TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

D I S S E R T A T I O N

The Design of a Building Model Service

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der Sozial- und Wirtschaftswissenschaften
unter der Leitung von

Univ. Prof. Dipl.-Ing. Dr. techn. Ardeshir Mahdavi
E 259/3 Abteilung Bauphysik und Bauökologie
Institut für Architekturwissenschaften

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Klaus A. Brunner

Matrikelnr. 9626334
Pachergasse 16, 2344 Ma. Enzersdorf, Österreich

Wien, im März 2007

Kurzfassung der Dissertation

Gebäude-Informationsmodelle sind hochauflösende, universelle, integrierte Informationsmodelle physischer Gebäude. Sie enthalten nicht nur geometrische, sondern auch semantische Daten von Baukomponenten und ihren Zusammenhängen. Zweck solcher Modelle ist es, den gesamten Lebenszyklus von Gebäuden von der Konzeption über Betrieb und Wartung bis zum Abbruch zu unterstützen. Gebäudemodell-Dienste sind Softwarekomponenten, die solche Gebäudemodelle betreiben und den Zugriff für Anwendungen ermöglichen.

Bestehende Software-Architekturen für Gebäudemodell-Dienste orientieren sich stark an den Anforderungen der Entwurfsphase und teilweise auch der Errichtungsphase von Gebäuden. Diese Anforderungen ähneln jenen von Versionierungssystemen für die Softwareentwicklung: dabei werden in relativ großen Zeitabständen Daten (potentiell in großen Blöcken) abgefragt oder eingepflegt.

Gegenstand dieser Dissertation ist der Entwurf von Gebäudemodell-Diensten, die sich für die Unterstützung des Betriebs von Gebäuden eignen. Als Referenzanwendung dient simulationsgestützte Beleuchtungsregelung, die ein besonders hochauflösendes und aktuelles Modell erfordert. Das Modell ist daher mit Sensoren und Aktuatoren im Gebäude verbunden, wodurch es laufend aktualisiert wird und für Anwendungen als Schnittstelle zu den Gebäudesystemen dienen kann.

Die Hauptprobleme beim Entwurf solcher Dienste sind 1) die Integration einer breiten Palette verschiedener Datenquellen und -senken, sowie 2) die effiziente Verarbeitung von anhaltend hohen Datenraten, wobei gleichzeitig einfacher, ununterbrochener und rascher Zugriff auf Modelldaten möglich sein muss. Einige Anwendungen (etwa Beleuchtungsregelung) haben sogar „weiche“ Echtzeitanforderungen. Besonderes Augenmerk ist daher auf Performanz, Skalierbarkeit und Modifizierbarkeit zu legen. In dieser Dissertation wird argumentiert, dass bestehende Architekturen von Gebäudemodell-Diensten für diese Anforderungen wenig geeignet sind.

Im Kern der vorgeschlagenen Architektur steht das Objektmodell in einer bitemporalen Hauptspeicher-Datenbank mit persistenter Speicherung. Änderungen an Objekten sind versioniert, was den einfachen und transparenten Zugriff auf die gesamte Modellhistorie ermöglicht. Die Datenbank ermöglicht serialisierbare Transaktionen mit hoher Nebenläufigkeit durch den Einsatz von Multiversions-Synchronisierung auf Basis einer Variante von *multiversion transaction ordering* (MVTO). Transaktionen können für zeitgerechte Aktualisierung priorisiert werden. Die Geschwindigkeit und Skalierbarkeit dieses Synchronisations-Mechanismus ist sehr gut im Vergleich zu einfachen Locking-Ansätzen. Sowohl die Reihenfolge der Datenbank-Aktualisierungen als auch der gespeicherten Ereignisse kann jederzeit wiederhergestellt werden, auch wenn Daten ungeordnet oder verspätet eintreffen. Dies ist möglich, indem Daten auch nachträglich in die Versionshistorie eingefügt werden können.

Der Verteilungsaspekt ist einer der wichtigsten Punkte der Architektur: statt eine herkömmliche Referenzarchitektur (z. B. Client/Server) zu verwenden und ihre jeweiligen Beschränkungen zu umgehen, wird für jeden Prozess jeweils die passendste Lokation bestimmt, wo er ausgeführt werden soll.

Es wird von zwei Arten charakteristischem Verhalten von Anwendungen ausgegangen. Interaktives Verhalten besteht in raschen Folgen von kurzen Schreib- und Lesezugriffen auf verschiedene Objekte, etwa als Reaktion auf Modelländerungen. Interaktive Applikationen benötigen wenig CPU und Speicher, aber raschen Modellzugriff und sofortige Benachrichtigung bei Modelländerungen. Solche Anwendungen werden als Agenten innerhalb des Modelldienstes ausgeführt, um Code möglichst dort auszuführen, wo die Daten sind. Agenten können zur Laufzeit installiert und aktiviert werden, ohne den Dienst zu unterbrechen.

Batch-Verhalten besteht in intensivem, langem Arbeiten mit fixen Modellausschnitten, was typisch für Simulation und andere Analyseaufgaben ist. Schreibzugriffe sind selten. Solche Aufgaben benötigen viel CPU und sind von gleichzeitigen Modelländerungen weniger betroffen. Batch-Applikationen werden daher auf andere Knoten verteilt und arbeiten auf Kopien der benötigten Modellteile. Zusammen mit einer auf *spaces* basierenden Kommunikation ist diese Verteilung sehr transparent und erlaubt einfache Lastverteilung mit einem Minimum an Administration.

Die Kommunikation mit Gebäudesystemen wird über eine externe Infrastruktur für asynchrones Messaging abgewickelt. Der Modelldienst verwendet eine flexible nebenläufige Verarbeitung, um Zielobjekte zu lokalisieren und mit den empfangenen Daten zu aktualisieren. Für interne Kommunikation zwischen nebenläufigen Programmen (z. B. Agenten) wird ein einfache asynchrone Messaging-Infrastruktur zur Verfügung gestellt, die auch mit externem Messaging verbunden werden kann. Code kann zur Laufzeit getauscht werden, um hohe Verfügbarkeit und Modifizierbarkeit zu erreichen.

Abstract

Building Models or Building Information Models (BIM) are fine-grained, multi-faceted, integrated computational models of physical buildings. They capture not only geometric, but also semantic data of building components and their relations. The promise of building information models is to support the entire lifecycle of a building, from conception through operation and maintenance to decommissioning. Building model servers are software systems that maintain such models and provide access to local and remote client applications.

Existing software architectures for building model servers are strongly focused on the requirements of the design phase and, to a lesser extent, the construction phase of the building lifecycle. These requirements resemble those of source-code version control systems, with infrequent but potentially massive check-in/check-out operations on a central model repository.

The subject of this dissertation is the design of building model servers suitable for support of the operation phase of a building. The prototypical application used here is simulation-based lighting control, which requires a fine-grained and up-to-date model of the controlled space and devices, much more so than typical facility management applications. The model is therefore connected to sensors and actuators within the building, keeping itself updated with sensor data and also acting as an interface to building systems for client applications.

The main challenges for such operation-phase building models are 1) the integration of a wide range of different data sources and endpoints, and 2) the efficient handling of high sustained rates of incoming data while ensuring simple, uninterrupted, low-latency access to model data. Certain client applications (such as lighting control) even have soft real-time requirements. Great emphasis must therefore be placed on performance, scalability, as well as modifiability. This dissertation argues that existing model server architectures are not suitable to ensure these quality attributes, particularly because they are not designed for high concurrency.

At the core of the proposed architecture, the building object model is implemented in a bitemporal main memory database with persistent backing storage. Changes to model objects are versioned, allowing simple and transparent access to a full model history ("time travel"). The database ensures serializable transactions with a high degree of concurrency using a variant of the multi-version transaction ordering (MVTO) algorithm. Transactions may be prioritised to ensure low-latency updates. The performance and scalability of the concurrency control mechanism compares favourably with a simple locking scheme. The model server provides full support for happened-before and known-before ordering of data even when the data arrive unordered or very late by allowing retroactive insertion of records.

Distribution design is one of the key points in the software architecture: instead of using a reference architecture (such as client-server) and working around its limitations, the most beneficial runtime locations of tasks and data for the given requirements are considered. This is particularly important for client applications.

Two classes of client component behaviour are identified. Interactive behaviour is characterised by bursts of short read/write accesses to a varying set of model objects, possibly in reaction to model changes. Interactive components require little CPU and memory, but call for low-latency model access and immediate notification of model changes. They access the model as agents that are executed as threads within the model service, following the principle to run code where its data are. Agents can be installed and activated at run-time without service disruption.

Batch behaviour is characterised by intensive, long-running work on a fixed set of model objects, as is typical for building simulation and other analysis tasks. Write operations are rare. Such tasks call for great amounts of CPU cycles and are typically less affected by concurrent model changes. Batch components are therefore distributed to other systems and work on snapshot copies of the required model subset. Combined with a space-based communication layer, this distribution is transparent to a high degree and allows simple load distribution with a minimum of central administration.

Communication with the building systems is performed through an external message-queue infrastructure. The model service uses a flexible processing pipeline served by thread pools to locate and update the target objects corresponding to incoming messages. For concurrency-optimised internal communications (e.g. between agents), the model service provides a simple generalised message queuing system that can transparently connect to the external messaging system if needed. A facility for runtime-pluggable code is provided to achieve a high degree of modifiability and availability.

Contents

List of Figures

Acknowledgements

This dissertation was written as part of a multidisciplinary project on sensor-driven building models. My thanks go to the entire project team at the Building Physics and Building Ecology group of the Institute for Architectural Sciences at Vienna University of Technology: Oğuz İçoğlu, Josef Lechleitner, Bojana Spasojević, and Georg Suter. In particular, I would like to thank my advisor, Prof. Ardeshir Mahdavi, for his valuable support throughout my employment and thesis work – and of course, for his insistence on submitting papers for publication and attending conferences.

I am very, very grateful for my family's unwavering and unconditional support throughout my entire academic curriculum.

—*K.B.*

The work documented in this dissertation was supported in part by a grant from FWF (Fonds zur Förderung der wissenschaftlichen Forschung), project number P15998-N07.

1 Introduction

This dissertation describes the design and prototypical implementation of a dynamic, integrated building model service. Its purpose is to build and maintain a live computational representation of a building and to provide open access to client applications interested in monitoring and controlling the building and its subsystems. The model is continuously updated through various sensors, keeping track of changes as they occur.

Much research into intelligent buildings has focused on sensors and actuators and the communications between them and control or monitoring applications. However, one significant shortcoming of these efforts is that they are often hard-wired to specific applications, and in many cases, there is no communication between domain-specific information systems within the very same building. This is a situation where an additional abstraction layer makes sense: a layer that unifies the various data sources and sinks of a building and offers a transparent, open interface to any number of applications.

The primary goal of this work thus is not to increase the amount of *data* sensed and communicated within a building. The goal is to make full use of the available data, avoiding duplication of efforts, improving efficiency, and gaining as much *information* as possible. To achieve this goal, an integrated, contextual information model is necessary.

Context is required to gain information from data: a temperature sensor reading is meaningless without knowledge of its location, its surroundings, and the factors influencing its value – e.g., the room where it is located, the height at which it is mounted, the HVAC and shading systems that affect its value. Today, modern office buildings are often equipped with considerable networks of sensors and actuators. However, there is generally a lack of integration and open access to make full use of these available data.

1.1 Motivation

This work is part of a research project on *sentient buildings*, which are essentially buildings that have an internal model of themselves as outlined above (Mahdavi 2004). The project focuses on the application of such internal models toward supporting indoor-environmental control systems of buildings (e.g. heating, cooling, ventilation, and illumination systems). Specifically, it aims to explore the potential of dynamic building models to enable simulation-based building control strategies (Mahdavi 1997, 2001, Clarke et al. 2001).

1.1.1 Simulation-Based Control

Simulation-based control can be seen as an extension of a technique called Model Predictive Control (MPC), which has been used for industrial production processes since the early 1970s. Predictive control relies on a parametric model of the controlled system to estimate the expected outcome of a control action (Rawlings 2000). Deriving such a model is not trivial and may require frequent test runs of the system to assess its reaction to various inputs; a human expert is needed to develop and optimise the model. In complex situations with many variables and nonlinear behaviours, creating such a model may be infeasible.

While MPC demands this *custom* approach, simulation-based control as discussed here allows a more *generic* approach. This means that the controlled system's relevant components are modelled individually and used to populate a simulation space in which their reactions to influences received from the outside and from other components are calculated by a simulation tool. Instead of trying to develop a global equation describing a specific system's overall behaviour, systems are reconstructed part-by-part in a virtual model (a potentially automatable task) to run repeated experiments. The big advantage of simulation-based control versus MPC is thus its potential to eliminate the need for a human expert hand-crafting or adapting a parametric system model each time the system's structure changes.

In the building domain, the basic prerequisites for simulation-based control are available:

- Models of individual building components (product models) are relatively easy to create or derive from design data, or even readily available from manufacturers.
- The spatial configuration of the components can be derived from design data or through location sensing technologies (Icoglu et al. 2004, Suter et al. 2005).
- Simulation tools for various aspects of building performance have been available commercially and as research projects for some time (Ward 1994, Rindel 2000,

Citherlet and Hand 2002).

Challenges

One issue of simulation-based control approaches is the high processing volume, requiring considerable computational power to keep simulation times low enough for control applications. Advances in computer hardware and simulation algorithms have, however, brought simulation times to a level that is useable for building energy management systems applications.

Moreover, simulation requires a fairly detailed model of the building, its systems, its context, and its occupancy. However, creating simulation models is still manual labour to some extent. The transition from initial CAD (computer-aided design) building documents to simulation models is hardly seamless and often requires additional domain-specific information and extensive post-processing.

Given the dynamic nature of building-related processes, such a model must be continuously updated to be of any use in the context of building systems control. In a statically designed control system, a model of the controlled system is established by an engineer and remains fixed until further human intervention – even when the real world context changes. If it is not accurate enough, the control system will fail: e.g. if a new wall is erected within the building, separating a sensor from the region of interest, the context model is out of touch with reality and any control system based on it will not function correctly.

Ideally, simulation-based control utilises a model of the building's status that is updated without human intervention. This evidently requires an extensive sensor infrastructure in the building generating a huge amount of raw data – and consequently, software that processes these data, collating and organising them contextually for access by other software.

1.1.2 Other Uses

While simulation-based control is the primary driver for this work, it is not the only application that could be based on it. On the contrary, one of the underlying assertions of this dissertation is that a dynamic building model can be useful for many other purposes besides simulation-based building systems control, offering a level of abstraction and a common interface that has not been available so far. For example, another major application that would benefit from an up-to-date building model is facility management. Today, a number of proprietary solutions supporting tasks such as asset inventory and condition assessment exist, but there is little information sharing between them (Hassanain et al. 2003).

1.2 Research Statement and Thesis Overview

This dissertation aims to contribute to the state of knowledge in the following ways:

- First, by stating the requirements of a building model service with a strong focus on the operational phase of a building's life-cycle. This is documented in chapters 2 (Background) and 3 (Specification).
- Second, by proposing an architecture to fulfil the functional requirements and quality attributes outlined previously. This is the main purpose of chapter 4 (System Architecture and Design).
- Third, by proposing a suitable software design for the operational core of a building model service, given the stated requirements. This is the focus of chapter 5 (The Model as a Temporal Database).

Finally, chapter 6 describes the setup of an application prototype system that was built for the purpose of experimentation and evaluation of the architecture and design.

2 Background

The goal of this chapter is to demonstrate a general understanding of the state of the art, examine related research work, and thus place this work's contribution in context.

2.1 Building Communication Systems

The purpose of building communication systems is the remote control and monitoring of appliances such as light fixtures, valves, and sensors. Typically, each appliance can be connected anywhere on the system and accessed from any other part of the system based through an address code.

As of today, various mostly incompatible building communication systems exist, differing in hardware technologies, application scope, and the layers of the OSI reference model they implement (Kastner et al. 2005). Recently, the growing coverage of offices with networks based on the internet protocol (IP) has driven the utilisation of such networks for building control purposes. IP networking is then used as a tunneling medium or as a replacement of some layers in existing building control protocols. Whether this partial convergence will ultimately lead to a full integration of building control and office communications is still subject of discussion (Finch 2001).

Building communication systems are a crucial element of building automation and a prerequisite for creating self-updating building models. Although there is little overlap between the two areas, there is a clear need to understand the design approaches used in building communication in order to design effective interfaces between them.

2.2 Building Automation and Control Systems

Building communication systems provide the basic layer for remote interaction with various devices. This is a requirement, but not sufficient for building automation, which also implies some form of automatic control. Designing control systems entails a number of design choices: how to model the controlled zone, which sensors and actuators to use, which control algorithms or heuristics to use, how and where to deploy the control system, to name a few.

Most of these questions have been the subject of intense research for decades. During this time, advances in electronics and computing have driven progress and widened the possibilities: new modeling approaches have become feasible, just as integration of previously disparate control systems (Pargfrieder and Jörgl 2002), simulation-based control (Mahdavi 1997, 2001, Clarke et al. 2001), and adaptive or self-learning control (Guillemin and Morel 2001).

In the research projects just cited, control applications are generally hard-wired to a communication system and expected to have exclusive access to the controlled devices. This approach is acceptable for research prototypes, but not for most real-world applications. As building communication systems – the lowest layer – and applications – the highest layer – have become more complex and powerful, the need for an equally powerful middle layer has clearly emerged. This middle layer is not merely a mediator between communication systems and applications, but also structures the way applications interact with each other, and may prescribe how applications are deployed during run-time. It is not merely an addition, but introduces a new quality of system architecture.

Research on such system architectures has picked up only recently. Many of the systems are agent-based and decentralised, such as the one proposed by Sharples et al. (1999) and refined by Cayci et al. (2000). Davidsson and Boman (2005) suggest a similar approach. Their common theme is that control is distributed on local devices (typically one per space), where *agents* are processes acting on behalf of users and other parties to decide on the control actions to be taken.

The OWL framework by Brügge et al. (1999) is a distributed, event-based system that adds an object-oriented layer of abstraction on top of existing communication buses, with a focus on facility management applications.

The primary advantage of decentralised approaches is improved scalability and reliability: new devices can be added as necessary without increasing the load on existing devices, the failure of a single device does not necessarily imply a total system failure. The main disadvantage is that these approaches are not well-suited for simulation-based control strategies that span multiple spaces (see chapter 4 for further discussion of this issue).

2.3 Pervasive and Ubiquitous Computing

In Pervasive and Ubiquitous Computing,[1] environments are envisioned as providing an ubiquitous computing infrastructure that adapts to the needs of users and assists in their everyday tasks – ideally in an intuitive, non-obtrusive manner to the point of "disappearance" (Weiser 1991). The PC as a complex universal tool loses importance in favour of networks of specialised devices that may collaborate to support some user-defined goal. A typical research focus is the intelligent support of collaborative teamwork.

While it certainly comes with a huge amount of technology, pervasive computing is focused on assisting humans and human-to-human communications. The building is principally seen as a hardware skeleton that structures the environment and carries the necessary infrastructure, but it is usually not of interest by itself.

Despite this different focus and the different design approaches that follow from it, pervasive computing has to solve problems very similar to those of sentient buildings: both must handle potentially large, heterogenous and changing networks of devices that interact with the physical environment.

2.4 From Product Models to Building Information Models

From the viewpoint of information technology, the current situation in building industry is characterised by weakly connected "information islands" for the various parties involved in the construction of a building: architects, structural engineers, HVAC planners, and so forth. While most of these participants rely heavily on IT support for their efforts, the overall picture is one of barely connected fragments.

Information exchange between these fragments is still largely a matter of sending files and manually translating them to the various specialised applications involved, such as CAD software, building performance simulation applications, structural engineering software, and so forth. Subsequently, design iterations cause a cascade of back-and-forth file transfers and conversions. The process involves lots of overhead and repetitive work and is hardly conducive to concurrent and collaborative engineering. A central, integrated, managed repository of building-related data is rarely used.

Overcoming this situation requires two major development efforts. First, a common vocabulary for describing buildings and building-related things must be defined. This

[1]The terms are often used interchangeably and have become almost synonymous. From this point on, the term "pervasive computing" is used.

is a nontrivial task, given the various participants in the building lifecycle, their vastly different areas of concern, and their fundamentally different ways of describing the same things. Second, a well-defined collaboration framework based must be created to connect all participants interested in building data in a structured manner. This, again, has to be done in consideration of the different requirements of the participants and lifecycle phases.

The ultimate goal then is to create so-called Building Information Models (BIM) that follow a building's entire life-cycle from conception to destruction. The US National Institute of Building Sciences defines a BIM as "a digital representation of physical and functional characteristics of a facility . . . a shared knowledge resource for information about a facility forming a reliable basis for decisions during its life-cycle from inception onward." (Nat 2006).

2.4.1 Building Product Models

A wide range work has been done in the field of building product modelling with the aim of creating semantic models for buildings, mainly to improve collaboration in the design and construction phase of buildings (Eastman 1999). Such building product models are intended to define a common *vocabulary* for collaboration.

The Industry Foundation Classes

Currently, most work in building product modeling is done under the umbrella of the International Alliance for Interoperability (IAI). Originally an industry consortium, the IAI now also comprises members from academia. Their main area of work is the Industry Foundation Classes (IFCs), a data model definition with some object-oriented features aiming to capture all aspects of building projects throughout their lifecycle (International Alliance for Interoperability 2006, 2000).

As of their current version 2x edition 3, the IFCs are capable of modelling a range of diverse concepts including:

- Shapes (such as beams, walls, pipes)
- Building elements (doors, windows, roof)
- Relations between elements (holes, zones)
- Spaces (space, storey, part, wing)

- Grids
- Equipment and furniture (fans, pumps, tables, chairs)
- Actors (people, organisations)
- Costs
- Work plans and schedules
- Orders (work orders, change orders)
- Assets in the sense of facility management (inventories, maintenance histories)

The IFC definition is based on the ISO STEP (ISO 10303) data interchange development model, which uses EXPRESS as its normative data modeling language. Recent versions of the IFC documentation add an XML representation called ifcXML to enable the use of XML-based tools and interfaces.

A bottom-up view of the core IFC class structure is given in figure 2.1. The abstract root supertype IfcRoot carries basic properties for identification, ownership, history information. Three major branches derive from it:

IfcObject This is the supertype for the primary entities modelled by the IFCs, defined as products (physical objects), processes, controls (concepts that control or constrain objects), resources (concepts that are used by objects within a process), actors (human agents), projects, and groups (collections of objects).

