown in real-time in the container list the skippers see in the right.

The developed application can be driven to a full trip planning instrument adding some functions for voyage planning, accounting services and time schedule services.

The application can also be added to existing applications, to give LDP integration as a starting point for further application development.

5.8 Conclusions

Franzius-Institut has developed CosNotification event supplier and consumer units required to link port information systems (as an example of an ERP system) and other client applications with LDP. They provide enterprise application integration at a very high level and may serve as an example for other Java applications.

It is also possible to deploy these units in web browser environments. For instance, an applet can provide the runtime for asynchronous and stateful communication between LDP and the stateless browser.

Depending on the needs of the application, the units can be extended with active connection management, dynamic changes in event offers and subscriptions, etc..

Regarding the tempting simplicity of the Java programming environment, one must still keep in mind that unmanaged resource utilization can become a problem for a production system. For instance, when the garbage collector of the JVM starts to clean up, unpredictable and long-lasting locks can block the application. Therefore, object allocation and event processing techniques within the application should be considered carefully.

For very large numbers of suppliers/consumers using LDP, an operator of event channels should also consider scaling strategies like federation (for consumers) and proxy servers (for suppliers). Both concepts of scaling strategies have already been demonstrated in LDP during the FASTCORS demonstration in Rotterdam.

6 Development Process of LDP

6.1 Development using CORBA (TAO) on top of ACE

During LDP development and FASTCORS implementation we have envisaged no problems using CORBA standard or other components recommended for LDP development. Used components have proven their applicability to industry projects by means of functionality and robustness, which was known to us before implementation started, but often negated by project partners.

The integration of existing applications to LDP environment means implementation of supplier/consumer units using CORBA libraries. This task can be calculated as a two day job for an experienced Java or C++ programmer using the examples shown in this report.

Implementation of business processes on top of LDP is already possible and already demonstrated by FASTCORS.

In practice, the definition of requirements, metrics, work flows and exception handling will take much longer then pure system programming using LDP messaging backbone and business process manager.

Flexibility of the platform guaranties the transfer of events/data also in the future, when boundary conditions, ERP systems and law are changing rapidly as they are doing just now.

6.2 Future Development of LDP

The development of LDP so far was conducted according to the methodologies of business process analysis (BPA) and object-orientation (OOA/OOD), hence iteratively and incrementally, following three main threads in the first phase:

- Evolutionary prototyping of LDP core system components
- Comprehensive requirements and business analysis
    - Use Cases
    - Business Process Types, Design of Activities
    - Business Base Types
- Setup of a the RIS/LDP bridge (Messaging server)

In terms of the LDP core system, the generic LDP business abstractions (users and access control, sessions, processes, activities, etc.), the asynchronous process engine, server-side life cycle automation, and high-level system interfaces were designed in one turn with the asynchronous LDP client. This will reveal the specific system components design and the basic communication patterns of LDP - at a system framework level (the baseline).

In the following, metrics and stress tests have to be applied by means of technical use cases and synthetic processes. The system behaviour, its quantitative and qualitative runtime characteristics have to be recorded and judged according to empirical standards of enterprise systems, e.g latency and response time values.

Naturally, the next development cycles have to introduce significant low-level improvements before going over to a functional extension of the system framework.

Likewise, the RIS/LDP messaging system was established right from the beginning. The neartime cache, and especially the real - time mapping of entities into the LDP process space constituted mis-

sion-critical features. In their role of continuous data feeders, they will also belong to any test bed setting.

Further development has to consider the functional building blocks as needed. At a certain point of time, e.g. when there is a demand for full-featured demonstrators, the process application workbench, the process type repository, etc. will be addressed gradually. These buildtime tools will directly support the application developer by means of visual programming, code generation, and automatic LDP deployment procedures.

For the entire software lifecycle process of LDP, common conventions of industrial-strength software engineering have to be applied. The following practices are mandatory:

- UML applied to all deliverables from analysis and design,
- Low volume/high quality documentation,
- Full versioning of deliverables, build and release management,
- A test bed for release verification, integration, and demonstrators (gradually growing) and
- CASE tool-based Round-Trip Engineering.

For the overall success of LDP it is mandatory, that Use Cases are well defined to exploit all functionalities and strength of the platform. Future project should calculate much more time on evaluation of these Use Cases and should place industry experts for this on every work package involved in software engineering to make sure that architecture is defined by Use Cases and not vice versa.