IfcPropertyDefinition Properties are used to further define classes, object groups, and individual objects. The IFC properties definition is very powerful and flexible, which alleviates the need to derive a large number of early-bound, specialised subtypes to model real-world products – a typical problem of naive approaches to object modelling that tends to result in inflexible models.

IfcRelation In the IFCs, relations between objects are not direct links, but are also modelled as objects. This indirection allows to separate relationship-specific properties and semantics from the related objects and also improves modelling flexibility. Different classes of objects exist to cover various types of relationships such as containment and aggregation.

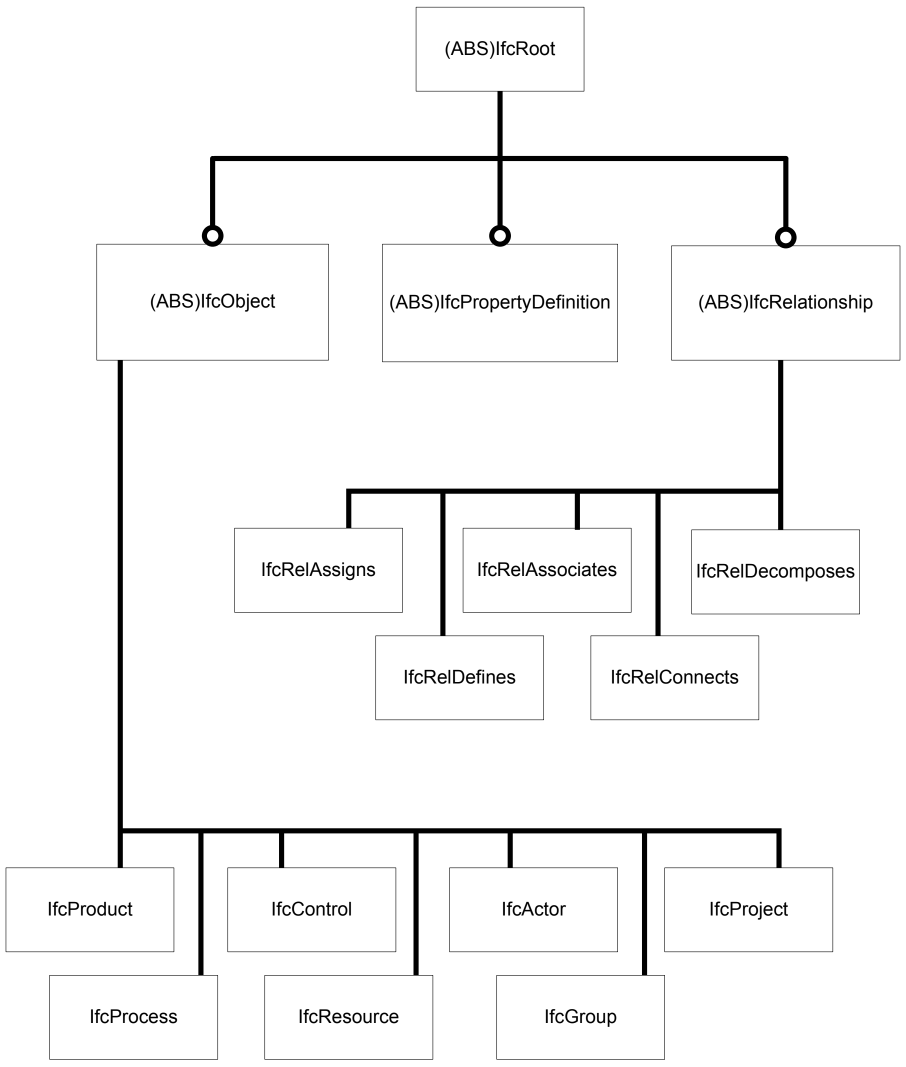


Figure 2.1: *Basic IFC entities in EXPRESS-G notation (simplified)*

The flexibility of the IFC design comes at a cost: it markedly increases the object count and model complexity. Additionally, the many freedoms afforded by the versatile property and relation model would make automatic model validation (a crucial tool for interoperability work) harder to implement. However, the properties and property sets have been designed to prevent the need for deep inheritance: instead of explicitly modelling every kind of possible building product, the IFCs stop at fairly general concepts like "window" and leave the detailed specification of a concrete window type to be expressed as property sets.

From a top-down perspective, the IFCs are structured in four layers (Figure 2.2) related in strictly unidirectional manner: higher levels can use or derive from the same or lower layers only.[2]

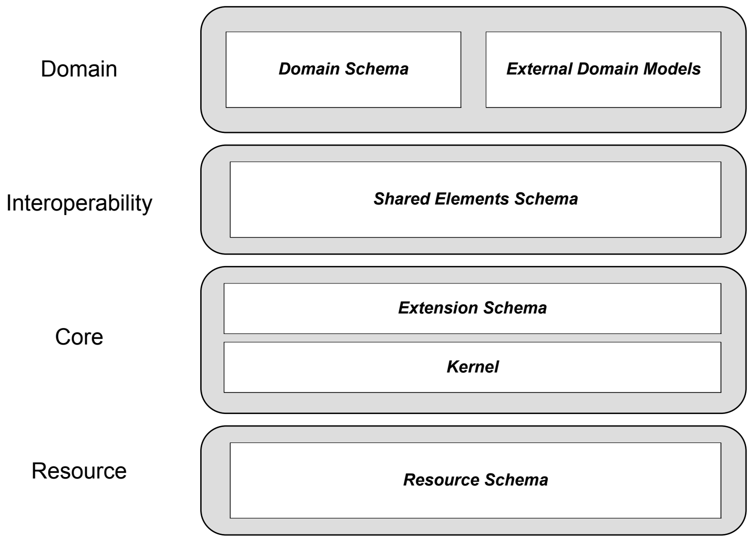


Figure 2.2: *A layered view of the IFC architecture*

Beginning at the bottom, the layers are:

Resource Layer This layer contains general-purpose and utility classes such as IfcGeometryResource or IfcDateAndTimeResource.

[2]The layers are not opaque as is often implied by layer diagrams that represent software interfaces. There is no isolation that prevents or discourages access from e.g. the domain layer to the core or resource layer: classes in the domain schema may directly use classes from all layers below it.

Core Layer The core layer defines the central, basic classes that form the foundation of the IFCs. Root classes are defined in the Kernel sub-layer, where they are still abstract to the point of having no discernible conceptual connection to AEC/FM applications (as seen in figure 2.1). This building-related link is first introduced in the Core Extensions sub-layer with concepts such as sites, buildings, and spaces.

Interoperability Layer The interoperability layer introduces concepts that are shared by two or more domain models, e.g. walls, doors, beams.

Domain Layer The domain layer contains domain-specific concepts. The latest version of the IFC specification as of this writing, version 2X3, defines nine domain model schemas. These include architecture, facilities management, building controls, plumbing and fire protection, HVAC, and others.

Mature software support for the IFCs in the industry is currently far from universal, but apparently growing. No other free industry standard of comparable scope, maturity, and industry support exists at this time.

aecXML

Sometimes considered a competitor to the IFCs, aecXML is more of a complementary effort. As opposed to the IFC goal of creating all-encompassing building models, the intention of aecXML is to facilitate automated business transactions in the Architecture, Engineering, and Construction (AEC) industry based on XML.

Work on aecXML has not progressed significantly in the past few years. It appears that the specification scope and relation to IFC are still unclear (Zhu and Weng 2001).

2.4.2 Building Model Servers

With the number of IFC-compliant applications growing, data exchange issues as outlined above have become more acute. Work on model IFC servers has started in the past few years in both the research community and industry (Kiviniemi et al. 2005). The currently published work is generally characterised by a strong focus on collaboration in the design and construction phase of buildings.

IMSVR

One of the first IFC model server projects was IMSVR. The project was finalised in 2002 after producing a research prototype (Adachi 2002a,b).

IMSVR stores IFC model data in a relational database, offering access to clients through web services based on the Simple Object Access Protocol (SOAP). The architecture comprises three major components (Figure 2.3):

Database A relational database system (Microsoft SQL Server) is used for persistent storage. The database schema is generated from the IFC EXPRESS definition using a two-phase conversion process that first generates an intermediate XML format. This format, in turn, is transformed to SQL DDL statements in order to create the database schema, stored procedures, and triggers.

Data Access Layer Component (DALC) The data access layer manages generic client access to model data, ranging from simple single-object access to complex queries specified in a specialised query language called PMQL (Partial Model Query Language). Communication with the database is based on SQL and XML returned by stored procedures.

Web Service Layer Component (WSLC) The web service layer provides SOAP access to the data acess layer with operations such as AddObject, GetObject, DeleteObject, etc. All operations are stateless.

No data have been published on the performance characteristics and scalability of the IMSVR system. The architecture appears unlikely to be designed for high performance: it relies heavily on XML processing and incurs synchronous database accesses for each model access.

Eurostep/Webstep

The commercial Eurostep Model Server for IFC (EMS) is largely similar in architecture to IMSVR, but offers a richer function set by providing tools for visualisation/model browsing, user management, version management, etc. (Hemiö and Noack 2002). EMS is implemented in Java and uses a relational database management system (such as MySQL, SQL Server or ORACLE) for persistent storage. Clients access the server via HTTP transport using the standardised ISO 10303 XML schema, which may also be

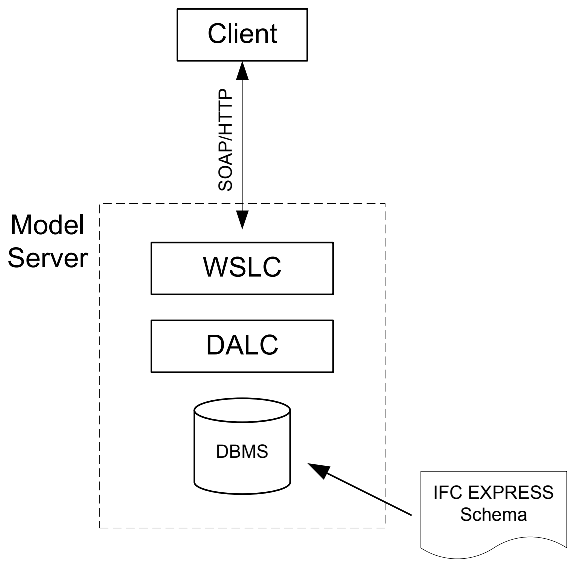


Figure 2.3: *Architectural overview of IMSVR. Adapted from Adachi (2002a).*

wrapped in SOAP envelopes. The communication pattern is session-based (i.e. stateful). Objects are stored in multiple versions, where only the latest version can be changed.

Product advertising material states that the model server can import data at a rate of about 1200 objects per second when importing a new model, and about 50 objects per second when appending new data to an existing model (Eurostep 2003). The performance of queries is claimed to be highly dependent on usage patterns, as caching is used extensively in the model server.

Just as IMSVR, the project has apparently not progressed significantly since 2002.

iCSS

The iCSS[3] system developed jointly by the Dresden University of Technology and industry partners aimed to create a cooperative work environment for building construction based on an IFC model server (Scherer et al. 2003). More than just offering a building data repository, it was also designed to support project management and work-flows. One notable innovation of iCSS is a conflict management tool to resolve inconsistencies that are an inevitable side-effect of parallel teamwork on the same project. In the

[3]Integriertes Client-Server-System für das virtuelle Bauteam

system core, an information logistics component (“Informationslogistik-Komponente”) unifies services – such as workflow, product model data, and contract data servers – and clients via a standardised Application Programming Interface (iCSS-Schnittstelle), accessible through TCP/IP and Java Remote Method Invocation. Little information on the software architecture has been published.

EDMserver

As of 2006, the only fully-functional, mature, and actively developed model server available appears to be EDMserver (EPM Technology 2005), a proprietary commercial product. EDMserver aims to support the entire product lifecycle including aspects such as facility management. An architectural overview is given in Figure 2.4, based on (Dahl 2005).

EDMserver can be used either as a single-user local application for model access, or in client-server mode. The server core, EDMdataServer, holds the actual product model and handles persistent storage. Client applications (which may include EDM application servers or external clients) can access the server core in two ways:

- Fat Client: This is a session-based (stateful) communication model that supports transactions. Fat clients typically check out portions of the model, work on them, and check them back in once they are finished. Checked-out data are usually locked for other applications.

- Thin Client: This is a stateless communication model that does not support transactions.

Most calls to the EDMserver are handled by EDM application servers, only some calls are served directly by the EDMdataServer core. The actual routing of these calls is, however, transparent for client applications. EDM application servers support synchronous and asynchronous modes of operation.

The transaction model allows multiple concurrent read-only transactions to a model or set of models, but only one open write transaction at a time. Any additional write transaction requests are enqueued and executed sequentially. Write transactions may be nested.

Versioning is implemented so that only changed objects are stored with each version to maximise space utilisation. Only the most recent version of the model can be changed, i. e. branching is not possible.

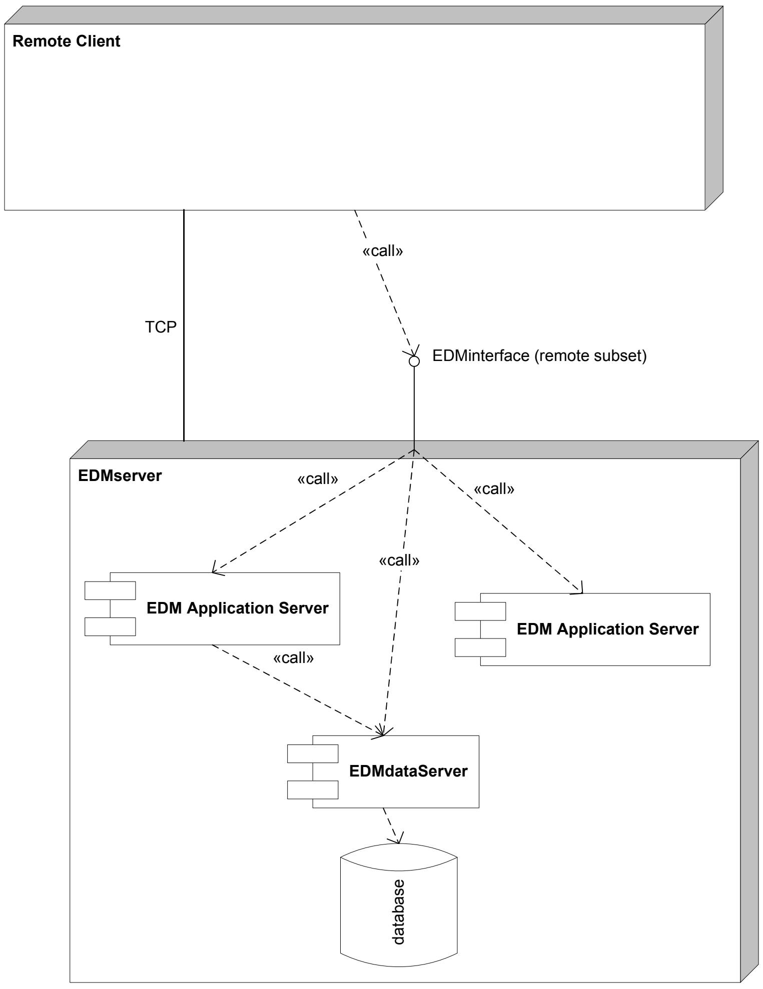


Figure 2.4: *A deployment view of EDMserver*

A notable EDMserver-based application for building information models has been outlined by O'Sullivan et al. (2004). Building performance data are collected from a building management system and incorporated into a central building model. The collected data may be used to evaluate control and design decisions. Moreover, the model is also envisioned to be used as a basis for predictive control routines, although no specifics on this kind of application have been published yet.

An Interface Standardisation Effort: SABLE

In anticipation of a growing number of IFC model servers, the SABLE (Simple Access to the Building Lifecycle Exchange) project aims to specify a framework and standardised API for communication between model servers and clients (BLIS Project 2005). SABLE is intended to act as a mediator and standardised protocol glue that simplifies communication between multiple clients and multiple model servers during the building lifecycle. No final and complete documentation on the project's outcome has been published to date. Draft documents suggest that the design of SABLE is inspired by IMSVR and Eurostep work, with an emphasis on web services.

SEMPER and S2

The SEMPER and S2 projects created a distributed object-oriented design environment for integrated building performance modelling, allowing remote collaboration over the Internet (Mahdavi et al. 1996, 1999). Using OMG CORBA technology to connect remote systems, users could simultaneously use various domain simulation modules to assess building performance for interactive design support, regardless of their physical location (Figure 2.5). As in the other projects discussed in this section, the focus was clearly set on design support only.

At the core of the S2 architecture, the S2 Kernel module provides model services such as managing persistent storage and concurrency control. It employs a simple "check out and lock" approach to manage multi-user access: as long as a user has checked out a project, other users may only access it for reading (Lam et al. 2001).

Unlike the IFC-based model server projects mentioned above, SEMPER/S2 uses the Shared Object Model (SOM) for building modeling. As opposed to the IFCs, SOM is based on a relatively simple and straightforward object hierarchy (see Figure 2.6) and geared toward building simulation. When used for concrete simulation application, the SOM is mapped to a domain object model (DOM) that may exclude parts of the SOM

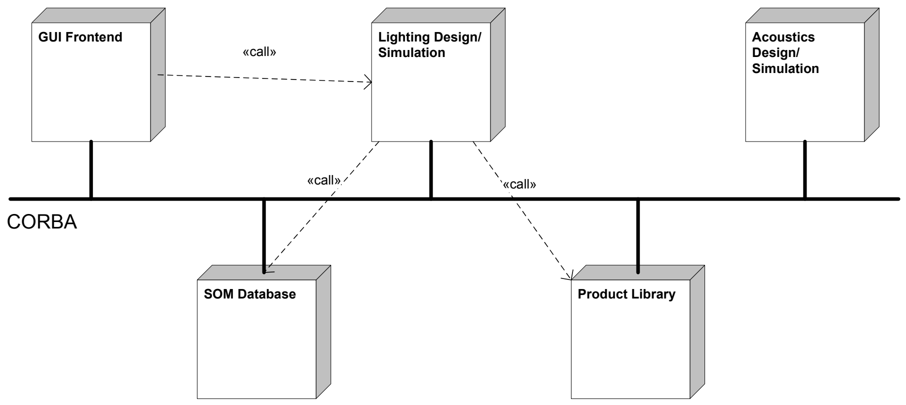


Figure 2.5: *An example deployment view of S2.*

data, model it in different ways (e.g. by mapping objects to attributes) and augment it with domain-specific data (Mahdavi et al. 2002).

Other Projects

The discussed list of model servers and related projects is not exhaustive, but represents the main architectural approaches that have been published so far. For instance, a three-tiered web-based architecture was also used in the WISPER project (Faraja et al. 2000), and a CORBA-based architecture similar to S2 was proposed by Brown et al. (1996).

Conclusion

There is relatively little published information on the software design of the building model servers described above, and even less on their performance characteristics. The servers represent different stages of development, from the research prototype IMSVR, which was finalised in 2002, to EDMserver, which is under continued development and appears as a comparatively mature and robust product with a large feature set. Product and design documentation suggests that the development of all three model servers is based on the requirements of supporting building design and construction. The Industry Foundation Classes (IFC) are generally used as basis for the data model and exchange format.

The architectural patterns of IMSVR and EMS are straightforward and proven: they

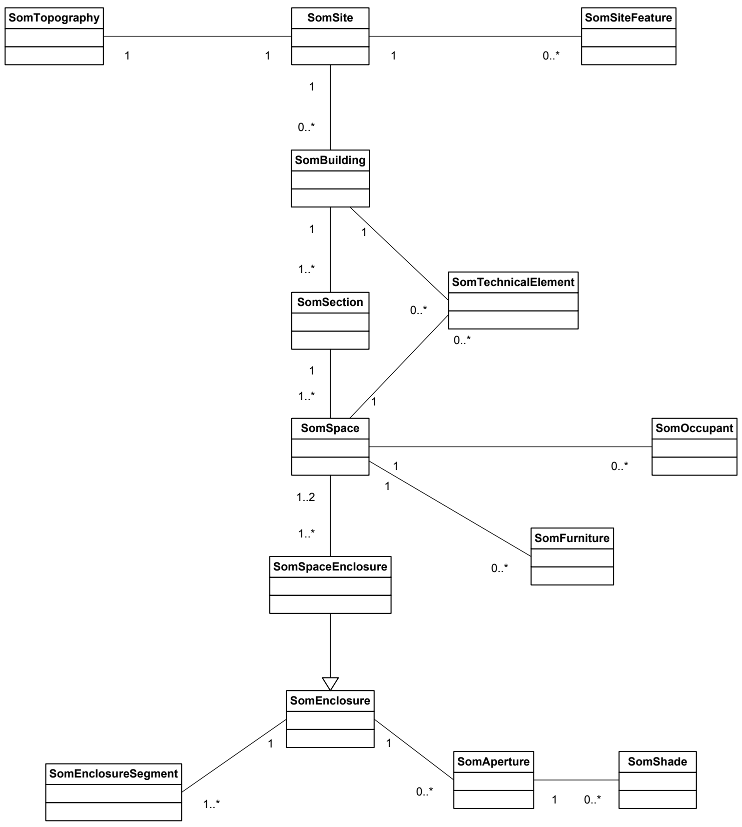


Figure 2.6: *Shared Object Model (SOM), simplified class diagram*

have been used successfully in information systems for years. However, they are not as well suited for the requirements of building monitoring and control based on large object networks. Specifically, the three-tier style is inflexible in that it is tailored for a client-server, request-response communication scheme. The available documentation suggests that massive concurrency was not a design consideration. EDMserver is considerably more complex, but appears more scalable and also offers greater flexibility in terms of client location and interaction patterns and thus shows higher potential for supporting building operations. However, there are no published data as to how well it performs in situations with sustained high update rates from multiple data sources.

Most current work on IFC model servers is proposing Web services (usually SOAP-based) as the interface to use for client communication. This technology holds a certain appeal for integration: as it is based on standard Internet technology and the XML format, interoperability is in principle easy to achieve between completely different programming languages and operating environments. The growing popularity of Web services may also owe to practical considerations such as typical corporate network firewall setups which often permit HTTP traffic readily, but deny most other protocols.

However, Web services entail substantial processing and communications overhead due to the necessary data format conversions (XML generation and parsing). Complex queries can exceed the limits of a simple parameterised interface and may have to be expressed as either a convoluted series of requests and responses, or encoded in a specialised parsing language which adds development overhead and another layer of text parsing.

2.5 Chapter Summary

In this chapter, related work in various fields was surveyed, with a focus on building model servers. In summary:

- Work on building models and model servers has so far been focused strongly on design support, although the vision of full life-cycle coverage is a strong driver for development.
- The technology to measure and control a wide range of building parameters and systems, as well as technology for in-building communications for these data, is available and has been used for years.
- Beyond conceptual sketches, there is little published work on integrated, open-access building model servers that support the operational phase of a building's life-cycle. Existing model server designs are not suitable for these requirements.

3 Specification

In this chapter, the system to build is defined in terms of functionality (functional requirements) and quality attributes (nonfunctional requirements). The specification is intentionally restricted to key features of the system that are deemed essential for building model servers supporting the operational phase of the building life-cycle. Some requirements are left out of the scope of this dissertation, but are relevant nonetheless and therefore included for completeness (3.4).

3.1 System Environment

The purpose of this section is to give an overview of the environment in which the model service is expected to operate, and to define the boundaries of the model service to its environment – in other words, to define what the model service is *not*.

To be of any use, a model service must be supplied with data about the physical world it is supposed to represent. The key issue for this work is that these data are not only supplied once at initialisation or even compilation time, possibly manually by a designer, but *continuously*, throughout runtime. While manual data input must be possible, once the system is operational, the bulk of data about the modeled domain is expected to come from sensors. As the model service is also envisioned to be an abstract interface for building control, there is a need to control actuators that influence the physical world.

The model service thus depends on sensors, actuators, and mechanisms to communicate with these devices, as discussed in 2.1. These are beyond the scope of this work: they are simply expected to exist in the environment. However, interfaces to building communication systems must be considered in the design of the model service. As there is a wide range of different building communications technologies, the model service must be able to deal with various interfaces. This entails understanding the data supplied by sensors, which may include complex geometry information, simple one-dimensional sensor readings (e.g. temperature), complex physical data such as sky luminance distribution patterns (Mahdavi et al. 2005), or highly domain-specific data for certain building subsystems (HVAC, lifts, security systems, to name a few).

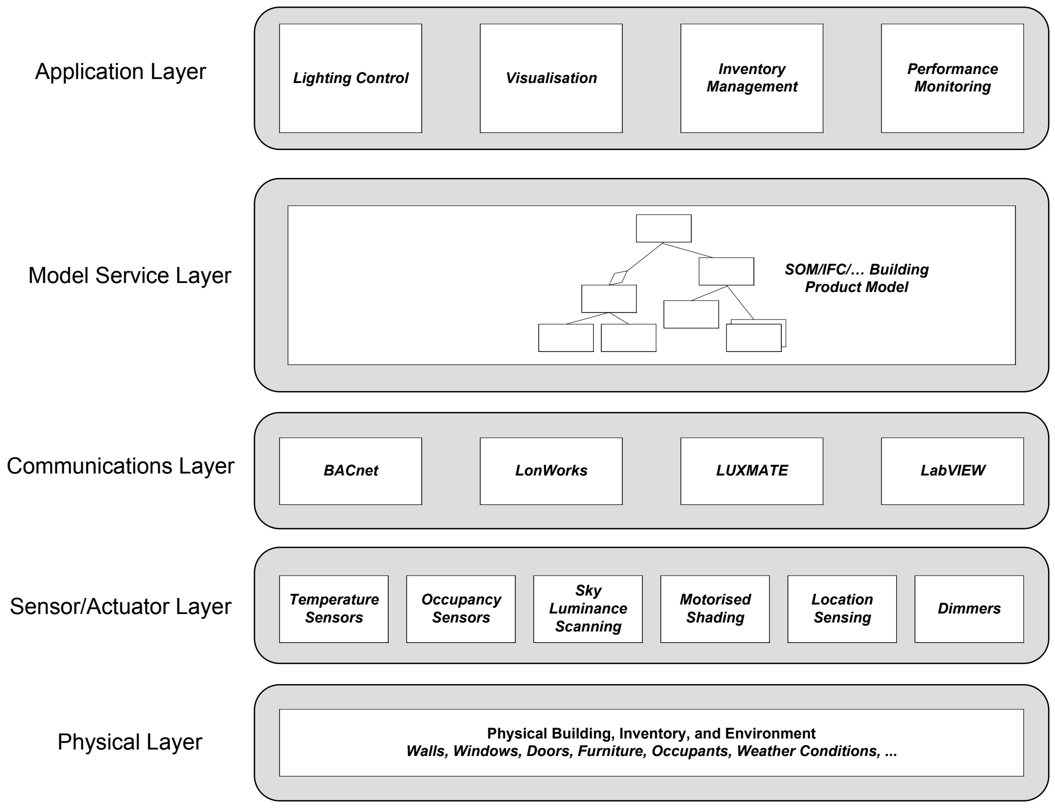


Figure 3.1: *System Layers View*

A layered overview of the system and its environment is shown in figure 3.1. At the lowest level, the physical layer comprises the building and its environment as such. Observing and controlling the state of these entities is handled by sensors and actuators, shown on the next system level. To gather and distribute data between these devices and computer programs using them, a communications layer is needed. This may include combinations of various specialised (LonWorks, BACnet, DALI) and general-purpose (Ethernet, TCP/IP, MQ) communication technologies.

The key element of the system is in the next level, the model service layer. Instead of letting applications communicate directly with the communications layer, it offers an additional level of abstraction that isolates applications from the details of communications and sensor hardware. The model service represents the current state of the building and its environment in the form of a live building product model. Applications are thus freed from dealing with specific communication, sensor and actuator systems, but instead communicate with objects whose properties, methods and relations with other objects provide a high-level interface to the physical world.

3.2 Functional Requirements

In this section, the functional requirements of the model service are outlined. They are grouped in three areas: the core functionality that maintains the model, the interface to the building systems, and the interface to client applications. This grouping does not imply a specific implementation structure, it only serves as a conceptual categorisation.

The listed requirements are to be considered a minimum set of functions; mechanisms for future extension must be provided (see Modifiability below). Each requirement is labeled with an identifier on the margin.

3.2.1 Core Functionality

FR1 Model

At the core of the system, a building model (in short: model) shall exist in the form of objects reflecting information on the physical building. The system must maintain this model, ensure proper initialisation and shutdown, ensure the model's integrity constraints, and handle persistent storage and retrieval.

This object model is a directed acyclic graph of objects, such as a tree structure that assigns a parent and any number of children to each object (except the root).

FR2 Online/Offline Model

To ensure that only a single live building model exists, model objects shall exist in two states: *online* and *offline*. All publically-accessible objects that are part of the core model are in online status, meaning that changes to them are observable by the model and client applications, and may result in automated persistent storage (see below). Method calls to online objects may result in data communication with building systems. Online objects may be changed by the model service based on incoming data, such as sensor readings.

When an object is copied, e.g. by a client application requesting an model object copy, the copy must be in offline status. Offline objects are completely de-coupled from the model: changes to these objects have no effect on online objects whatsoever, and are not observable by the model or other applications. Method calls on these objects (e.g.

attempts to send commands to a light fixture represented by the object) will not result in data communications to the physical building. An error must be signalled to the calling application if such operations are attempted. The model service will not change these objects even when sensor data are received. Offline objects cannot be switched online again, all further copies of offline objects are again offline.

FR3 Model History

Each model object carries a date of last modification, and all states are recorded and timestamped in persistent storage. The states of objects are recorded as versions: whenever the state of an object's data changes, its version is incremented. At all times, it must be possible to track the full change history of an object. This extends to the entire model: the state of the model at a given time in the past must be easily recoverable ("time travel").

Use Cases

Major use cases are described in tables 3.1 – 3.7.

Name	Insert New Object
Actors	Client
Preconditions	The client has a reference to an online parent object for the object to be inserted
Flow of Events	1. The client requests the creation of a new object, stating its type, and the reference to an existing online parent object. 2. The model service creates an object of the specified type and inserts it in the model as a child of the given parent object. 3. The client is returned the reference to the newly created object.
Postconditions	A new object has been created and inserted as a child of the specified parent object.

Table 3.1: Insert New Object (UC1)

Queries

The model service shall provide a way to retrieve objects based on certain criteria, which may be a combination of the following:

Name	Retrieve Object Copy
Actors	Client
Preconditions	The client has a reference to an online object and a version number.
Flow of Events	1. The client requests the creation of an offline copy of the specified object at the given version number. 2. The model service creates an object copy of the object version with the highest available version that is less than or equal to the given version number. A reference is returned to the client.
Postconditions	A new object has been created as an offline copy of the specified object. The copy carries no references to online parent or child objects. Its version number is less than or equal the requested version number.

Table 3.2: Retrieve Object Copy (UC2)

Name	Retrieve Object Tree Copy
Actors	Client
Preconditions	The client has a reference to an online object and a version number.
Flow of Events	1. The client requests the creation of an offline copy of the specified object and its descendants (child objects, child objects of child objects, etc.), given a version number. 2. The model service creates copies of the specified object and its descendants. The descendants' version numbers are the highest available version numbers that are less than or equal to the given version number. The client is returned a reference to the newly created offline copy of the specified object.
Postconditions	A new object has been created as an offline copy of the specified object. It has no references to parent objects, and references to zero or more newly created offline child objects. All objects' version numbers are less than or equal to the given version number.

Table 3.3: Retrieve Object Tree Copy (UC3)

Name	Change Object
Actors	Client
Preconditions	The client has a reference to an online object.
Flow of Events	1. The client requests a change to an object's data. 2. The client calls the object's appropriate function to change data. 3. The model service creates a new object version and applies the changes.
Postconditions	The online object has been changed as requested. Its version number has been incremented.

Table 3.4: Change Object (UC4)

Name	Command Object
Actors	Client, Physical Object
Preconditions	The client has a reference to an online object.
Flow of Events	1. The client requests an action from the physical object represented by the online object. 2. The model service communicates with the physical object to initiate the requested action.
Postconditions	A command has been sent to the physical object.

Table 3.5: Command Object (UC5)

Name	Execute Transaction
Actors	Client
Preconditions	The client has a reference to one or more online objects.
Flow of Events	1. The client announces the beginning of a transaction. The client reads or changes any of the objects. Any changes by the client will not be visible to other clients yet (isolation). 2a. If the client commits the transaction: The model server either applies all changes requested by the client since the beginning of the transaction, or rejects the transaction in case of error or conflict and forgets all changes (atomicity, consistency). Any applied changes are permanent (durability). The client is notified whether the transaction has been committed or not. 2b. If the client aborts the transaction: The model server forgets all changes (atomicity, consistency) requested by the client since the beginning of the transaction.
Postconditions	All changes made during the transaction have been applied and stored (persistence) , or no changes have been applied (if the transaction was rejected or aborted).

Table 3.6: Execute Transaction (UC6)

Name	Publish/Subscribe on Object Change
Actors	Model Service, Client
Preconditions	The client has a reference to an online object.
Flow of Events	1. The client announces its interest in changes to the object. It optionally specifies a filter to select only specific changes. 2. A matching change to the object occurs (is successfully committed). 3. The client is notified of the object change.
Postconditions	The client has been notified of the target object change.

Table 3.7: Publish/Subscribe on Object Change (UC7)

Find by Id Each object carries a unique identification datum (Id) that is not related to any physical attributes of the modeled object. The model service shall provide a function to retrieve an object based on its Id. Example query: "Return object whose Id is 3871232"

Find by Class Each object has a class (as in object-oriented terminology). The model service shall provide a function to retrieve all objects of a given class. Example query: "Return objects of class 'FileCabinet'."

Find by Descent Each object except the root object has one or more parent objects, and zero, one, or more child objects. The model service shall provide a function to retrieve all objects that are descendants of a given object. Example query: "Return objects that are descendants of <object reference>."

Find by Spatial Containment Each object carries a three-dimensional bounding shape of the modeled physical object. The model service shall provide a function to retrieve all objects whose bounding shapes intersect a given shape. Example query: "Return all objects whose bounding shapes intersect a sphere with a radius of 5 metres, with the centre at (x, y, z) coordinates $(30.2, 15.4, 29.3)$".

Find By Time Instant The model service shall provide a function that returns a copy of the object in the state at a given time t, meaning that the timestamp is the highest timestamp less than or equal to t.

Find By Time Range The model service shall provide a function that returns the object in the state at a given time t, meaning that the timestamp is the highest timestamp less than or equal to t.

3.2.2 Interface to the Building Systems

FR4 Model Input

The model state can be changed by data supplied by sensors or other applications. These data must be received and processed so that the corresponding model objects are updated, or objects are created or removed as needed. To support different sensor types and communication systems, it must be possible to dynamically add behaviour for handling them. Handling in this sense relates to two principal tasks: 1) connecting the communication system to the model service so that data flow is established, and 2) interpreting (translating) the received data in order to cause the appropriate model updates. The complexity of such code can vary greatly: from simple adaptation and translation code (e.g. in the case of temperature sensors) to complex, CPU-intensive operations that are driven by multiple sensor readings and may result in wide-ranging reconfiguration of the space layout (Suter et al. 2005).

Conversely, operations on model objects can result in data written to actuators or other applications. The model service shall route data sent through model object methods to their destinations, converting them if necessary. FR5 Model Output

3.2.3 Interface to Applications

As listed in the use cases, client applications shall be able to read the current state of single model objects or the entire model, call model object methods, and change model object states, including the creation and removal of objects.

It must be possible to retrieve the state of a single object, a group of related objects, or the entire model, for any given time in the past in which the model existed.

Isolation of and Cooperation Between Clients

By default, clients must be able to view and use the model as if in isolation, i. e. without particular regard for other clients. This means that inadvertent access to other clients' private memory or any other interference with their operation must be prevented. There is currently no requirement that resource exhaustion (e.g. excessive CPU or memory usage) must be prevented. FR6 Client Isolation

Clients should be able to cooperate with each other voluntarily. This means that they should be able to find each other based on unique identifiers and additional data they may supply themselves (such as classification strings, capability bundles, lists of offered services). A generalised communication mechanism must be provided to allow communication between clients. FR7 Client Cooperation

Transaction Processing

Following standard terminology, transactions – also known as logical units of work in some contexts – are sets of operations that must either succeed or fail as a group. The key properties of transactions are usually given under the acronym "ACID": Atomicity (a number of operations are grouped together as an indivisible unit of work), Consistency (the database must not be left in an inconsistent state after the transaction), Isolation (intermediate stages of a transaction must not be visible to any other tasks than the one initiating the transaction), and Durability (completed transactions are not lost; the database can be restored to a consistent state even after hardware or software failure).

The requirement for providing transactions comes from the fact that many real-world events are represented by multiple operations on multiple related objects in computational models. As an example, a building designer may want to move the position of a window in a wall. The window is represented as an aperture in the wall and additionally an object representing the window. Moving the window's position thus requires at least two operations: changing the aperture's position, and changing the window's position. If either one of these operations failed, the model would be left in a nonsensical state, with a gaping hole in one location and a window stuck in solid matter in some other location. A similar effect can occur if the model server is used like a version control repository, with users infrequently "checking in" batches of updates they have made on their local models over some time period: as long as a batch of updates is not processed completely, another user accessing the model at the same time might see an inconsistent model state.

FR8 Transactions

The model server must therefore ensure that these operations can be grouped together (atomicity) and that a concurrently operating task never gets to observe a model state that shows the effects of only a subset of the transaction's operations (isolation).

A trivial form of transaction processing would simply block access to the entire model as long as one user is accessing it, enforcing sequential processing of each transaction. While this is a very effective way of achieving isolation, it is obviously not scalable and hence not a feasible solution (cf. 3.3). The goal is to enable concurrent access while ensuring transactionality.

3.3 Quality Attributes

This section lists the nonfunctional requirements to be fulfilled by the system. Whenever applicable, concrete criteria are specified in the form of Quality Attribute Scenarios (Bass et al. 2003).

As seen in the survey of existing building model servers in 2.4.2, the main differences between a building model server that is useful for operational support and one that is not lie in these nonfunctional requirements.

3.3.1 Performance, Scalability

The most important measure of performance relates to the delay between a sensor's sending of a readout and the instant it is available in the object model to client applications.

Scalability in this context refers to the ability of the system to perform well (i.e. within the given time limits) when the number of certain entities – data sources, model objects, clients – increases.

Scenarios

Unlike in a real-world business situation, there is no exact hardware setup given in this specification. The goal of the performance scenarios here is to give an indication of the performance that would be required under realistic operating conditions using reasonably-priced hardware available at the time of this writing. All quantitative limits (e.g. performance figures) stated in the scenarios are intended as examples to give some guidance on possible and reasonable ranges. They have not been derived from a formal, quantitative requirements analysis.

The scenarios assume a hardware setup that represents a typical small-enterprise or workgroup server configuration in the year 2006: a dual-core processor machine running at 2 GHz clock frequency, equipped with 1 gigabyte of RAM and an ATA-attached harddisk drive with a capacity of several hundred gigabytes.

The scenarios are intentionally restricted to those situations that are expected to make up the vast majority of system interactions over the system's lifespan. Rare interactions such as massive multi-object inserts caused by import of design data (e.g. during setup) are not considered.

3.3.2 Availability

If a self-updating building model is to be used as a basis for building control tasks (as opposed to noncritical monitoring tasks), availability is a concern.

Availability is a quality relating to the system's continued operation in the face of failure: failure of hardware (e.g. network interruptions, harddisk crashes), failure of software (e.g. operating system crashes).

Although availability is to a large extent the responsibility of base systems, no software system cannot be fully shielded from the effects of failure: even an uninterruptible

Source	Sensors
Stimulus	Stochastic arrival of 10,000 simple updates per minute, concerning any of 1,000 existing model objects (randomly selected).
Environment	Normal operating conditions. No active client processing.
Artefact	Model Core
Response	Model objects updated.
Response Measure	For 99% of all incoming updates, the corresponding object is updated within 500 ms. No update takes longer than 1 s to be processed.

Table 3.8: Performance Quality Attribute Scenario: Simple data updates. A simple update is defined as one that does not affect more than a single object.

Source	Sensors
Stimulus	Stochastic arrival of 1,000 multi-object updates per minute, concerning any 3 of 1,000 existing model objects (randomly selected).
Environment	Normal operating conditions. No active client processing.
Artefact	Model Core
Response	Model objects updated.
Response Measure	For 99% of all incoming updates, the corresponding objects are updated within 500 ms. No update takes longer than 1 s to be processed.

Table 3.9: Performance Quality Attribute Scenario: Multi-object data updates. A multi-object update is defined as one that affects more than one existing object.

power supply can only mask a power failure for a limited time, so the system may have to be shut down on short notice. Network protocols can handle brief connectivity interruptions gracefully, but they cannot mend an accidentally cut wire. Extending the notion of failure, availability may also relate to dealing with scheduled maintenance interruptions such as hardware or software upgrades. Ensuring continued operation under these circumstances may be achieved using a redundant fail-over system, but seamlessly passing control between the systems does require cooperation on the part of the building model software.

For the purpose of this work, only a reasonable minimal degree of availability is specified.

Scenarios

Source	External to system
Stimulus	Impending system shutdown.
Environment	Normal operating conditions.
Artefact	Building model service
Response	State of model service stored and restored without loss, continuing operation where it was stopped. Sensor data are buffered (if separate system is available) and processed after restart.
Response Measure	Disregarding the time needed to restart base systems or transfer stored data, the model service is shut down within 2 minutes and restarted within 10 minutes.

Table 3.10: Availability Quality Attribute Scenario: Availability for shutdown announced on short notice.

Source	External to system
Stimulus	Unexpected system shutdown (e.g. CPU failure).
Environment	Normal operating conditions.
Artefact	Building model service
Response	The building model is restored to a consistent state. Model service is continuing operation. Clients are notified of interruption and asked to resume.
Response Measure	The building model is restored to a consistent state corresponding to the instant just before shutdown or an instant no longer than 1 minute before.

Table 3.11: Availability Quality Attribute Scenario: Availability for unexpected shutdown.

3.3.3 Adaptability and Portability

Adaptability relates to the ability of the system to be adapted to different environments. More specifically, the portability requirement arises from the need to operate on different hardware and software platforms. For a software system that is envisioned to have a very long operational life-cycle phase – which is certainly the case for a system closely tied to a physical building –, this quality is essential to ensure continued functionality and manageable running costs, as hardware and software platforms come and go in relatively short cycles compared to the lifespan of buildings.[1]

Scenarios

Source	External to System
Stimulus	The hardware, operating system, or another basic environment support system is changed.
Environment	System Maintenance.
Artefact	Building Model Service
Response	Change in environment has no adverse effects on operation.
Response Measure	Service can be restored without code change, or code change restricted only to specific, portability-related modules (abstraction layers, device drivers, or similar).

Table 3.12: Portability Quality Attribute Scenario

3.3.4 Modifiability

Modifiability is a quality that relates to the ease (or difficulty) with which the system can be modified to fulfil new requirements. As described in the above section, this is a crucial quality for a system that is expected to operate for a long time. Changes in the original requirements for a software system are inevitable, even more so in the case of a system that strongly relies on interfaces to many external systems.

[1]For example, IBM Corporation currently guarantees a minimum of 3 years software support beginning at a product version's general availability. Microsoft Corp. states a minimum of 10 years for business/developer products, of which the second half is covered under a more expensive "extended support" scheme.

Modifiability has a few different aspects: it can refer to modifications during development, but also to modifications during run-time. A system that provides a high degree of run-time modifiability also has advantages for availability.

Scenarios

Adding or removing sensors that use a known, implemented interface should be as simple as possible, with minimal operator effort. This can also be seen as a measure of robustness: a few failing sensors should not disrupt the entire system's operation (Table 3.13).

Source	Sensor/actuator system
Stimulus	A sensor or actuator is added to the system.
Environment	System Maintenance.
Artefact	Building Model Service
Response	Addition of the sensor or actuator does not disrupt the operation of the system. Proper connection of sensor or actuator to corresponding object is configured manually or detected automatically through context, such as position information.
Response Measure	No downtime. In case of manual reconfiguration, changes take effect immediately.

Table 3.13: Modifiability Quality Attribute Scenario: New Sensor/Actuator.

Adding an interface to support a new technology (such as a specific building automation bus) should be possible with minimal changes to the existing system. No binary code changes or recompilation of existing code should be necessary. Similarly, changing interface code (updates, bug fixes) should be possible without disrupting the entire system's operation (Table 3.14).

3.4 Non-Requirements

This section lists requirements that are not within the scope of the work covered in this dissertation – however, they are not considered irrelevant.

Source	Sensor/actuator system
Stimulus	An new type of sensor/actuator interface is introduced, or an existing interface is updated.
Environment	System Maintenance.
Artefact	Building Model Service
Response	Addition of the interface module and removal of old interface module takes place at run-time, without shutdown of the system. Any buffered data generated while the new interface module is not operational are processed after interface startup without manual intervention.
Response Measure	No system downtime.

Table 3.14: Modifiability Quality Attribute Scenario: New Sensor/Actuator Interface.