7 Recommendations for Acting

7.1 Management of Requirements

Requirements, Figure 7-1 are usually inspected at three distinct levels: Business, User, and Functional. There are as well various so called non - functional requirements.

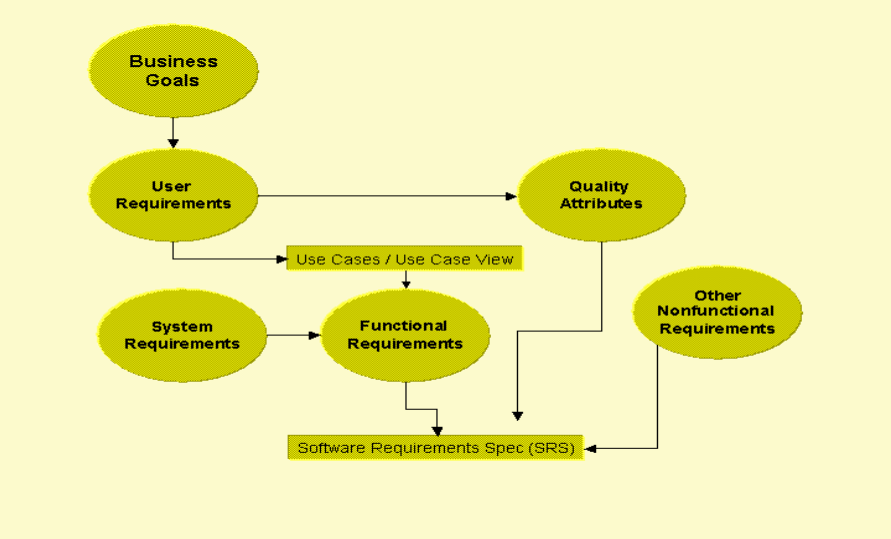

Figure 7-1: Input to Requirements Specification

Business requirements represent high-level objectives of the parties requesting LDP, the vision and the scope. This has been defined already - although not yet in great detail.

User requirements describe tasks the users must be able to accomplish with the platform. Usually, these tasks are captured by means of Use Cases, scenario descriptions, and coarse-grained processes (activities and transitions)23. Unfortunately, this category is lacking substance as yet, wherefore at present only a generic functional specification can be derived.

Functional requirements define the software functionality that must be build into LDP to enable users to accomplish their tasks, thereby satisfying the business requirements. Thus, a set of related functional requirements is called a feature of LDP.

Non - functional requirements cover various aspects, e.g. regulations, standards and interfaces to which the system must conform, performance, design and implementation constraints, etc. Although obvious in the first place, non - functional requirements tend to conceal other requirements and have to be examined carefully. After inspection, they sometimes turn out to be vital functional requirements.

Along with system requirements, the three levels will melt into a software requirements specification (SRS) for LDP. The SRS covering LDP in total is not yet available. Without having the Use Case document in the first place, it is impossible to derive a sufficient part of the SRS since user requirements (Use Cases) contribute a root node in the requirements hierarchy:

Compared to business systems at production level, LDP is still at an early and experimental stage. Therefore, it was commendable to establish a lightweight but high-quality requirements management process for the project - with initial emphasis on the Use Cases.

---

[23] UML defines the semantics and the graphical depictions of Use Cases, Processes, etc. Informal, template-like textual Use Cases and scenario descriptions are also common.

The resulting Use Case Model must hold

- the actors , both human and machine type (neglected frequently)
- the Use Cases and Event Chains at different stages and levels of granularity
- Extends- and Uses-relationships among Use Cases
- core sequences and carrier processes, options, alternatives, and outcomes

Engineering the requirements is the iterative and incremental process of developing the SRS. Both informal and semi-formal techniques are available and can be adopted from the industrial software development process (e.g. Wiegers, 1999 and Jackson, 1995). The actual choice will depend on the size, the timeline, and the structure of the project in terms of co-operative software development.

The first management step will be to define the methods and deliverables forming the project's common requirements development procedure, to negotiate them, and to bring in commitments of all project partners.

Franzius-Institut recommends a combination of informal and UML-based techniques to capture, model, and track requirements, especially Use Cases and core processes. If CASE tools are applied, in addition, the introduction of a (simplistic) requirements management tool will also facilitate the specification of test cases later on.