3.4.1 Security

The proposed architecture does not have to offer any dedicated security-related features. It is assumed that malicious users are kept off the system by external mechanisms.

Some aspects of security could be handled by the building communication systems, such as maintaining integrity of sensor and actuator data. Other security concerns can, at least in part, be handled by relying on security services of the underlying networking protocols and operating systems. These would be sufficient to restrict access to the model service authorised users only.

In a real-world application, there would be a need to identify users, keep track of their actions, and provide different levels of access to them. For the purpose of this work, all users and applications may be treated identically. The architecture should, however, be designed to make later introduction of security tactics as simple as possible.

3.4.2 Multi-model Support

Despite all attempts to create a single unified building model, there may be a need for keeping multiple separate models in the model server, and for providing merging and mapping functionality for this case (Kiviniemi et al. 2005). This is not a requirement for this work.

3.5 Chapter Summary

In this chapter, the key requirements for a building model server to support the operational life-cycle phase have been discussed. They were categorised as

- Functional Requirements that relate to the model core as well as its main interfaces to applications and the building, and
- Quality Attributes (Non-functional requirements), most importantly performance, modifiability, and availability.

The quality attributes and functional requirements are considered equally important. In fact, it is chiefly the quality attributes that differentiate the requirements of supporting the various life-cycle phases of a building. Most of the described quality attribute scenarios would be either irrelevant or very different for design support, while the functional requirements would be very similar.

4 System Architecture and Design

This chapter outlines a system architecture for the building model server. The description moves from a general overview and discussion of overarching design principles to selected key points of the design, driven by the requirements described in the previous chapter. The model server core is discussed in detail in the following chapter.

4.1 Overview and Context

Figure 4.1 gives an overview of the system's context. One of the major interactions is between the model service and the building control systems that are the model's source of information about the physical building, as well as a destination for commands that the service sends to change the states of physical objects. The system does not necessarily interact with only one such communication system, and the set of communication systems may change during runtime.

On the other side, human users as well as clients application interact with the model service to query its current state or send commands to building systems. Just as the set of communication systems, the set of client applications and users can change anytime.

Notably, this diagram considers system developers important enough to be considered part of the system's environment. As the model service is not merely an application, but a foundation for client applications, its design must consider runtime interaction with developers of such applications.

System managers are important as well, given the availability and maintainability requirements. Monitoring and configuring the system are crucial runtime activities performed by these users.

4.2 Some Architectural Considerations

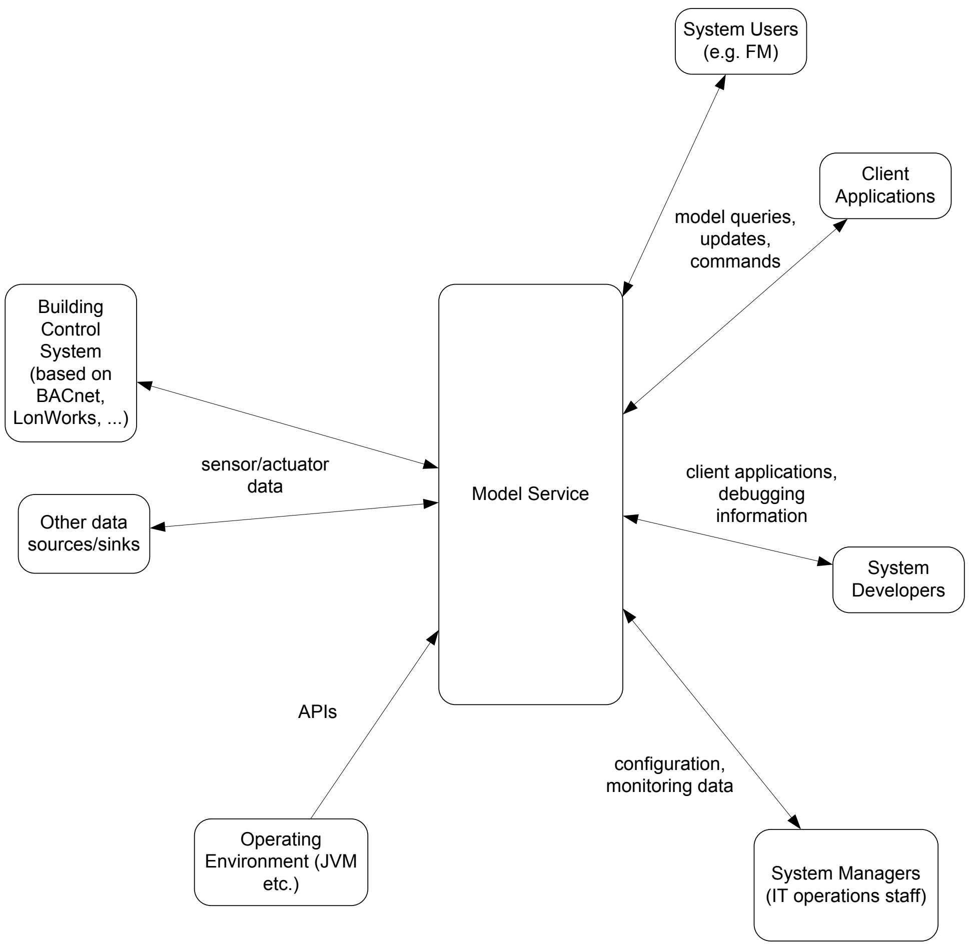

System Users (e.g. FM)
Client Applications
model queries, updates, commands
Building Control System (based on BACnet, LonWorks, ...)
sensor/actuator data
Other data sources/sinks
Model Service
client applications, debugging information
System Developers
APIs
configuration, monitoring data
Operating Environment (JVM etc.)
System Managers (IT operations staff)

Figure 4.1: *A context diagram of the model service.*

4.2.1 Base Technology

The Sun Microsystems Java SE platform is chosen as the principal implementation language and runtime environment. The system is mature, robust, and well-documented. It offers a high degree of portability to different hardware and software environments, as well as a large selection of commercial and open-source software tools and libraries.

While the general architecture is not tied to this specific platform, some design decisions are influenced by it.

4.2.2 Distribution

As shown previously in 2.4.2, most existing building model servers are based on a two-tier or three-tier client-server architecture. There are two likely reasons for this. First, the tools and methods for creating such applications are widely available and proven. Three-tier has become so prevalent in distributed business applications that it is often seen as the "default" architecture choice.

Second, two/three-tier is a good fit for the requirements of model servers for design support. The primary task for such model servers is to store and maintain a model that is created by independent, user-operated "offline" applications that submit their work from time to time, similar to software revision control systems such as CVS. The applications are distributed by necessity, simply because their operators will work in different companies, locations, and of course on different computers (compare fig. 4.2).

On the other hand, our requirements of a building model server are driven by the need to have a single, centralised model updated by a range of data sources that continuously deliver raw data. Most applications – such as building systems control – usually do not check out a model copy from time to time, but must react immediately on model changes and use the model as an interface to the physical building. Such applications are generally long-lived, which means they operate 24 hours a day – similar to the *daemons* of the UNIX world. While they may be distributed on different computers, there is usually no pressing need for it (compare 4.3).

Classic two- or three-tier architectures are not a good fit for these requirements. Their design is primarily motivated by support for client distribution and flexibility of client code. As mentioned, client distribution is not a must-have requirement. On the contrary, it is a disadvantage for long-lived applications because server environments usually provide a far more robust, managed operating environment and the communications load

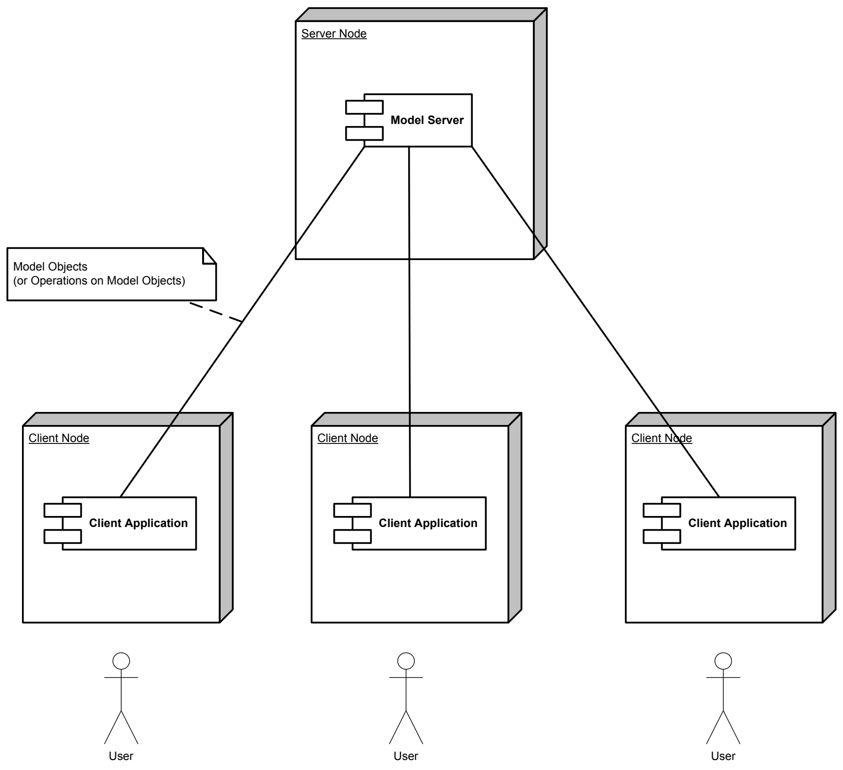


Figure 4.2: *A sample deployment view of a model server for design support.*

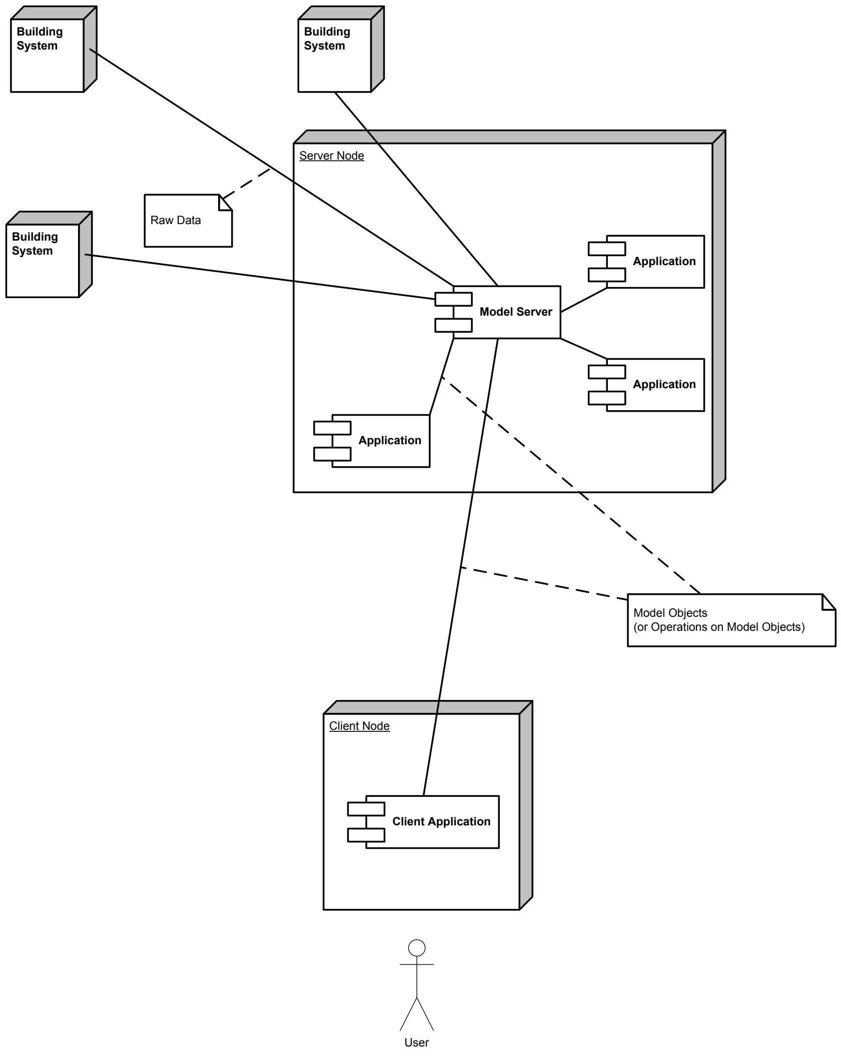


Figure 4.3: *A sample deployment view of a model server for operations support.*

between client and server can be heavy. Flexibility, on the other hand, is a must (compare 3.3.4): applications should be changeable without any disruption on the server.

What is needed, therefore, is a distributed architecture that allows applications to run where their data are for high efficiency, but is still flexible enough to allow remote on-the-fly modifications of these applications. At the same time, it should be inclusive enough to accommodate "classic" distributed client applications if needed.

4.2.3 Centralised versus Distributed Model

One of the first questions for decomposition is whether to represent the building as a collection of – more or less – loosely connected fragments, or as a strictly centralised data structure.

There are two related, but different aspects to this matter. The first, the *application* aspect, is about the logical view that model-using client applications see. Are they presented with a single, unified object model of the entire building, or with a network of model partitions? In the first case, random access to any part of the model is possible so that every object can be accessed by every application in the same way. In the second case, applications work on parts of the model and must explicitly switch context to other parts of the model if necessary. These context switches involve more than just following a reference, but possibly interprocess communications across networks. As an example, the system proposed by Sharples et al. (1999) comprises a network of embedded computers distributed across a building, each responsible for a single room.

One problem with the latter approach is to decide how to break the overall model into parts: a decomposition that works well for one application may not be suitable for another. For example, an inventory management application may be based on a decomposition of the building into organisational units. However, the boundaries between such units may not have any relation to the boundaries that are relevant to a heating control application (walls and floors) – in fact, they can be purely virtual. Some systems cross-cut buildings in a way that even seemingly obvious and "neutral" divisions into stories and rooms are hardly useful: consider lifts, staircases, and other vertical elements of a building. Moreover, modern open-plan offices are designed so that their internal layout can be changed easily with room dividers. Any decomposition based on a notion of "rooms" would have to be reconfigured accordingly.

Another difficulty with a split-up model is that it does not help efforts to take the "big picture" into account for local control decisions. Unlike simple room thermostat control, the idea in modern building control is to consider a range of external constraints to arrive at a control action that balances local goals (maintain set temperature, ensure appropriate workplace illuminance) with overarching, building-wide goals (such as: reduce energy usage, increase daylight usage, and others). This is related to the fact that the effects of many control decisions are not strictly local, as they affect the situation in adjacent rooms and floors – consider heating, for example.

It follows that a single, unified model view for applications is desirable, so that applications can decide based on their own specific needs which parts of the model they access, unhindered by an arbitrary partitioning.

The *implementation* aspect is about the actual storage of the model at run-time, which is not necessarily visible to applications. Here, keeping everything in one place (one process, one database) would also be desirable for the sake of simplicity. However, it may not be feasible to do so given a very large object network. Breaking up the model in parts assigned to different processes possibly on different computers can make sense for performance reasons. On the other hand, keeping mirrored (replicated) copies of the model could be done to improve availability.

For the system architecture, it can be concluded that a single, unified model should be presented to the application, but the physical design should be flexible enough to allow partitioning and replication if needed.

4.2.4 The Model is the Interface

The model is the only view of the building systems that client applications can get. In other words, it is an opaque layer (compare figure 3.1). Therefore, it must be rich enough to capture all available information that may be interesting to client applications, and it must be updated quickly enough to eliminate the need for special low-level access.

An alternative design may choose to allow clients low-level access to incoming or outgoing data, possibly in the form of processing hooks. Experience shows, however, that such “backdoors” tend to get over-used for supposedly temporary quick and dirty problem solutions. They are also detrimental to reuse when each client creates its own processing behaviours or views without sharing them through the model. Another problem to expect is that multiple client applications hooked into input processing would slow down model updates for other clients.

4.2.5 Avoiding State

"State is hell. You need to design systems under the assumption that state is hell. Everything that can be stateless should be stateless." —Ken Arnold (Venners 2002)

Keeping state in this context refers to distributing state information over multiple components, possibly operating in separate processes or on separate machines. A classic example of this is a TCP connection: both ends of the connection must keep state to ensure packet ordering, for flow control, and for reliable transmission in case of errors. Each incoming packet is handled differently based on the history of previously received packets. Both parties need to reserve some local memory for each connection and must be able to handle various forms of failure. As opposed to streams of unrelated packets, it is not easily possible to pass control of a connection to another process, or to distribute workload among multiple processes. Similar issues exist in other forms of session-based communication patterns between modules.

Such distribution of state among components is a force opposing the push for looser coupling (and less complexity). While its use cannot be avoided altogether, it should be limited to those occasions and places where it is absolutely necessary and directly reflects a functional requirement. Stateless components have a number of advantages, such as:

- Simpler testing and debugging. Operations inside a module depend only on the request that is currently processed. To test the module, sending one request per test case is enough. To debug the module, it is not necessary to follow an entire conversation history; it is sufficient to inspect the current operation.

- Better scalability. The number of requestors has no direct effect on resource consumption (given a constant rate of incoming requests, it is irrelevant whether they are generated by one sender or many). If a single task is overwhelmed with processing, the load can be easily distributed by setting up multiple tasks that take turns processing incoming requests. Moreover, stateless modules need relatively little initialisation and do not have to discard data after each finished session, which impacts garbage collection.

- Simpler semantics. Stateful communications have to follow a common protocol that is impossible to specify as a static model and can become quite complex, especially for exception handling.

- Harder to break. A communications partner that does not follow the agreed-upon protocol can – inadvertently or maliciously – disrupt operations on the other side, to the point of denial-of-service attacks. Many attacks on the TCP protocol (such as the well-documented “SYN flood” attack) exploit a combination of the facts that a) the remote party must reserve some resources for each connection, and b) it cannot easily decide when to free these resources, so it must hold them for some time to account for network delays and other temporary problems in order to achieve fault tolerance.

4.2.6 Interface Design

Plain Objects

As industry experience with technologies such as CORBA or Enterprise Java Beans (EJBs) shows, layers upon layers of proxies, adapters and interfaces to implement – often generated by development tools – are a frequent source of problems in the development of distributed systems, particularly when existing interfaces are changed. As an example, every single EJB requires at least an EJBHome interface and implementation, an EJBObject interface and implementation, an XML deployment descriptor, the actual bean class, and context objects (DeMichiel 2003). These are coupled fairly tightly: even a minor interface change in the bean class requires synchronised changes to most related interfaces and implementations. In fact, most of these objects are implemented by code generation, with even a trivial EJB resulting in hundreds of lines of deployed code, the vast majority generated and often hardly readable. From the developer’s viewpoint, this code appears to have little if any benefit.

Recently, there has been a broad move to rid the development process of these intermediate steps, reduce visible complexity and enable developers to work with “plain old Java objects” (POJOs) as much as possible (DeMichiel and Keith 2006). One of the basic rules of the model service’s architecture and design is to use such “plain objects” throughout whenever feasible, and to avoid code generation altogether.

4.2.7 Component Design: Reference Isolation

In a complex environment with a changing set of components and threads sharing the same process context, the design must limit the scope of object references and prevent unintended reference sharing. Components must take care to only expose interface classes to facilitate run-time implementation changes (4.7). References to internal mutable objects must not be "leaked" to avoid compromised encapsulation: object references returned to component callers should be either immutable or copies of the internal representation.

4.2.8 Loose Coupling Through Messaging

In software design, *tight coupling* relates to various appearances of the same design property: that code or data in one place is highly dependent on code or data in other places. For instance, when changes in a data structure of one part of a system necessitate changes in components all over that system.

As this has turned out to be a major problem in the development and maintenance of software systems, all significant milestones in the progress of software engineering have contributed to enabling *looser* coupling in some way: structured programming pushed for subroutines (instead of GOTOs) and other modularisation techniques. Object-oriented programming emphasised this further with its principle of encapsulation. In network communications, layered protocol stacks are an approach to establish clear, strictly enforced interfaces between communicating systems that hide implementation details. Here as well, the aim is to prevent the propagation of small, local changes in system elements to the whole system by containing them within explicitly defined bounds. Bass et al. (2003) generalise this concept as the "prevention of ripple effect" modifiability tactic.

Tight vs. loose coupling can also be seen as a temporal property of interaction between communicating tasks. In this context, tight coupling usually refers to a *remote procedure call* (RPC) pattern that emulates the semantics of a local procedure call. Consider a system with two concurrent tasks, both of which are involved in some computation. At some point, task 1 may decide that it needs to call a procedure provided by task 2. Just as in a regular procedure call, this means that task 1 passes control of the overall computation to the procedure in task 2 and is effectively suspended until the remote procedure completes. The execution of task 1 is therefore tightly coupled to the execution of task 2 – unless there is some form of timeout mechanism, task 1 might be stopped indefinitely until task 2 passes back control.

This temporal coupling can be loosened significantly by allowing *asynchronous remote procedure calls*. This would allow task 1 to submit its request to task 2 and immediately return to its own processing, while task 2 receives the call and commences execution of the requested procedure. Once finished, the results of the procedure may be returned to task 1 by using a callback mechanism (a reverse procedure call) or by task 1 actively polling for them. As opposed to synchronous calls, an asynchronous call thus does not force the caller into suspension. As a result, the overall level of concurrency is increased. This comes at the cost of increased complexity for the developer, who has to deal with losing the accustomed sequential ordering of events and a variety of exceptional situations (what happens if task 2 responds very late or never?).

Messaging is a generalisation of this asynchronous communication pattern. The most fundamental difference to RPC is that the data sent from one task to another is not necessarily a procedure call – it may be any chunk of data presumably understood by the recipient. Messaging facilities are typically provided by a messaging infrastructure that takes care of addressing (providing abstract communication channels called *queues*), guaranteed delivery, buffering, and other intricacies of asynchronous network communications. Unlike synchronous RPC, the messaging paradigm has been shown to scale very well in practice from intra-process communication up to enterprise application integration (Hohpe and Woolf 2003). Beyond point-to-point communication (see 4.4), most messaging infrastructures also support some form of message broadcast (one sender, multiple receivers) in the form of publish-subscribe messaging.