Furthermore,

- a project-centric (data) dictionary,
- a conceptual object model (logical view) and
- a glossary

should be gradually developed and introduced into the requirements process, hereby promoting common technical and business understanding among project partners.

7.2 Collect Use Cases

The collecting, the informal (textual) recording, and the semi-formal notation (UML) of Use Cases must start immediately. This is preliminary to the modelling and optimization of business and information processes that constitute the true goals of LDP.

It is not recommended to enforce the use of UML at this particular stage. Most humans, especially those stakeholders not experienced in software engineering methodologies, will describe their use case at best by means of writing and informal drawings. For these groups a textual use case template can be provided and may facilitate the process.

7.3 Establish the Process View

Along with the requirements analysis, the business process view should be developed. Whereas the Use Case View depicts the application scenarios for each actor, the Process View depicts the decomposition of activities into workflow graphs and their state transitions - to be implemented on top of LDP. Activities define a significant unit of work at business level (not at algorithmic level) such that, after completion of each activity, the process is making progress until successfully finished, finally.

UML activity diagrams should be used to describe the processes. Furthermore, the flow of resources among activities (object flow), the constraints, and preconditions of activities have to be modelled gradually.

7.4 Develop a System of Logistical Business Types

7.4.1 Business Base Types

Look for common business base types in the logistics domain - but only as far as processes in LDP are concerned with them. A subset of base types is common to any enterprise system, e.g. Date, Time, Money (Amount, Currency), etc. Some base types will be specific to the logistics domain, e.g. enumerations holding goods classes and goods types.

Frequently, business base types qualify for immutable semantics since they often specify one particular state of an object aggregate. It is also beneficial to have business base types designed at object level, and not as flat data items (objects may serialize/flatten themselves as needed). If for instance within one activity, the user changes the currency from EUR to USD all other objects participating in the process could catch the event and switch accordingly.

Depending on the usage context, and especially because of the holistic, transient and life cycle-managed proxy world of LDP, some of the immutable base types also qualify for reference semantics (singletons), thereby reducing garbage collection metrics and allowing for a static, high performance in-memory region of LDP business types.

Moreover, starting with the business base types, further modeling will lead to common analysis patterns in the logistics domain, e.g. typical trading scenarios, negotiation patterns, real - time information procurement patterns, etc. This is the main application area for business-related OOA/OOD and will reveal extraordinary value for LDP development.

7.4.2 Transform of RIS Deliverables

It was already assumed that incoming messages from RIS may qualify for a business object transform when picked up by LDP user processes. This step can be further investigated when the top level Use Cases have been captured.

Nevertheless, from a technical and business perspective, it is vital to know the semantics, the low-level structure and the codings of messages delivered by RIS as soon as possible since LDP will need to define a dynamic mapping between its common process resource space and the entities coming in over the RIS/LDP bridge. For example, how will they classify/identify, what is their grammar for de-serialization, etc ?

7.4.3 Define Metrics

Metrics are applied to derive estimates for system dimensioning, system management, to determine performance, throughput, to assess ergonomic measures for user interfaces (cascading menue levels, entry fields), and many more.

Common metrics specify the number of concurrent users, transactions, processes, object instances, threads, etc. Other metrics characterize business-related measures, for instance the actual amount of user activity required and the mean time spent to successfully finish a contracting process.

In sum, metrics will help to adjust a system according to practical business needs, and provide the knowledge where to spend optimization efforts.

8 Conclusions

LDP constitutes a development kick-off and delivers key elements of application architecture and system technology qualified for software design.

Herein, LDP can be seen as a large-scale electronic business platform and information broker system for the logistics domain.

In order to provide the logistic user community with real-time/near-time service offerings, the platform must be continuously fuelled with data from RIS and other transport modes. Therefore, RIS systems have to implement supplier/consumer units and redefine their scope.

Therefore, the connectivity between LDP and RIS is treated as a contractual obligation at system level. This particular relationship is addressed by means of a messaging backbone architecture.

The application framework of LDP is, in itself, based on a model-driven approach since there is a variety of Use Cases to expect in the future. A generic process model defines the application layout and the platform-controlled elements that constitute the LDP core system.