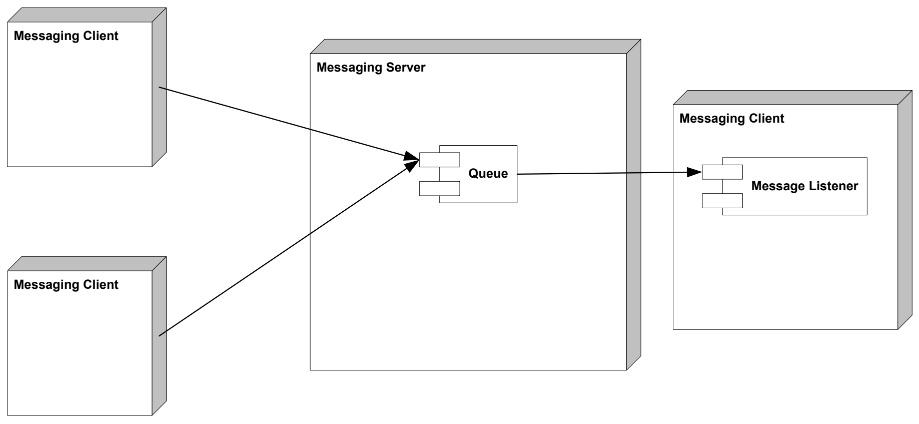


Figure 4.4: *An example deployment view of a typical point-to-point messaging setup. The clients to the left submit messages to an existing* queue *on the messaging server. The client to the right has registered itself as a consumer of this queue and gets the messages delivered by the messaging server.*

The conceptual scalability of messaging, however, does not mean it is a full replacement of synchronous remote procedure calls. In many situations, the semantics of synchronous calls are still needed. While it is possible to emulate RPC using messaging, the resulting code tends to be convoluted and less efficient than using "native" RPC.

On the Java platform, the Sun Java Message Service (JMS) specification provides an interface to a range of existing messaging infrastructures such as IBM's Websphere MQ and native implementations (Hapner et al. 2002).

Space-Based Computing

Although it shares some concepts with messaging, *space-based computing* is a fundamentally different approach to distributed computing. In the space-based paradigm, a virtual shared data container (the space) is visible to all participating tasks and can be accessed by them as if it were local memory (Carriero and Gelernter 1989). Tasks can add data to the space (the *put* operation) and *take* data from the space based on certain criteria. The details of distribution are completely hidden: a task does not have to know which other tasks are using the space, where they are located, nor how they are implemented. The space infrastructure handles all the details of providing the distributed shared memory. This form of collaboration is an instance of the Blackboard pattern (Buschmann et al. 1996).

JavaSpaces

On the Java platform, the JavaSpaces specification describes an infrastructure for object-oriented collaboration based on the spaces concept (Freeman et al. 1999). Tasks can put objects into the space and take objects based on their types and the values of their public fields[1]. Additionally, tasks may *read* objects (without removing them from the space) and register their interest in certain objects to be notified asynchronously when matching objects are put into the space (*notify*). JavaSpaces also support transactions and transparent persistent storage of the space.

One of the standard applications for using spaces is the master/worker pattern, used for distributed processing of work items. Tasks post work requests by posting items of work into the space, which are taken from the space by worker tasks, who then post the results back into the space. The particular benefit is that workers can be added anytime

[1]This is not a full-fledged query mechanism, however. Matching is restricted to *exact* (bitwise comparison) matching or *any* (wildcard) by supplying a single template object.

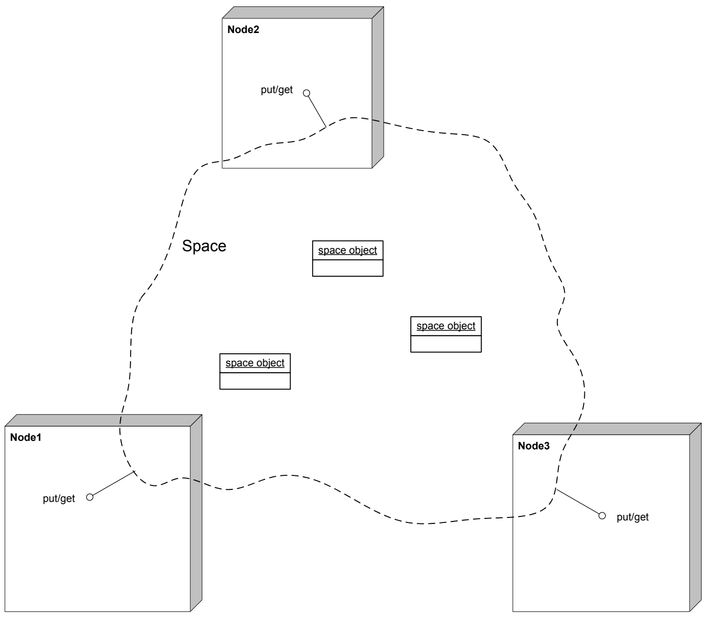


Figure 4.5: *An example deployment view of a space shared by three nodes. The space interface (put/get) is available through local method calls. Distribution is handled transparently.*

simply by starting them and having them wait for new work items. Unprocessed work requests are buffered by the space until they are picked up by one of the workers. The developer is freed from concurrency issues, as the *put* and *take* operations are atomic.

Issues with JavaSpaces

Spaces can be used to emulate messaging semantics. However, certain requirements such as maintaining order (first-in-first-out semantics for queues) must be hand-coded, e.g. using the *Channel* pattern described by Freeman et al. (1999). This requires transactions and multiple space accesses for each messaging operation and is therefore bound to be less efficient than a dedicated messaging solution that provides FIFO functionality. Specifically, many patterns providing complex distributed data structures require access to one or more singleton control objects in the space (e.g. an index object tracking a channel's head and tail). With growing numbers of tasks competing for access to these central objects, concurrency decreases and temporal coupling between the tasks tightens as they wait to take turns using the control objects. Emulating broadcast semantics (delivering one posted object to multiple recipients) requires similar efforts.

Most of these difficulties stem from the fact that the JavaSpaces specification provides very few guarantees on which objects are picked up when, and in which order. If a task requests to take an object with specified type and field values from the space, any random object matching the template may be returned. If a task repeatedly uses the read operation, it may read the very same object again and again even if other matching objects exist – or not, as the specification leaves this open. Due to the limited querying/matching semantics, simple requests such as "take the object with the smallest timestamp value available in the space" are not easily fulfilled. If a task has registered to be notified when an object matching a certain template is posted to the space, it will be duly notified of this event. However, there is no guarantee that it will be able to read or take that exact object – the notification does not contain a reference to the posted object or any other way to address it directly.

In summary, spaces are not a replacement for messaging, but serve certain applications very well. Specifically, they allow for elegant load distribution with minimal administrative overhead.

4.3 A Top-level Decomposition

A simple decomposition can be derived from the requirements. The following logical components must exist to support the required functionality (4.6):

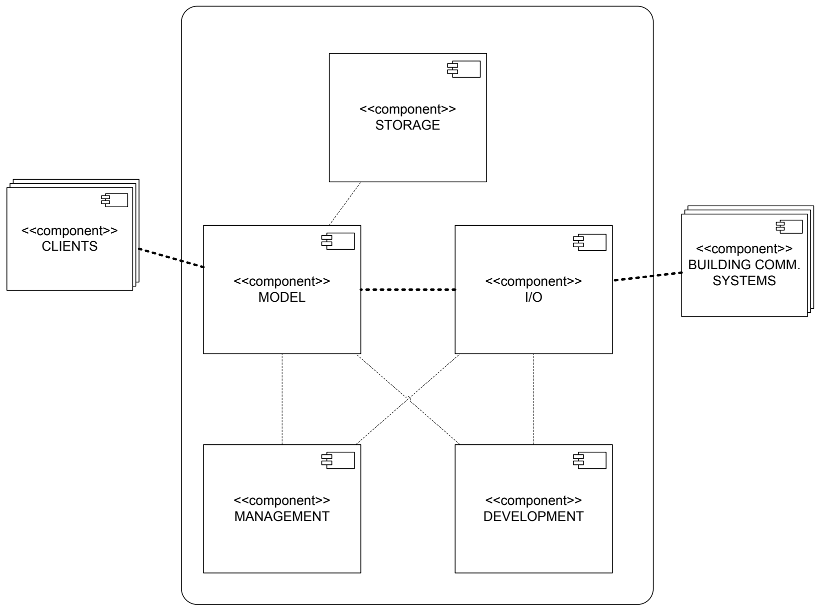


Figure 4.6: *Top-level logical components. Slashed lines denote interactions.*

MODEL This component holds the actual building model. Its main responsibility is to keep the model consistent and available for client access.

STORAGE The responsibility of this component is to keep the current state of the model, as well as past version, available in persistent storage for later retrieval.

I/O The I/O component is responsible for getting data in and out of the model. For data supplied by building communications systems, this implies converting incoming data to understand their contents, mapping the data to target objects, deciding which actions to take on which fields of the objects, and executing these actions. Similarly, outgoing commands triggered by commands on model objects must be converted to the appropriate actions in the building communication system's context.

DEVELOPMENT The development component must provide the necessary functions to aid in deployment, testing, and debugging of client applications.

MANAGEMENT This component provides the necessary functions to support monitoring and configuration of the system.

There is obviously a close relationship between the MODEL and I/O components, and in turn, the clients and communication systems that act as sources or destinations of the data flows. Across these three (or more) components, the vast majority of data are expected to flow in and out of the model. Considering the quality attributes relating to performance (3.3.1), particular attention must be paid to the design of this path.

4.3.1 Distribution

The logical view leaves the actual locations and communication methods between the components open. For optimal adaptability and modifiability, clients and building communication systems should be physically decoupled from the model component, as would be the case in a typical client-server architecture. While such distribution can also improve performance by offloading some tasks to separate hardware, it tends to incur a heavy cost in network traffic. On the other hand, placing the clients on the same hardware, or even within the same process context, reduces network communications to nil. The downside is that in such a "monolithic" setup, run-time modifiability may suffer.

As a design guideline, we identify two types of runtime behaviour in terms of model access patterns:

Batch behaviour An application component collects some model data, performs intensive processing on it, and returns some output data. One example is model-based lighting simulation (by ray tracing or radiosity), another is spatial reasoning (e.g. to generate space boundaries from tag locations). The object working set is typically fixed before the actual processing begins.

Interactive behaviour An application component keeps accessing a number of objects repeatedly, possibly reacting to events and changing the objects. It requires little processing power, but low-latency object access. One example is a lighting controller task that monitors workplaces and registers any relevant events that may occur, e.g. changes in occupancy or daylight. The object working set may change in reaction to object states or events.

Components with Batch Behaviour

Components with batch behaviour benefit from distribution to keep high CPU workloads off of the computer hosting the model service. This distribution is implemented using a spaces-based infrastructure. For instance, an application's request for lighting simulation can be posted to the space and subsequently picked up and processed by any connected machine running an instance of such a service. Once completed, the results are placed back into the space to be picked up by the client. This allows a simple and transparent load distribution that decouples components in time and space as much as desirable. Neither party needs to know anything about the other except how to access the common space and the signature of the relevant request and response objects. Components can choose a synchronous or asynchronous mode of operation: either posting requests and waiting for responses sequentially, or posting a batch of requests at once and coming back later to pick up the results. The latter scenario is particularly suited for simulation-based control programs, which frequently need to commission a set of simulations to select the best control decision. The advantage of using spaces instead of messaging here is that objects can be shared: common data used by multiple participants can be posted to the space just once, instead of duplicating them for each one. This also allows for simpler co-ordination between participants.

Components with Interactive Behaviour

The characteristics of components with interactive behaviour suggest a different approach. It is desirable to keep these components' code close to the data during runtime without losing the flexibility and loose coupling of the system by hard-wiring their code into the model service. One way to achieve this goal is to design an elaborate query language for the model: the main advantage of this approach is that components are not bound to any specific programming language, as long as they can submit well-formed query strings to the model service and process the results. However, the developer effort of translating query or program logic into an intermediary is considerable, and there would be significant communications overhead incurred by repeated queries and responses. Ideally, components should be able to access the model just like any other Java object.

This is achieved by implementing them as *agents*, which might also be called “mobile plug-ins”. Agents are objects that may be submitted to the model service over the space, where the are started as separate threads within the service process. They can directly access the object graph and use all public operations on the objects as well as a number of utility methods for traversing the object graph, retrieving historic versions, and communicating with other modules. Moreover, agents can register for events on specific objects to be notified of data updates. While agents in this context do not refer to mobile agents in the strict sense (as they do not migrate from system to system on their own), some of the advantages outlined by Lange and Oshima (1999) do apply just as well:

- They reduce network load by “packaging conversations” and moving code to data.
- They limit the effects of network latency to a minimum.
- They execute asynchronously and autonomously – which means that the agent can remain active even if network connectivity is lost or the originating computer is shut down.

As an example, the core of a control application can be sent to the model as an agent, where it can examine the relevant objects, derive a number of possible control decisions and send a batch of simulation requests (containing relevant model data) to the space. Using the results provided by one or more simulation services connected to the space, the controller can take appropriate action (e.g. opening a valve). Further control cycles can either be triggered in time intervals or based on update events (e.g. when a temperature sensor reading rises above a threshold).

For both batch and interactive access patterns, this design ensures modularity and flexibility while the specific runtime characteristics are taken into account. Naturally, some applications will be hybrids of these main types: such applications are split into two communicating, but separately distributed components.

Architecture Overview

Figure 4.7 shows a top-level architectural overview of the system. The model’s principal form of communication with building systems is through message queues. Its persistent storage is maintained by a separate database management system. Clients access the model – and vice versa – primarily by message-based communications. The design does not consider model distribution (partitioning): the model is kept in one process. However, it does offer some extensibility for replication (see 5.5).

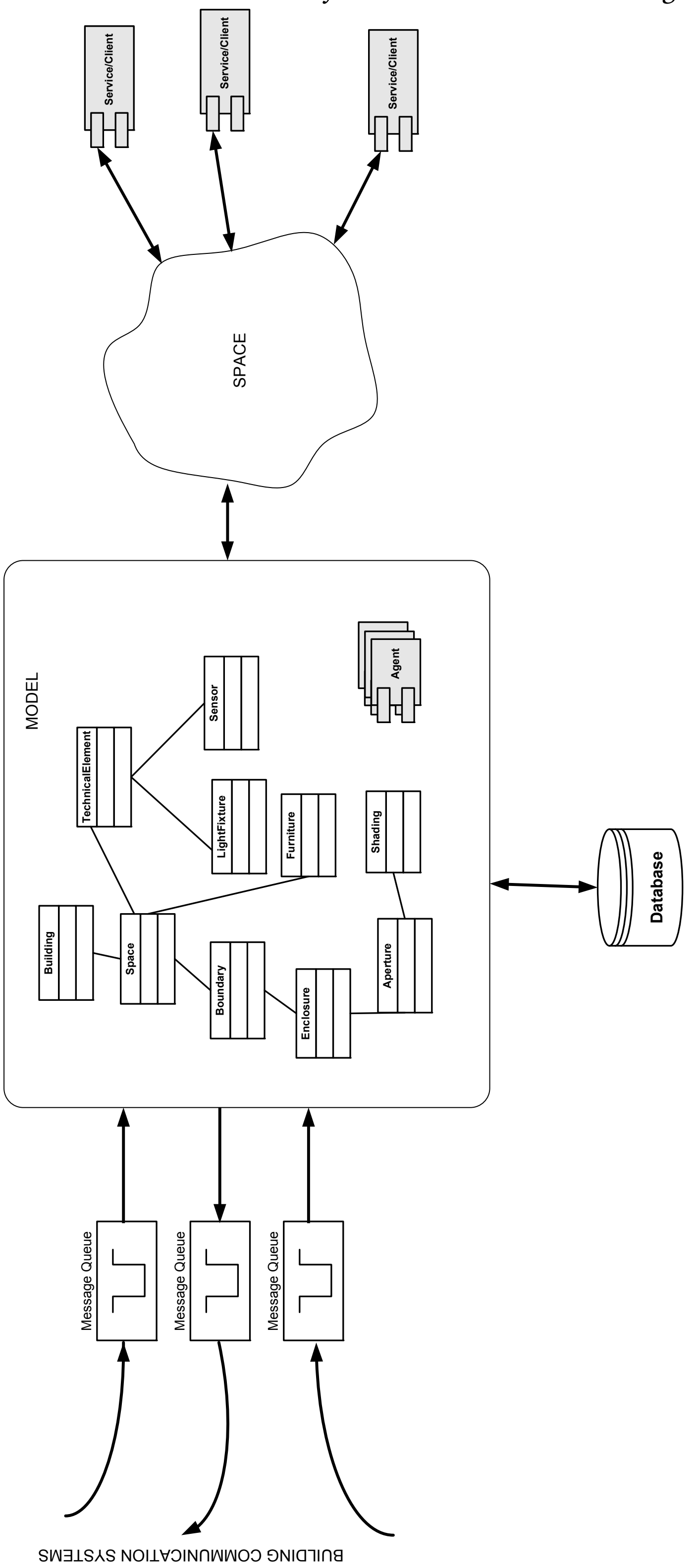


Figure 4.7: *Top-level architectural overview. Arrows denote data flows.*

The following sections of this chapter discuss the interface with the building communication systems, as well as the client interface. Specific issues relating to concurrent transactional access to the model, as well as some observations on its role as a temporal database, are discussed in chapter 5.

4.4 Building Data Interface

The building data interface's main task is to get information from the real world into the virtual world by acquiring data and triggering the necessary changes on the appropriate objects, and vice versa. The design is focused on sensors as its data sources, although it is general enough to allow other data sources (such as import from design software) as well.

4.4.1 General Considerations on Data Flows

Figure 4.8 illustrates on a high level the data flows between the model and the building systems. Data about the building's current status are received from the various building systems (sensors etc.) and then typically undergo a stage of pre-processing.

This processing may be a simple and short format conversion, but can also involve long-running, computing-intensive stages such as spatial reconstruction from sensor data (Suter et al. 2005). In consequence, there may not necessarily be one-to-one relation between building system updates and model updates: a model update could require collection of multiple sensor values (either a series of values from the same source, or a set of values from multiple sources) to result in a meaningful model update.

In the next stage, the processed data are used to update the model. This again can range from updating a single field of a single model object to a massive restructuring of the object network, including deletion and creation of multiple objects. Once updated, the model objects are available to client applications.

In the other direction, client applications may trigger commands on model objects representing controllable physical entities, such as motorised window blinds. The outgoing data volume and processing requirements are expected to be significantly lower than those for incoming data; the following discussion is therefore focused on incoming data under the assumption that outgoing communications are handled in the same manner, just in the opposite direction.

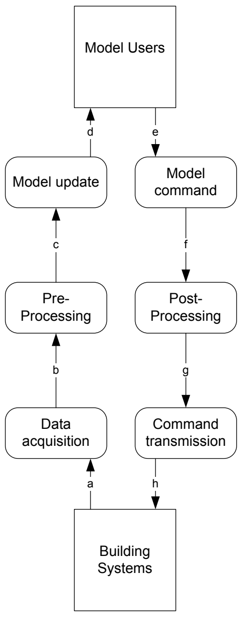


Figure 4.8: *A conceptual overview of data flows between model and building systems.*

Communications at this level are stateless. Specifically, there is no notion of a connection. This is a crucial point that aids in performance and flexibility, reduces resource consumption, and simplifies the design (see 4.2.5).

Who Drives Data Flows?

There are principally two modes of operation for getting sensor data into the model, depending on who initiates the data transfer. In *sensor-driven mode*, the sensor submits status data on its own volition: based either on a fixed schedule or a change-triggered mechanism, sensor data are sent to the model. The main advantage of this approach is that it simplifies things on the sensor's side: it operates only as a sender, not a recipient of information and consequently does not have to handle the intricacies of two-way communication. For simple sensors, this approach also reduces effort on the model side. The main disadvantage is that the sensor cannot adapt to the model's data requirements and might either overload it with more data than it needs or is able to handle, or it might starve it by not sending data when they are needed. Sensor-driven mode is most applicable for simple, low-cost sensors that generate little data for each update. It is also appropriate for sensors that generate data in irregular intervals, triggered by significant events in the physical world (e.g. occupancy sensors).

In *model-driven mode*, the sensor transmits data only by request from the model service in a polling pattern. The main advantage here is that the model gets exactly the data it needs and when it needs them, not more or less. However, this places a higher burden on both the sensor, which has to understand requests and react accordingly, and the model, which has to run a task that requests data as needed. Additionally, this mode of operation generates more network traffic for periodic updates than sensor-driven mode. Model-driven mode is most applicable for sensors that are needed infrequently or in unpredictable schedules, and that generate a higher amount of data for each update. Note that even in model-driven mode, the actual data reception works the same way as it does for sensor-driven mode. The difference is that some process must send a request to the sensor to submit data.

There may also be a combination of the two modes for certain types of sensor devices. Sensors could be programmed by the model service to transmit data based on a given schedule or whenever changes deemed significant by the model occur. This can solve the problem of increased network traffic for periodic updates, but requires more sophisticated sensor devices on the one side and more sophisticated knowledge of how to access and program sensors on the other side.

It follows that the design must be flexible enough to allow different approaches of driving communication between model and building system. To achieve this, the design for handling of incoming data assumes sensor-driven mode (which essentially means "data can come in any time from anyone"). If model-driven mode is required, the commands to trigger data updates are treated just as any other outgoing data.

4.4.2 Design

The building data interface's most natural decomposition is a sequence of stages, pipe-and-filters style.

1. **Data Acquisition** Data must be picked up from the data sources. The model service is designed to get its data from some form of message queue. To get data from an arbitrary building communication system, it is therefore necessary to provide such a queue and route messages to it. This is achieved using a separate adapter process that knows how to communicate with the BCS and the message queue.

In consequence, three separate tasks are responsible for transporting data from sensor to model object: the BCS, the adapter, and the model service itself (see figure 4.9).

2. Repackaging Once received from the queue, data must be repackaged for further processing. This is a lightweight operation that simply normalises all messages to an object structure that is independent of the used messaging application.

3. Target Localisation In the next step, the target model objects must be found. This can be a complex operation depending on message content. It may involve multiple objects and result in the need to create new objects, move existing objects, or delete objects.

4. Model Update Finally, the target objects must be updated.

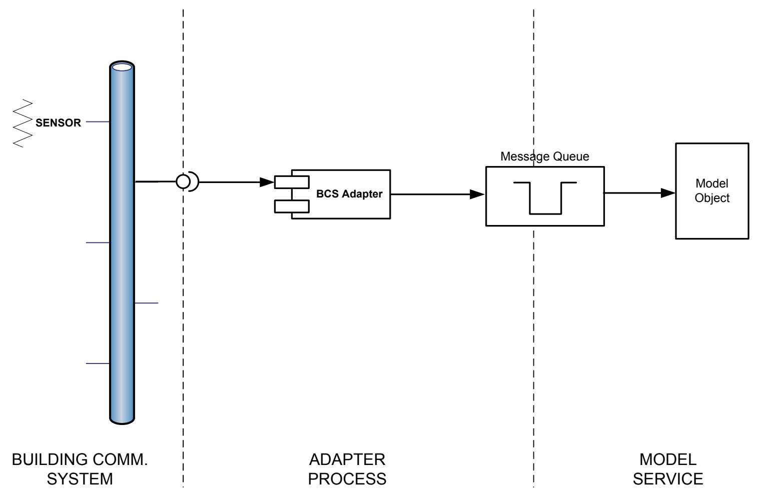


Figure 4.9: *Responsibilities for data flow from sensor to model object.*

A number of factors make the entire process more complex than its high-level description suggests. First, the various formats of incoming data from various sources must be understood. Second, a mapping of sensor data to objects must either exist *a priori* or must be derived dynamically from the incoming data. It may be necessary to query the model or run lengthy decision algorithms or heuristics to find the target objects. Third, updates that require multiple messages may have to be collected, assembled and applied together, which takes time and requires some temporary memory. Fourth, all behaviour in these stages must be fully reconfigurable during runtime, as a hard-wired data processing chain would not fulfil basic requirements of the model service (cf. 3.3). If the model, the data sources, the messages they send and the frequency at which they do so were known and fixed at design-time, input processing would be a matter of a few method calls. In this project's specification, all of these properties are variable.

Data Acquisition

In terms of functionality, data acquisition is simple: incoming messages are pulled from a queue and passed on to the next module. Messaging systems such as JMS provide an asynchronous call-back interface that allows message listeners to register to be notified of incoming messages. However, it is already at this first stage that the design must consider concurrency.

There is a limited number of input queues. Processing of messages is, as established above, nontrivial and can involve lengthy processing. To keep updates flowing into the model with low latency even when some of the updates take longer, multiple messages must be processed concurrently.

The design of web servers and CORBA object request brokers has some similarity to this design problem, and has been studied extensively (Schmidt et al. 2001). The main difference between web servers and the model service is that the latter does not implement two-way communication: messages are received and processed without returning a response to the sender .

Some frequently-used concurrency strategies include:

Thread-per-message A separate thread is created for each incoming message. While easy to implement, this approach tends to be inefficient due to thread initialisation and destruction. It is also hard to apply to multiple-message processing.

Thread pool A pool of pre-created threads is available; incoming messages are distributed among the threads. The pool can be of static size or resized adaptively depending on load levels.

Processes (per request, pooled): Similar to thread approaches, except that full processes are used. Spawning processes is a heavyweight operation compared to threads and incurs additional cost for interprocess communication. Benefits include isolation (a crashing process usually does not harm other processes) and possibly easier load distribution across multiple machines. This approach is not considered any further.

Reactor/select Some web servers employ a single-threaded model that utilises operating system facilities like UNIX's select() system call to operate on multiple connections quasi-concurrently (generalised as the Reactor pattern by Schmidt et al. (2001)). What this pattern implies is reliance on the operating system's "hidden" I/O multiprocessing. It therefore works well for I/O-bound tasks that spend most of their running time inside operating system functions, e.g. when reading files from disk. However, this is not expected to be the case for the model service.

Of these options, the thread pool offers the best balance of performance and ease of use. The thread pool design is a variation of the Leader/Followers pattern (Schmidt et al. 2001).

When using callbacks (e.g. the MessageListener interface for JMS (Hapner et al. 2002)), the messaging system calls a specific method of a registered message listener to deliver the incoming message. As this happens in the context of a messaging system thread, the called method should return quickly to avoid blocking further delivery. Therefore, the message (or a reference to it) is typically stored temporarily and a worker thread is notified to pick it up and commence processing. This approach is cumbersome and introduces at least one unnecessary context switch between the messaging system thread and the actual worker thread, plus some data copying and concurrency control mechanisms.

Instead, the Leader/Followers approach is to let the worker threads take turns waiting for new messages. The worker threads form a list, with the first thread (leader) using a blocking call to the message system that returns as soon as a new message is available. Once this is the case, it notifies the next free thread (follower) to move into its place and wait for the next message. It then proceeds to work on the received message and returns to the queue after finishing (Figure 4.10).

In our adaptation of this pattern, the workers are selected in MRU (most recently used) order to achieve a degree of CPU cache affinity. Instead of attaching to the queue's tail after processing a message, the thread is placed as the first follower. With this approach, the queue is effectively used like a stack. A practical side-effect is that the size of the thread pool can be tuned very easily by watching the usage counts of the

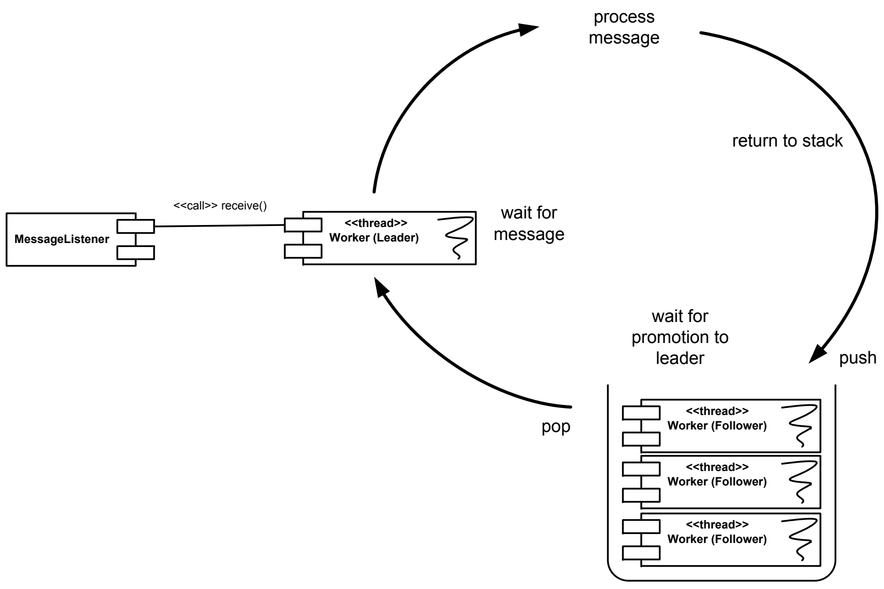

process message
return to stack
<<call>> receive()
MessageListener
<<thread>> Worker (Leader)
wait for message
wait for promotion to leader
push
pop
<<thread>> Worker (Follower)
<<thread>> Worker (Follower)
<<thread>> Worker (Follower)

Figure 4.10: *Life-cycle of worker threads.*

follower threads. If a number of threads are never used because they remain on the stack all the time, the thread pool size can be reduced. If, on the other hand, the stack's bottom is hit frequently, new threads should be spawned.

Data Repackaging

Once processing has been taken over by a worker thread, messages must be converted to a format that is understood by the following stages. This operation is usually lightweight and chiefly consists of wrapping the message into a standardised message class for further handling in the sense of an Envelope Wrapper (Hohpe and Woolf 2003).

The base message class comprises the actual message payload and header fields, including a history of stations passed so far (Figure 4.11). This history can, and should, begin with the originator. In the simplest case, and as a general fallback method, the message is wrapped without further inspection. However, header data can be extracted from the message contents if the type is known using Inspector objects. Such objects can be registered and de-registered at runtime (see 4.7).

At this stage, inspection should be kept to the absolute minimum required to derive

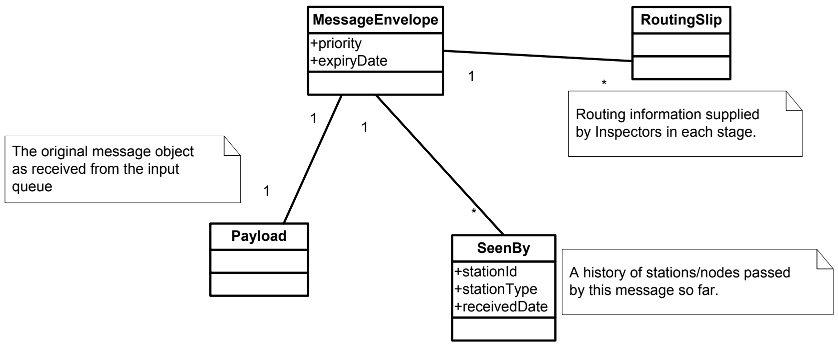

MessageEnvelope
+priority
+expiryDate
RoutingSlip
Routing information supplied by Inspectors in each stage.
The original message object as received from the input queue
Payload
SeenBy
+stationId
+stationType
+receivedDate
A history of stations/nodes passed by this message so far.
1
1
1
1
*
*

Figure 4.11: *Message envelope class diagram (simplified).*

meaningful header information. As a general rule, the process should not involve any other information than that contained in the message itself (e.g., no queries to back-end systems) and be kept to simple data conversion and copying. Complex processing, if necessary, is to be executed further down the pipeline.

Target Localisation

Multiple steps are taken to decide which objects are affected by the incoming message and which actions must be taken on these objects. In the simplest case, there is one target object for the message, which is determined from the message headers and a table or b-tree look-up. This is the case when sensors have unique identifiers, the corresponding model objects already exist, and they are indexed according to the ID.

In complex cases, multiple messages must be correlated, and potentially lengthy model queries or decision algorithms are executed to determine the affected objects and decide on the necessary update actions. One example of such a scenario is the reconstruction of spatial geometries from sensed positions of tags that are attached to walls, floors, and other surfaces (Suter et al. 2005). This reconstruction requires a multitude of tag locations and requires processing time in the range of seconds to a few minutes.

Target localisation consists of two major steps: first header inspection and then content inspection. The latter is only executed if the message has been flagged accordingly during header inspection. Each step consists of iterating through a list of Inspector objects that can be registered and de-registered at runtime, calling their inspect() methods. The result of inspection is a list of references to Updater objects (which may be empty if the message matches no Inspector), which is added to the message's RoutingSlip.

The rationale for separating different inspection stages and the actual updates is to allow different workflows for target localisation and updating. Figure 4.12 shows three variants. In the first case, the first Updaters are run immediately after header inspection, with content inspection running in parallel. The idea here is to begin updates as soon as possible, assuming that content inspection takes somewhat longer than header inspection. In the second case, the same steps are run sequentially, with updating in two stages. The third variant runs all cumulated updates of the inspection activities in one step.

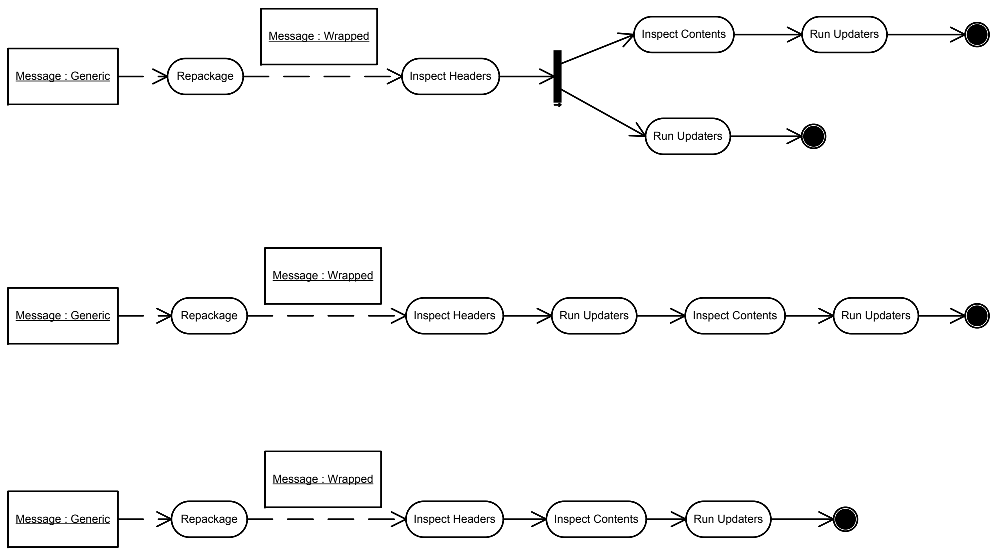


Figure 4.12: *Three different workflows for processing messages.*

The choice of workflow depends on the messages and usage patterns to expect at runtime. To ensure flexibility, workflows can be defined in user-supplied classes and selected at model service initialisation.

Model Update

The same message can trigger multiple update actions on different objects, with different complexities. In the spatial reconstruction example mentioned above, a sensed tag location results in a simple update to the corresponding tag object, as well as a series of complex update operations on the object or objects that the tag is attached to. It is desirable to execute simple updates as soon as possible without waiting for long-running updates. This can be achieved by sorting the updates' execution schedule according to their expected running time, or by executing all updates concurrently. Both choices have their problems: estimating expected running time is not always possible with reasonable accuracy and may add substantial overhead before the actual update even starts. Forced concurrency is not effective if most updates are short, with significant overheads introduced by thread initialisation and communication.

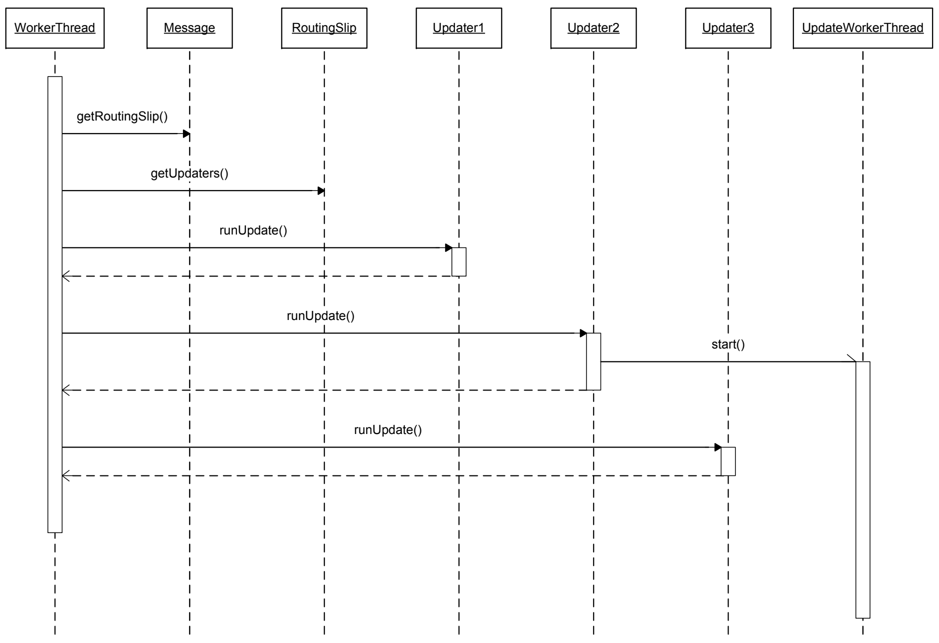


Figure 4.13: *Example sequence of updates with one long-running update.*

In the chosen design approach, updaters are called sequentially and are expected to return quickly. Complex updaters are responsible for spawning worker threads or using pooled threads when necessary (Figure 4.13).

Message Correlation

Message correlation relates to the requirement of combining information from multiple messages before a meaningful model update can occur. This means that a number of messages must be held in temporary storage and any related messages must be acquired before proceeding with the update. Some problems arise here: first, temporarily-stored messages may pile up and claim significant amounts of memory – during that time, they are not easily available to other model clients. The temporary buffers would have to be flushed periodically to prevent losing messages, prompting the question of what do to with them (use for updates or throw away). Second, finding related messages can prove nontrivial if messages from different sources must be correlated. If messages are held by one correlator and required by another, which in turn holds messages required by the first correlator, deadlock results.

Supporting message correlation is therefore in direct conflict with performance requirements (cf. 3.3) and would complicate the design considerably. One of the basic premises of the model service architecture is that the model is the interface (4.2.4). Consequently, no temporary pre-buffering of messages occurs: each message is expected to result in an update to a model object. Correlation of multiple messages occurs by correlating the model objects' values, not by intercepting messages before they result in updates. This also reduces state-keeping in the processing pipeline.

As an example, the spatial reconstruction algorithm outlined by Suter et al. (2005) relies on the positions of multiple tags that are attached to objects and building elements (walls, ceilings, floors). Instead of intercepting the tag positions before they are used to update the corresponding "host objects", the tags are modeled as individual objects and updated individually. The spatial reconstruction task is decoupled from message processing: it acts upon the tag objects' properties.

Handling Priorities

A typical requirement for real-time applications is that different messages are processed with different priorities. To achieve this, the first question is how to decide which messages have which priority. If the relative priorities are known to the data source or the adapter (i.e. before the messages enter the building model service), a simple solution is to create dedicated queues for each priority and set up a separate thread pool for each. If the priorities are determined dynamically for each message, based on an inspection of its contents, an alternative approach is to let the processing threads raise or lower their own priorities accordingly. In Java, threads can change their own scheduling priority using the Thread.setPriority() call (unless explicitly disabled by security policy). Additionally, the database concurrency control mechanism may offer different transaction priorities.

4.5 Application Interface

4.5.1 Agents

As discussed before, clients access the model as agents. Agents are threads that reside within the model server process and act either autonomously, or on behalf of a remote application, with which they may communicate. In this context, the process of delivering agent state and code from a remote node into the server and setting the agent up to be started is called *deployment*. Inside the server, agent deployment and lifecycle support is provided by the AgentMgr component.

It should be noted that the current design and implementation of agents in the model server is restricted to the specific application's needs and not comparable in generality and functionality to full-fledged agent systems.

An agent object must implement the java.lang.Serializable as well as the Agents interface:

```
public interface Agent {
    enum State { STOPPED, RUNNING, SUSPENDED }

    void setId();
    String getId();
    Properties getProperties();

    void setAgentEnvironment(AgentEnvironment env);

    // current agent state
    State getState();

    // contains actual agent code
    void start();
    // request agent termination (async)
    void stop();

    // request temporary suspension (async)
    void suspend();

    // wake up from suspension (sync)
    void resume();
}
```

Deployment and Undeployment

Deployment comprises delivering the agent code (classes) and its state (object data) to the server. The interface for agent deployment is a dedicated message queue that accepts deployment messages.

The process is illustrated in figure 4.14. Agent object data are included as a byte array containing an image of the object obtained by serialization (see 4.6.1). This approach is necessary to prevent classloader bootstrap problems on the server side, when an incoming message is automatically deserialized by the messaging framework: as the code for the incoming objects is not known at this stage, the object cannot be restored yet. It is similar to the approach exemplified by Java's java.rmi.MarshalledObject class.

The agent class code is supplied by providing it to an HTTP server as one or more JAR (Java archive) files and transmitting their URLs together with the deployment message. A custom classloader instance is set up for each agent, augmenting its codebase with the given JAR files; the separate classloaders afford a degree of isolation between agent threads. An advantage of the URL approach is that it prevents unnecessary retransmission of class code, especially when many agents share the same utility code. The disadvantage is that agent deployment therefore requires two separate communication channels. If this is a critical problem, the deployment message could be augmented to include JAR files with little effort.

Agents can be undeployed on their own volition or by an undeployment request from the remote client.

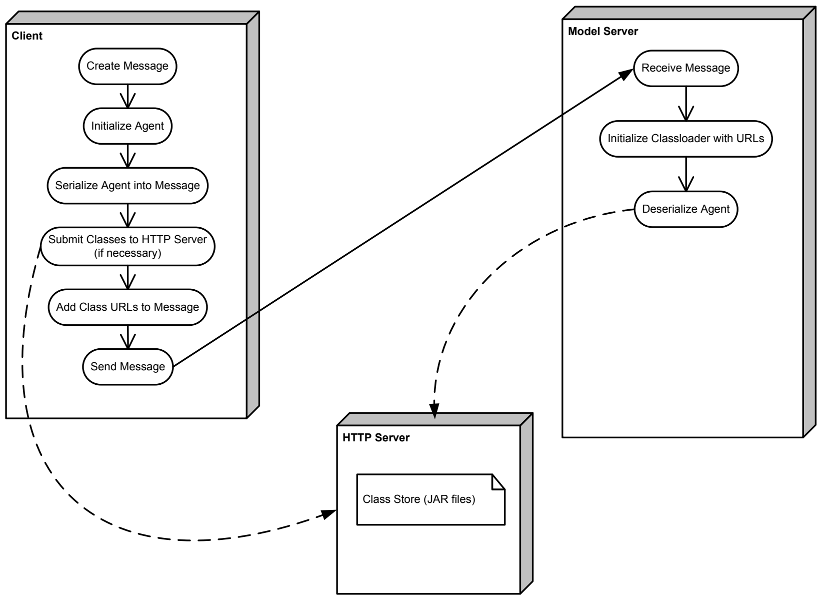

Client
Create Message
Initialize Agent
Serialize Agent into Message
Submit Classes to HTTP Server (if necessary)
Add Class URLs to Message
Send Message
Model Server
Receive Message
Initialize Classloader with URLs
Deserialize Agent
HTTP Server
Class Store (JAR files)

Figure 4.14: *An overview of the agent deployment process.*

Runtime Environment

Once deployed and instantiated by the DeploymentHandler – the component responsible for lifecycle management of agents – the agent is passed a reference to an AgentEnvironment object, registered by its ID with the singleton AgentRegistry, and started by calling its start() method within a newly created Runnable wrapper.

To handle planned outages and quick recovery without manual re-deployment of applications (see 3.3.2), agents can be placed in suspension. The suspend() method signals to the agent object that it is about to be suspended. Typically, the agent will then stop all its operations and prepare itself for the invalidation of all external object references. Once all agents are suspended, they are serialized to disk storage.

When server operations resume, the stored agent objects are deserialized similarly to a regular deployment, but started using the resume() method. The agent must then recover all external object references, but can continue with the internal state it stored prior to suspension.