In this framework, business process types denote sets of activities to be executed by the user (or the platform) in order to achieve a distinct business goal. The business processes, which in turn are runtime instances created from the respective business process types defined at buildtime, are hosted autonomously and on behalf of the user in a stateful, multi-threaded application server environment. This runtime environment can apply dedicated policies to take full control of server-side resources, e.g. by process lifecycle management. It will allow process execution while being disconnected, context sharing among processes, and asynchronous messaging, thereby providing non-blocking client/server communication.

The LDP client technology is designed to be stateful, asynchronous, and multi-channeled, thereby enabling the client to digest incoming messages while sending requests at the same time, for instance the user in synchronous interaction with one of his business processes.

Accordingly, and to draw a parallel, the technologies and paradigms that qualify for LDP can be compared to those of realtime market data and trading systems in the financial world, e.g. stocks exchange, derivative trade, etc..

Apart from information broking, the business processes to be hosted by LDP will be by far more complex, especially with regard to planning and intermodal contracting scenarios. Therefore, the approach merges selected features of distributed business process systems with the capabilities of real-time/neartime information systems into a new class of responsive e-business systems.

The methodologies and techniques for of LDP development cover business process engineering, object-oriented analysis, design, and programming of concurrent event-based systems, the architecture and object services standards of message-oriented middleware (MOM) and either Common Object Request Broker Architecture (CORBA) or Java 2 Enterprise Edition (J2EE), depending on further decision making dedicated to the technology of the LDP core system.

Functionality, robustness and applicability of LDP were demonstrated for a first use case - a software application to establish fast transport of containers outside regular services (FASTCORS). Therefore, a consumer and supplier unit were incorporated in a port information system sending containers to be transported to the market place. This market place by means of LDP dispatches incoming offers and send them to dedicated users abroad. This was demonstrated by a LDP Ship Client designed to pick containers from a list of offers, showing each his information (weight, price, dimensions etc.) to the interested skipper. For acception, cancellation and updating of an offer, different events were implemented. Stability and applicability of the system was demonstrated on work package meetings and stakeholder sessions.

In order to facilitate further steps towards practical software development, mandatory work items had to be derived - and have to be - from the current COMPRIS material and its predecessors, e.g. INDRIS. These items stress requirements, business abstractions, and the Use Cases in particular.

From an IT perspective, especially in terms of business systems engineering, future projects must perform a shift from the technical views to the business process view, hence focus on the actual business and information procurement goals, how to achieve them by means of processes, and how to further optimize these processes in the light of competitive capacity.

9 References

BROSE, G.; VOGEL, A.; AND K. DUDDY (2001)

Java Programming with CORBA. Advanced Techniques for Building Distributed Applications. 3rd Ed., John Wiley & Sons

FRANZIUS-INSTITUT (2003)

COMPRIS Questionnaire to evaluate User Needs for the Development of Logistics and Transport Applications around RIS Version 1.0 (16/01/2003)

FRANZIUS-INSTITUT (2004)

Final Report "Consortium Operational Management Platform Rriver Information Services" "COMPRIS", Contract No. GRD2/2000/30161

HENNING, M. AND S. VINOSKI (1999)

Advanced CORBA Programming with C++, Addison-Wesley

JACKSON, M. (1995)

Software Requirements and Specifications, Addison-Wesley

KAY, M. (2000)

XSLT Programmer's Reference, Wrox Press

LEYMANN, F. AND D. ROLLER (2000)

Production Workflow—Concepts and Techniques, Prentice Hall

OBJECT COMPUTING (2002)

TAO 1.2 Developer's Guide, Object Computing Inc., St. Louis, MO

OBJECT MANAGEMENT GROUP (OMG) (2002)

CORBA Notification Service, Aug. 2002, Version 1.0.1 formal/02-08-04, www.omg.org

OBJECT MANAGEMENT GROUP (OMG) (2003)

Common Object Request Broker Architecture: www.omg.org

SCHMIDT, D.C. (2002)

C++ Network Programming, Addison-Wesley

SUN MICROSYSTEMS (2003)

Java 2 Enterprise Edition: http://java.sun.com/j2ee/

JBOSS, (2003)

Jboss Application Server Project: http://www.jboss.org

WARMER, J. AND A. KLEPPE (1999)

The Object Constraint Language - Precise Modeling with UML; Addison-Wesley

WIEGERS, K.E. (1999)

Software Requirements, Microsoft Press