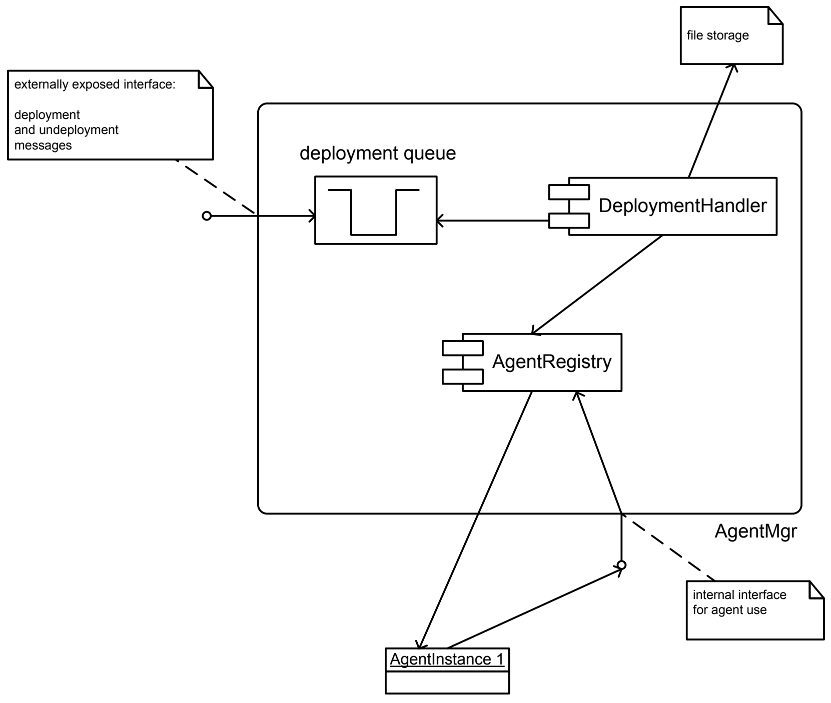


Figure 4.15: *The deployment manager component.*

4.6 Messaging Facility

To allow efficient intra-process as well as inter-process communications, a simple generalised messaging component is provided within the model server. For intra-process use, the messaging component allows other modules to register central message queues, identifiable by string IDs. Messages can be pushed to the queue's head and retrieved from the tail in first-in-first-out fashion. The implementation is a thin wrapper around Java's ConcurrentLinkedQueue, a FIFO implementation optimised for high concurrency (Goetz et al. 2006, chap. 5). In order to prevent memory overflows caused by inactive message recipients, the queues are bounded by the wrapper so that messages that would exceed the queue's maximum length are rejected[2]. Each queue carries a string ID which can be used for lookups.

To simplify communications with remote processes, queues can be mapped transparently to external queues provided by the messaging framework used for this purpose. The mapping requires the creator of the new queue to pass the external queue's name and creates a proxy queue object. An example is shown in fig. 4.16.

4.6.1 Issues in Intraprocess Communications

When passing messages – or, more generally, any object references – between otherwise independent threads, they may become indirectly related by sharing references to the same object (sometimes called *aliases*). While such reference-sharing is generally unproblematic for immutable objects (e.g. java.lang.String), it can result in adverse effects that are very difficult to retrace in the case of regular, mutable objects unless this interaction has been considered in the design of all participating threads (compare 4.2.7). Typical symptoms include:

1. Objects may suddenly change their state without any apparent cause and possibly at different times when the program is tested repeatedly.
2. Objects may exist much longer than expected, causing memory and resource leaks.

[2]The wrapper keeps its own count of put/get operations to avoid calling the expensive size() method of ConcurrentLinkedQueue. The bounding is designed for speed and minimal locking, using an atomic variable without additional locking. It may therefore be imprecise, allowing the queue size to overshoot the limit slightly or blocking before the limit is reached. For the purpose of resource usage limitation, this is acceptable.

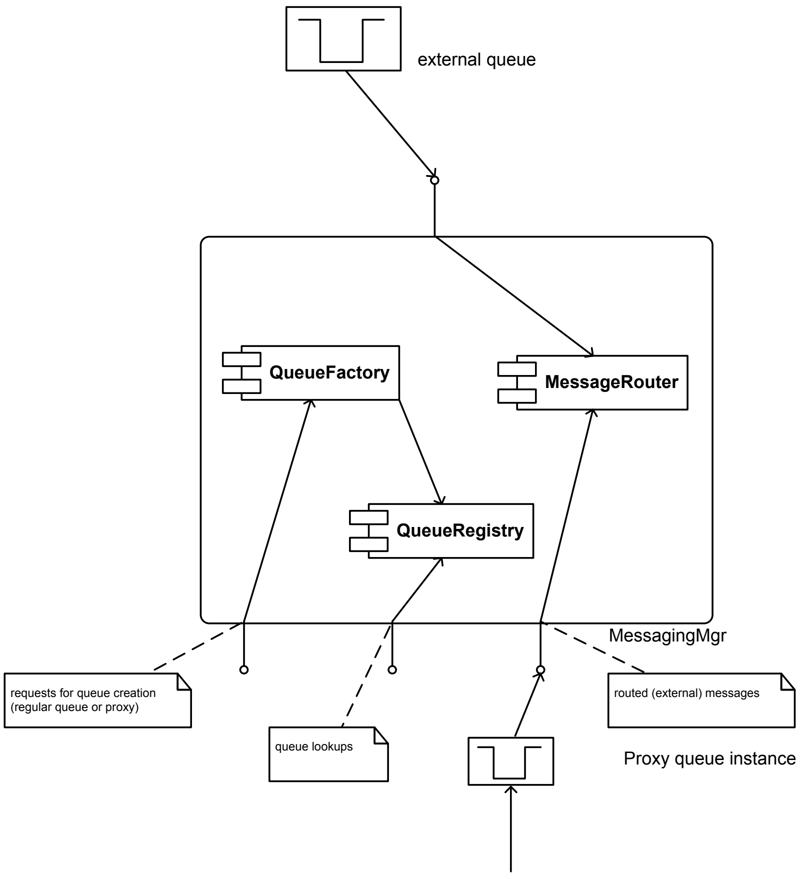


Figure 4.16: *The messaging manager component.*

3. Objects may be in invalid states because they are not designed for thread safety.

Two general strategies for preventing these issues exist.

Passing Immutable objects

As stated above, one approach is to pass only references to immutable objects between threads. While this approach is simple, elegant, and safe in theory, it proves difficult to enforce in practice as there is no built-in support for immutable objects in the Java language. That is, it is not possible to rely on language runtime support to ensure that a given object is immutable: immutability, although it has been an important technique in Java development since its inception, is purely a convention of the language user. Researchers have addressed this problem in a number of experimental Java extensions, none of which have been incorporated into the standard language yet – see e.g. Tschantz and Ernst (2005) for a recent example including discussion of earlier efforts.

Passing by value

An alternative approach is to make sure that objects are copied when passing them to other threads, emulating pass-by-value semantics for objects. That is, when a reference to object A is passed to another thread, an exact copy of A is created and it is a reference to the new, copied object that is actually given to the other thread. Java offers two mechanisms for creating object copies: cloning and serialization.

Cloning

The Java approach to cloning is two-fold. A Cloneable marker interface exists "to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class." (Sun 2004). The Cloneable interface does not define any methods; the actual cloning code is left to the clone() method of the Object class. However, this Object.clone() method is marked as protected and must be overridden with a public implementation to enable cloning from other classes – by convention, such an implementation simply calls super.clone() to invoke a bit-wise exact copy of the object. This bit-wise copy is not applied recursively to any referenced objects. The conventional semantics of Java cloning are therefore those of a "shallow copy", although it is feasible to implement a deep-copy mechanism within this framework.

Java cloning is only a partial solution to the reference sharing problem. Given shallow-copy semantics, the problem is not guaranteed to be eliminated, but rather deferred: the cloned object is likely to comprise not only primitive types, but also hold references to other objects, which are cloned as well and thus again shared between sender and receiver. Moreover, the Cloneable interface is not implemented by most classes defined in the Java standard library including frequently used classes such as String – and, as explained above, it is not even a guarantee that a public clone() method exists in a given class. A method accepting Cloneable arguments thus cannot rely on being able to call the clone() method on the passed object unless it performs a prior run-time check using reflection.

Serialization

Java serialization is a mechanism intended to write out the state of an object to a stream so that it can be reconstructed later on by deserialization. This can be used to store object states in persistent storage or to transmit object states over network connections, as used in the Remote Method Invocation (RMI) framework. It may also be used to create object copies within the same thread by serializing to a temporary in-memory buffer and reconstructing object copies by deserialization.

Just as in cloning, one element of serialization is a marker interface (Serializable). Unlike cloning, however, a default serialization implementation exists and is used automatically for all classes that implement Serializable, but do not overwrite the writeObject() and readObject() methods. This default implementation performs a deep copy, in turn serializing all referenced objects that implement Serializable as well[3]. The resulting representation of the object graph state is self-contained, as it does not contain any references to other objects. The semantics of serialization do not guarantee that it succeeds on all Serializable objects, but that it signals certain breaches of contract: if a referenced nontransient object is not marked as Serializable, an exception is thrown at runtime.

Temporary serialization appears as an excellent candidate for emulating pass-by-value semantics as it ensures that no references are inadvertently shared. It is supported by the language core and implemented by many Java runtime classes. However, it still requires some attention on the developer's part. For instance, deep copying carries a risk of inadvertently dragging an entire complex object network along with the passed

[3]Unless they are exempted from serialization by designating them "transient."

object – even though the recipient may only be interested in a single field of the passed object. This could be prevented by either using specialised "transport objects" with limited references to other objects, or by intentionally breaking those references upon serialization with an appropriate implementation of writeObject() and the affected data access methods.

While it may be the best option in terms of functional requirements, the runtime performance of serialization must be considered. Compared to the other options – cloning and passing immutable objects –, it tends to be computationally very expensive (see table 4.1).

	elapsed time [ms]		
virtual machine	reference	clone	serialized
Sun 1.5.0 06-b05 Hotspot Client	277.1	423.9	12811.2
Sun 1.5.0 06-b05 Hotspot Server	197.4	315.7	9716.7
IBM J9 VM 2.3	213.9	378.4	11471.3

Table 4.1: Comparing the performance of passing by reference, passing cloned object copies, and passing object copies created by in-memory serialization and deserialization. The passed object contains only primitive types, no object references. Times given are the average of 30 consecutive test cycles (after one discarded cycle to allow for virtual machine warm-up) comprising 200,000 calls each, measured on a 1.9 GHz Intel Pentium M single-processor machine with 512 megabytes of RAM running Microsoft Windows XP Build 2600.

The performance of Java serialization can be improved by using custom serialization mechanisms that omit some of the standard mechanism's features, such as compatibility with different Java virtual machine versions. For older versions of the Java VM (1.1 and 1.2) Philippsen et al. (2000) demonstrated serialization time reductions by 80–90% using a highly optimised implementation.

It should be noted that even though the raw message-passing performance of immutable objects is far superior to object copying, its use can cost performance in other places. Since immutable objects cannot be changed, change of object state can only be represented by creating new objects. If object data change very frequently, the effort to create new objects and garbage-collect old ones may become significant, possibly compensating the savings due to reference passing versus serialization in some cases[4]. This dilemma is reflected in the Java standard library by the existence of StringBuffer and StringBuilder as mutable complements to String.

Design Decision

As shown above, the mechanisms for handling reference sharing differ quite significantly in both performance and capabilities. Finding the appropriate solution thus depends very much on the exact usage pattern. In a general-purpose message passing system, making assumptions about usage patterns is highly speculative. The best solution appears to be implementing a safe default behaviour (in this case: serialization), but optionally allowing the use of a faster and riskier mechanism: in this case, passing plain object references and leaving it to the user to ensure that they point to immutable objects. Thus, the messaging system's put() operations allow the caller to override the default serialization behaviour to pass unchanged object references.

4.7 General Facilities

4.7.1 Runtime-Changeable Code

To provide the required runtime modifiability, component implementations as well as certain other code (such as processing chains of the building data interface, 4.4) can be changed at runtime (figure 4.17). Components are exclusively accessible through a façade object that delegates the requested calls to the concrete implementation class or classes. These implementation classes are loaded and initialised by an ImplLoader object that encapsulates class loading. Currently, two implementations of this loader exist for loading classes from a local directory (used for loading at model server initialisation) and for loading classes submitted at runtime through an incoming message (see 4.5.1). To allow seamless code exchange, the component façade acts as a gateway that blocks client calls while the implementation is changed.

The sequence for the replacement of an existing implementation is shown in figure 4.18. The ComponentFacade uses a read-write lock to allow concurrent component calls, but block all access while the component implementation is switched. A new

[4]This depends not only on the actual application, but also on the runtime system. Recent versions of the Java VM have improved both object creation and garbage collection performance significantly (Goetz 2005).

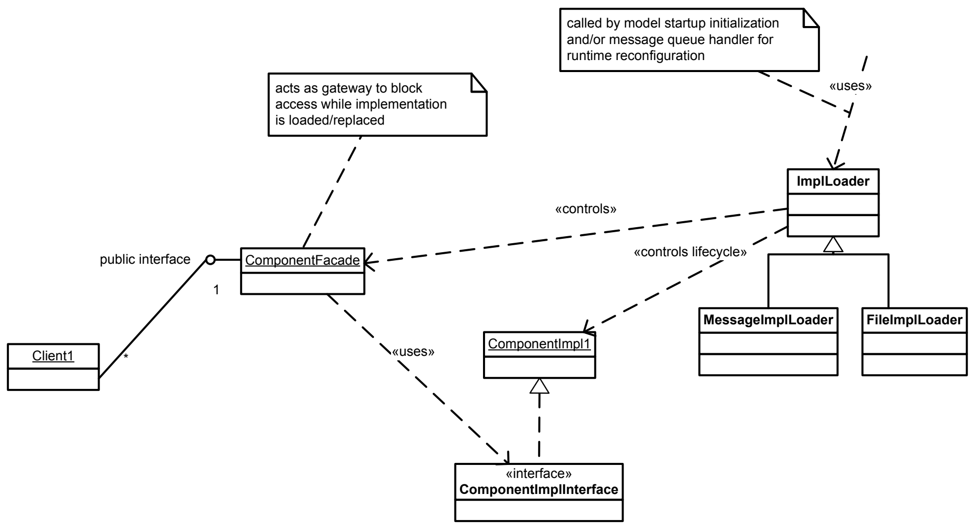


Figure 4.17: *Classes involved in run-time changeable component implementations.*

component implementation is loaded (using a new instance of the custom classloader), initialised with a reference to the ComponentRegistry (see 4.7.2), and given a reference to the current implementation to copy its state as needed.

Component implementation classes must take care not to expose internal references to clients. Specifically, they should only return copies of internally-used objects or immutable objects (compare 4.6.1). Otherwise, clients may hold on to references used by a component implementation that has already been superseded by another.

4.7.2 Component Registry

The initial point of contact for any piece of non-fixed code in the model server (including agents) is the ComponentRegistry. This simple, central singleton keeps references to all available components; it is therefore the only reference required by newly deployed code to navigate the model server environment.

4.7.3 Management and Development

The management and development components should provide support for system operation, maintenance, and code development (e.g. debugging). In the prototypical

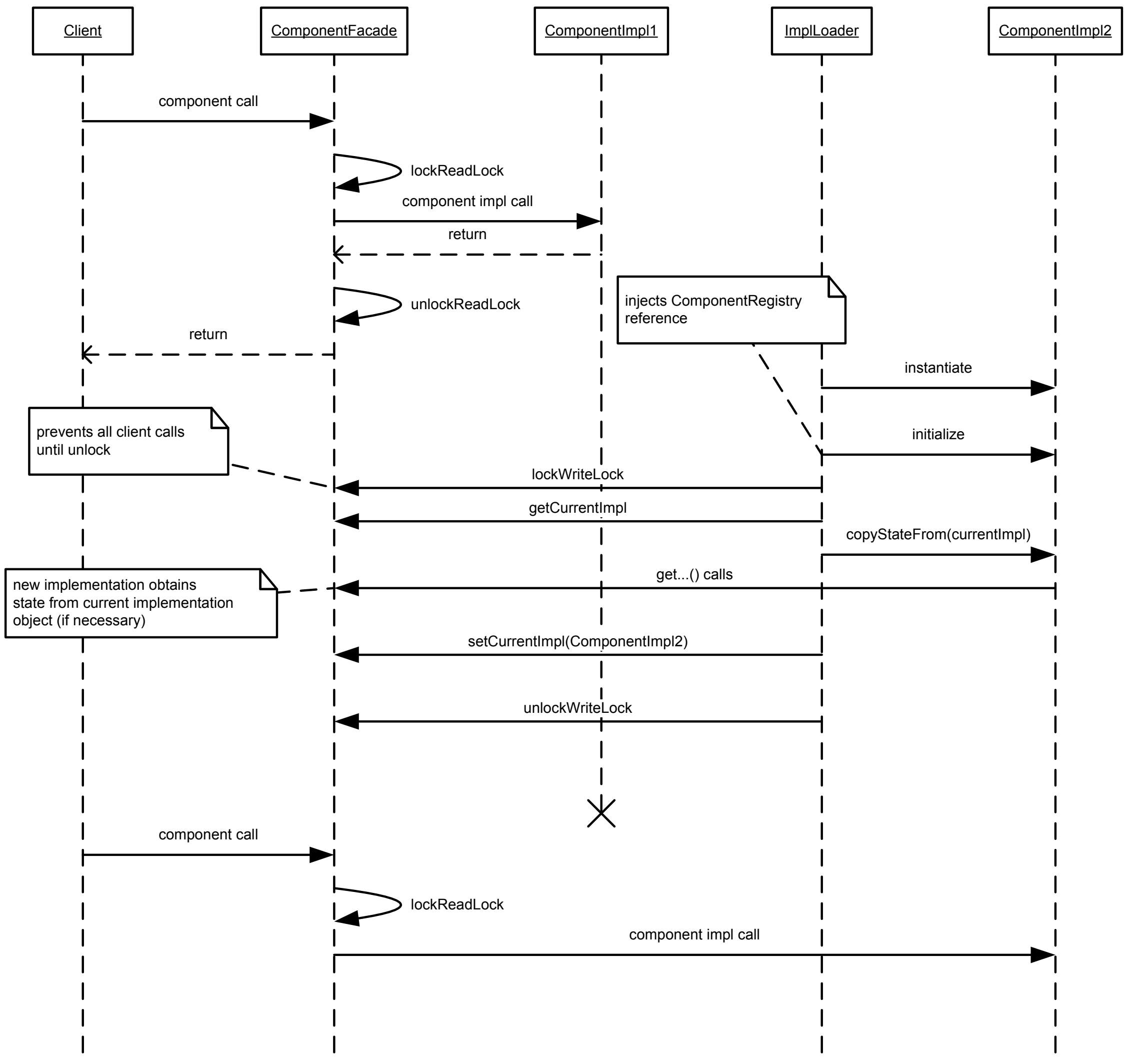
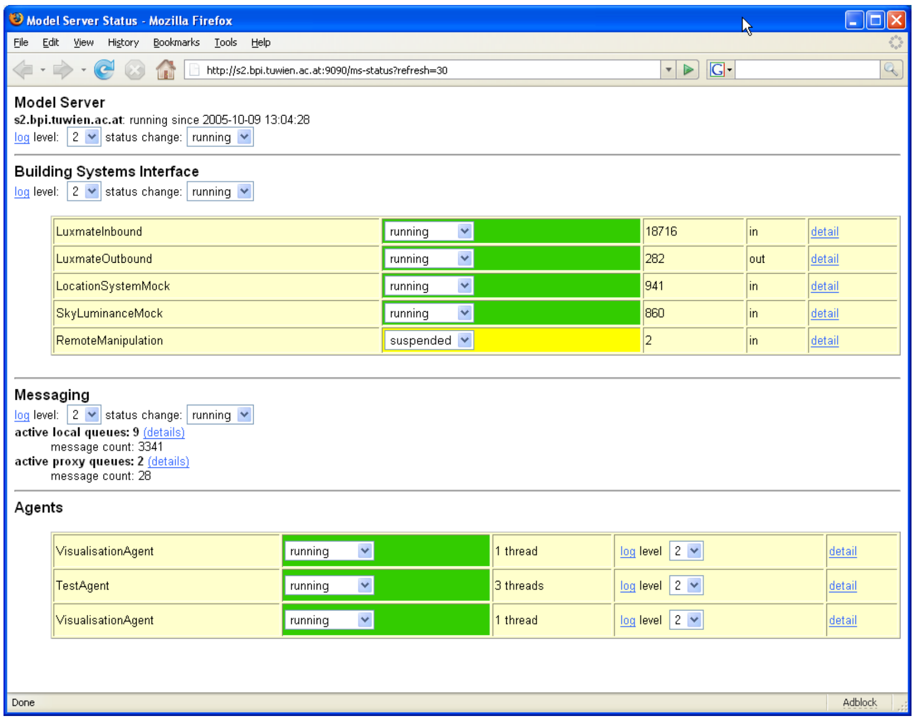


Figure 4.18: *Sequence: Regular component operation and installing a new implementation.*

implementation, a simple web-based application provides an overview of the server's status and allows monitoring and control of the entire server as well as installed components and agents (figure 4.19).


Model Server Status - Mozilla Firefox

http://s2.bpi.tuwien.ac.at:9090/ms-status?refresh=30

Model Server
s2.bpi.tuwien.ac.at: running since 2005-10-09 13:04:28
log level: 2 status change: running

Building Systems Interface
log level: 2 status change: running

LuxmateInbound	running	🟩	18716	in	detail
LuxmateOutbound	running	🟩	282	out	detail
LocationSystemMock	running	🟩	941	in	detail
SkyLuminanceMock	running	🟩	860	in	detail
RemoteManipulation	suspended	🟨	2	in	detail

Messaging
log level: 2 status change: running
active local queues: 9 (details)
message count: 3341
active proxy queues: 2 (details)
message count: 28

Agents

VisualisationAgent	running	🟩	1 thread	log level 2	detail
TestAgent	running	🟩	3 threads	log level 2	detail
VisualisationAgent	running	🟩	1 thread	log level 2	detail

Done Adblock

Figure 4.19: *Prototypical web-based system monitoring and management console.*

4.8 Chapter Summary

In this chapter, the system's overall architecture was outlined. The key points in summary:

- At the core of the architecture, a central live building model is maintained as an object network.
- The architecture recognises two groups of client applications, those with interactive and those with batch characteristics. This distinction guides its approach to distribution: interactive applications run as agents within the model server, with direct access to model objects, while batch applications can be distributed to other nodes for workload distribution. The distribution is based on "spaces", achieving a high degree of transparency and flexibility.
- Performance and scalability are supported by concurrency and stateless design of the system's I/O paths from and to building systems. Loose temporal coupling for higher concurrency is gained by using asynchronous communications whenever feasible.
- Modifiability and availability are supported by designing for run-time replaceable code from the ground up.

4.8.1 Requirements Overview

Table 4.2 lists the main requirements of chapter 3 and those sections of the architecture that primarily deal with them.

ID	Description	Definition in Section(s)	Design/Implementation in Section(s)
FR1	Model	3.2.1	5
FR2	Online/Offline Model	3.2.1	5.5
FR3	Model History	3.2.1	5.5
FR4	Model Input	3.2.2	4.4
FR5	Model Output	3.2.2	4.4
FR6	Client Isolation	3.2.3	4.5.1
FR7	Client Cooperation	3.2.3	4.5.1, 4.6
FR8	Transactions	3.2.3	3.2.3
UC1	Insert New Object	3.2.1	5.4.1
UC2	Retrieve Object Copy	3.2.1	5.4
UC3	Retrieve Object Tree Copy	3.2.1	5.4
UC4	Change Object	3.2.1	3.2.3
UC5	Command Object	3.2.1	4.4
UC6	Execute Transaction	3.2.1	3.2.3
UC7	Publish/Subscribe on Object Change	3.2.1	5.4.2
PER	Performance & Scalability	3.3.1	5.2, 4.4
AVA	Availability	3.3.2	4 (4.5.1)
APO	Adaptability and Portability	3.3.3	4.2.1
MOD	Modifiability	3.3.4	4.7

Table 4.2: Requirements Overview

5 The Model as a Temporal Database

The core component of the building model service is an object-oriented, temporal, main-memory database. It is *object-oriented* by virtue of storing plain Java objects. It is *temporal* by maintaining full histories of each object and the entire model's state, unlike snapshot databases that store only the most recent version of an object. As the current version of the building model is kept in main memory and unlike in most database management systems, client applications get direct access to actual object references, it is a *main-memory database* (Garcia-Molina and Salem 1992). However, previous versions are kept in persistent storage; the mechanics of this storage and retrieval are delegated to a third-party database management system.

In this chapter, the design and implementation of the model core is discussed. The design is driven by functional as well as nonfunctional requirements: in fact, it turns out that quality attributes relating to performance and scalability dominate most design decisions.

5.1 Building Model

In the prototypical implementation used for this dissertation, the System Object Model (SOM, Mahdavi et al. (2002), see also Figure 2.6) is used as the reference building model. As opposed to the complex Industry Foundation Classes, SOM is a fairly straightforward object hierarchy and serves well as an example implementation that has also been tested in earlier work for similar applications. The design of the model service is largely independent of the used model schema, although a few general assumptions are made to guide the design:

- The model is a directed acyclic graph of objects.
- The model is reasonably fine-grained, providing a level of granularity that is practical for most uses: there is neither a single "house object" that contains thousands of properties describing the entire building, nor a mass of "single-property" objects that separately capture parts of parts of things down to some micro-level.

- There is only a single model, i.e. one object graph following one model specification. The model service is not specifically designed to support multiple models of the same building. However, care is taken not to introduce design decisions that prevent support for this scenario.
- Sensors and actuators are considered first-class model objects that are represented explicitly. This means that if a room contains a temperature sensor, the room temperature is not modeled as a property of the corresponding space object. To get the room's temperature, a model user must locate the space object, find those temperature sensor objects that are located inside, query their current values, and calculate the room temperature. This may add some complexity to the model and its usage, but is necessary to capture the building as faithfully as possible and keep the model flexible enough for all sorts of applications. In general, it is not the model service's responsibility to maintain physical relationships between model objects.
- Each model object has at least one ID that is unique among all objects (past, current, and future) in the model.

Although SOM and the IFC are referred to as *object models*, it should be noted that they are actually *data models* using some properties of object modelling, specifically inheritance. No actual behaviours are specified, unless the enforcement of constraints and possibly getter/setter methods are counted as such.

5.2 Handling Concurrent Model Access

As the model server is intended for massive multi-user access (cf. 3.3), it must be able to cope with concurrent transactions which may include both reads and updates to one or more model objects.

5.2.1 Correctness Criteria

In databases, transactions are series of read and write operations on database objects. If multiple transactions are to be executed, it is desirable to interleave their operations so that they effectively run in parallel. However, such interleaving can lead to undesirable effects that break the desired correctness of each transaction. One such anomaly is the

Lost Update: Assume two transactions P and Q that operate on the same object. Both transactions read its current value, increment it by 1, and write the incremented value back to the object. If the initial value is 0, a non-interleaved execution history could be

$$Pread(0), Pwrite(1), Qread(1), Qwrite(2)$$

yielding the expected result, 2. An interleaved history might be

$$Pread(0), Qread(0), Pwrite(1), Qwrite(1)$$

giving an incorrect result, as the first update by P was missed by Q.

The goal is therefore to schedule the transactions's operations to allow for concurrency, but ensure that the outcome is correct. Various concepts of correctness exist, such as sequential consistency and linearizability (Herlihy and Wing 1990). In database systems with concurrent access to multiple objects by multiple threads, the most commonly used notion of correctness in literature is *serializability* (Papadimitriou 1979). In plain language, this is the intuitive idea that any execution schedule of transactions must be equivalent to some serial ordering of transactions at the granularity of a full, atomic transaction – i.e. all operations are ordered so that the effect is the same as if the transactions take turns, without interleaving individual operations. There are some variants of serializability; for practical use, the most relevant ones are strict and conflict serializability.

In some cases, a history is only serializable if a somewhat counterintuitive reordering takes place, i.e. in the equivalent serial schedule, transactions are executed in different order than in the original schedule (Papadimitriou 1979, p. 644). *Strict* serializability restricts the possible serializable schedules by prescribing that no re-ordering of already ordered histories can occur.

A *conflict* occurs when two or more transactions access the same object, and at least one of the operations is a write. In consequence, any non-conflicting sequences of operations can be re-ordered without affecting the transactions' results: if one history can be transformed into another by such reordering of non-conflicting operations, the histories are *conflict-equivalent*. If a history can be transformed to a serial, conflict-equivalent history, it is considered *conflict serializable*. An interesting property of this correctness definition is that while deciding whether a history is serializable is NP-complete, deciding conflict serializability is not.

In commercial database systems, serializability is often relaxed to avoid concurrency bottlenecks, at the cost of accepting certain anomalies to occur. The ANSI SQL 92 standard introduced a classification of four isolation levels ordered by the range of concurrency anomalies that are acceptable in each level. Berenson et al. (1995) extended this classification, introducing the notion of *snapshot isolation* (SI). In short, the underlying principle of SI is that a transaction sees exactly the (committed) state of the database that existed in the instant that it was started; two concurrently-executing transactions therefore never see the state generated by the other transaction. While SI does not guarantee serializability, it still avoids most of the well-known anomalies (such as Lost Update) and has therefore been adopted in many database systems.

Which correctness criteria to apply is largely application-dependent and may also have to be decided on a case-by-case basis. Barghouti and Kaiser (1991) discuss a range of different correctness notions and concurrency control algorithms with a view to applications that involve long transactions, such as CAD and software development collaboration. One of their conclusions is that serializability is often too strict a correctness measure and may have to be relaxed somewhat to accommodate the usage patterns in collaborative work. They suggest that the user should have some control over which form of correctness the database enforces. Fekete et al. (2005) introduce a static-analysis approach to decide for a set of transactions whether using snapshot isolation is enough to get serializability – this, however, requires an analysis of all participating transactions before they are executed.

In many database applications, database designers, database administrators, and application programmers know about each other and can coordinate their work. In such situations, using relaxed correctness criteria to improve performance is feasible. However, if the database is intended as a public interface to client applications that are unknown to the database designer, it is sensible to enforce rather strict correctness criteria. For the building model server, *serializability* is therefore chosen as the desired isolation level.

5.2.2 Design

Requirements

The design choice for the transaction processing algorithm is driven partly by the required correctness criteria, partly by the usage patterns. Specifically, the ratio of read-only transactions and the transaction lengths can have significant impact on the performance of concurrency control. The following assumptions are made:

- Concurrent update transactions affecting the same set of objects (i. e. competing for writes) are rare.
- Read-only transactions are frequent.
- A significant portion of read-only transactions are long-running and affect multiple objects.

Lock-Based Concurrency Control Methods

One of the oldest and still most popular concurrency control mechanisms for database access is two-phase locking (2PL), a method that achieves strict serializability (Sethi 1982, Eswaran et al. 1976). For single objects, 2PL is easily described and easily implemented: all transactions follow a protocol that consists of one growing phase (in which locks can only be acquired) followed by one shrinking phase (in which locks can only be released). Among 2PL's problems are the possibility of deadlock, which requires additional deadlock-resolving methods, and its poor concurrency. Specifically, 2PL has a tendency to "starve" writers if many readers access the same object. For systems running processes with different priorities, as is often the case in real-time applications, this may result in priority inversion when high-priority tasks must wait for low-priority tasks to release their locks.

Locking, in general, becomes significantly more complex for hierarchical databases as in the case of a building object model: an update to an object may invalidate the states of some (or all) of its descendants. Consider, for instance, an operation that shrinks a space containing a number of objects (Figure 5.1). When the space is shrunk, some of the object positions and dimensions may become invalid, as they are left outside of the containing space. The transaction must correct these positions inside the space-shrinking transaction to avoid showing an inconsistent state to other applications. In the shown example, some contained objects must be changed (such as the meeting room's dimensions, or the topmost desk), while others can remain as is (such as the table in the meeting room, or the other desks).

A simple consistency-preserving approach to this problem is locking the entire subtree that descends from the changed object. A tree-locking algorithm proposed by Silberschatz and Kedem (1980) allows more selective locking of the object graph, guarantees serializability and is deadlock-free. However, it may still lock a large part of the model if

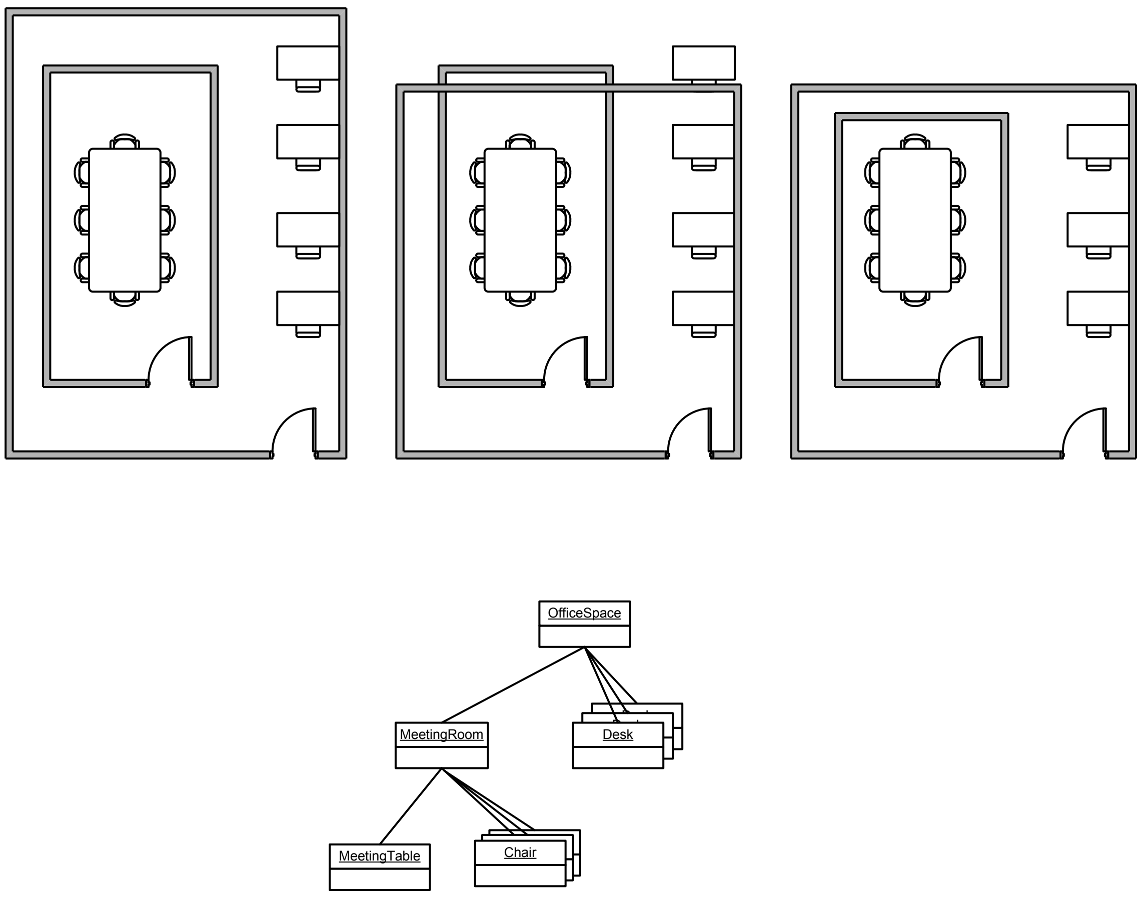


Figure 5.1: *Transaction: shrinking a space*

the locked object is near the model root and the affected descendants are far away from it. A large number of separate locks may have to be maintained, as the tree-locking algorithm requires that locks must be set on each node on the path from the root to each descendant object of interest.

5.2.3 Multiversion Methods

Multiversion concurrency control (MVCC) is based on the principle that updates to objects are implemented by creating new object versions instead of changing a single object instance, reducing the need for synchronisation in many usage patterns (Reed 1978, 1983, Bernstein and Goodman 1983).

Specifically, in most multi-version algorithms, "the idea is to permit long read-only transactions to read older versions of objects while allowing update transactions to create newer versions concurrently" (Carey and Muhanna 1986). Readers can therefore choose to work on older object versions to avoid synchronisation expenses. The main performance advantage of MVCC is therefore in situations with a mix of many readers and few writers on the same object – a fairly typical usage pattern for many databases, and also the expected usage pattern of a building model.

Counted among the downsides of MVCC are the somewhat higher implementation complexity and, most importantly, the cost of creating, storing, and possibly pruning old object versions. However, as keeping all historic object versions is a functional requirement of the model (see 3.2.1), this is not an issue for our purposes.

A number of different multiversion concurrency algorithms have been proposed, varying in their runtime characteristics, implementation complexity, and correctness criteria. Most of the algorithms are derived from two archetypes:

Multiversion Timestamp Ordering – MVTO (Reed 1983, 1978). This is generally considered the classic MVCC algorithm and uses timestamp comparisons throughout instead of locks for concurrency control. A detailed description of an MVTO algorithm is given below.

Multiversion 2-Phase Locking – MV2PL (Chan et al. 1982). This extension of two-phase locking follows the standard 2PL protocol for update transactions with the addition of new versions for each update and adapts MVTO's timestamp ordering principle for read transactions. A recent derivate of MV2PL is Multiversion Query

Locking, which allows relaxed consistency for increased performance (Bober and Carey 1992). Just as 2PL, MV2PL is prone to deadlocks.

Given the expected usage patterns of the building model with relatively little contention between writers and the desire to choose a deadlock-free algorithm to ensure limited response times, an MVTO approach was chosen.

5.2.4 Multiversion Transaction Ordering

The concurrency control mechanism used for the building model is based on the one first proposed by Reed (1983, p. 17) and studied by Carey and Muhanna (1986) as a simplified variant of a more general algorithm.

1. Each object is viewed as a sequence of versions. Each update results in a new version of the object; old versions remain available indefinitely. Old versions, once created, are never changed.

2. Each transaction T_i is issued a startup timestamp STS_i. Startup timestamps represent transaction time (Salzberg and Tsotras 1999) and are guaranteed to be unique and increasing in strict monotonicity: two transactions cannot have the same startup timestamp, the one that was started later has a higher value of STS than the one that was started before.

3. The most recent version of each object O_i has a read timestamp $RTS(O_i)$ that contains the startup timestamp of the youngest reader of the object, and a write timestamp $WTS(O_i)$ that contains the startup timestamp of the youngest writer of the object. The write timestamp is effectively the version number of each object version.

4. Write requests by a transaction T are only granted if $STS(T) \geq RTS(O)$ and $STS(T) \geq WTS(O)$, otherwise they are rejected. If a transaction's write request is rejected, it must be restarted.

5. When a write request has been granted to a transaction, it is said to be *pending* until the transaction commits or aborts (put differently, the object "has a pending write").

6. As long as an object has a pending write, any read or write request on that object is blocked (suspended).

7. Read requests may be blocked, but are never rejected. Blocking of readers only occurs if the most recent object version is read and it has a pending write. Reads of a non-recent version of an object are never blocked. Generally, read requests by a transaction T are directed to an object version that has $STS(T) \geq WTS(O)$. Readers may opt to read a non-recent version to avoid blocking when necessary (fallback reads).

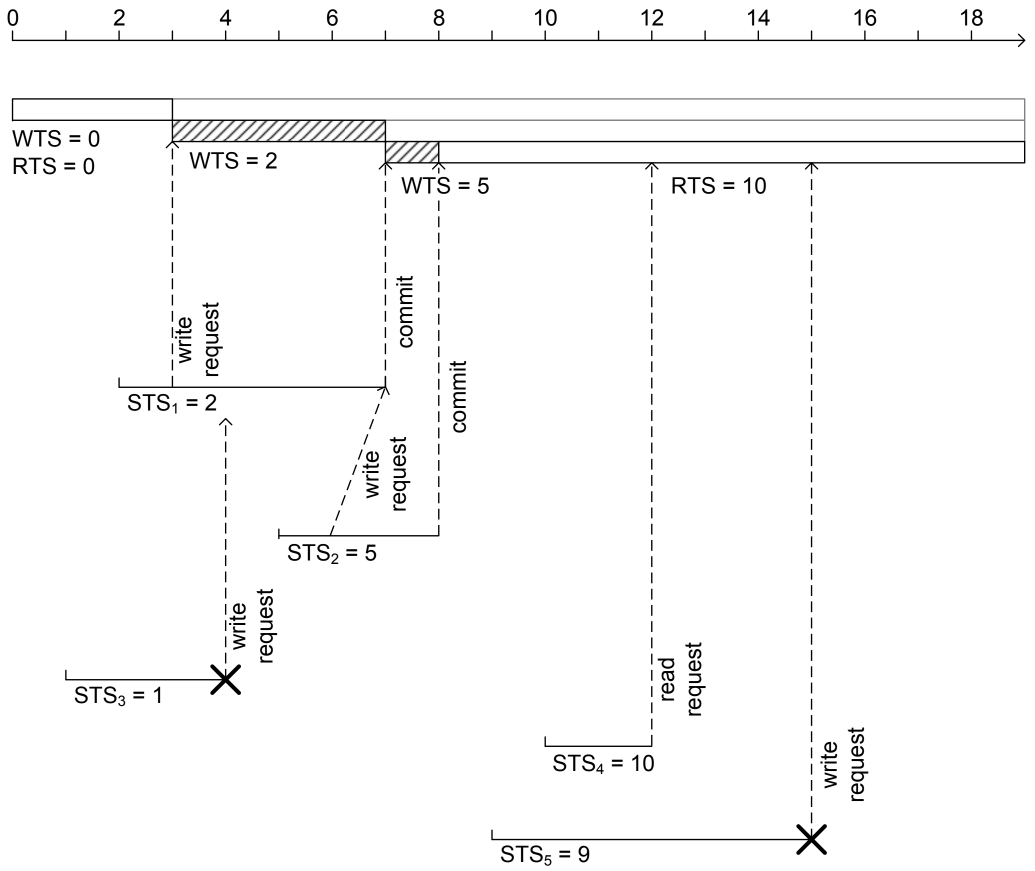


Figure 5.2: *Multi-version transaction ordering: some typical sequences. The time arrow at the top of the diagram shows "ticket" counter pseudotime. Note that the ticket counter may be advanced by other events than those depicted (e.g. other tasks running in parallel) and is generally not required to advance in increments of exactly* 1.

Figure 5.2 illustrates some typical transaction sequences in the MVTO scheme.

- At the beginning, an object with $WTS = 0$ and $RTS = 0$ is assumed to exist, depicted as a horizontal bar. A transaction T_1 starts and is assigned a startup timestamp $STS_1 = 2$. It issues a write request on the object, which is granted because the condition of item 4 (above) holds. This results in a new object version with $WTS = 2$ with a pending write (depicted as a hashed pattern in the diagram).

- Some time later, transaction T_3, which was started earlier with a timestamp $STS_3 = 1$, requests a write. This write is rejected because item 4 is violated; the transaction is aborted and may be attempted again at a later time.

- Transaction T_2 is started with $STS_2 = 5$. It issues a write request on the object, which is deferred because the object still has a pending (uncommitted) write request – this deferral would also apply to a read request on object version $WTS = 2$. A read request to the old object version $WTS = 0$ would be granted immediately, as old versions are always read-only and thus free to read any time by any number of concurrent transactions.

- T_1 commits some time later, which allows T_2 to immediately go ahead with its deferred write request and create a new object version with $WTS = 5$. Later, T_2 commits.

- Subsequently, T_4 is started with $STS_4 = 10$ and issues a read request on the latest object version ($WTS = 5$), which is granted immediately.

- Afterwards, T_5, which was started earlier with a timestamp of $STS_5 = 9$, issues a write request. This request is rejected because item 4 is violated: the transaction is aborted and may be retried later.

A Note on Terminology: “Optimistic” Concurrency Control and MVTO

With a low likelihood of conflicting writes, *optimistic* transaction processing tends to be at an advantage compared with *pessimistic* variants. In optimistic concurrency control, transactions are generally assumed to go through without interference, but must be restarted if conflicts are detected at (or before) commit-time. Pessimistic algorithms, on the other hand, assume that conflict will occur and aim to prevent it in the first place.

The computational cost is thus distributed differently: optimistic methods tend to be lightweight in non-conflict situations, but incur high cost once conflict is detected (as the entire transaction must be rolled back and restarted). Pessimistic methods always expend some time for concurrency control, regardless of whether conflict occurs or not[1].

In the strict sense, optimistic concurrency control (OCC) refers to a class of methods that do not check for conflicts until they are ready to commit a transaction (Kung and Robinson 1981). Transactions are divided into three phases: a *read* phase in which the state of the objects is read and writes are performed on transaction-local copies ("copy-on-write"), a *validation phase* in which it is checked whether the transaction can commit serializably, and finally the *write phase* when the actual commit occurs by writing the contents of local copies to global storage. Transactions are ordered using unique transaction numbers; these numbers are assigned at the end of the "read" phase.

MVTO, on the other hand, checks for conflict as soon as a write is attempted on an object. The transactions are ordered by their startup timestamps, which are assigned before the transaction starts. The method is weakly optimistic in the sense that it is also based on conflict detection (as opposed to locking, which is a method of conflict *prevention*). It does not, however, perform the entire transaction in local storage before attempting to commit it. In MVTO, validation is simpler than in OCC, as it is enough to compare two timestamps to detect a conflict, while OCC validation requires checking the write sets of all concurrently executing transactions. In MVTO, each object may have only one set of pending writes assigned to exactly one transaction. In OCC, each transaction has its own write set: this results in higher overall memory requirements for concurrent write transactions on the same objects.

Basic Implementation

The algorithm chosen for this implementation is based on the one proposed for the SWALLOW project, a simplified version of the general algorithm outlined by Reed (1983).

A TicketDispenser singleton object is used to obtain unique startup timestamps for transactions.

In the algorithm as implemented for the model service core, each object carries a write timestamp, containing the startup timestamp (WTS) of the last writer to the object

[1]However, resolving deadlock situations can require costly restarts even in pessimistic approaches.

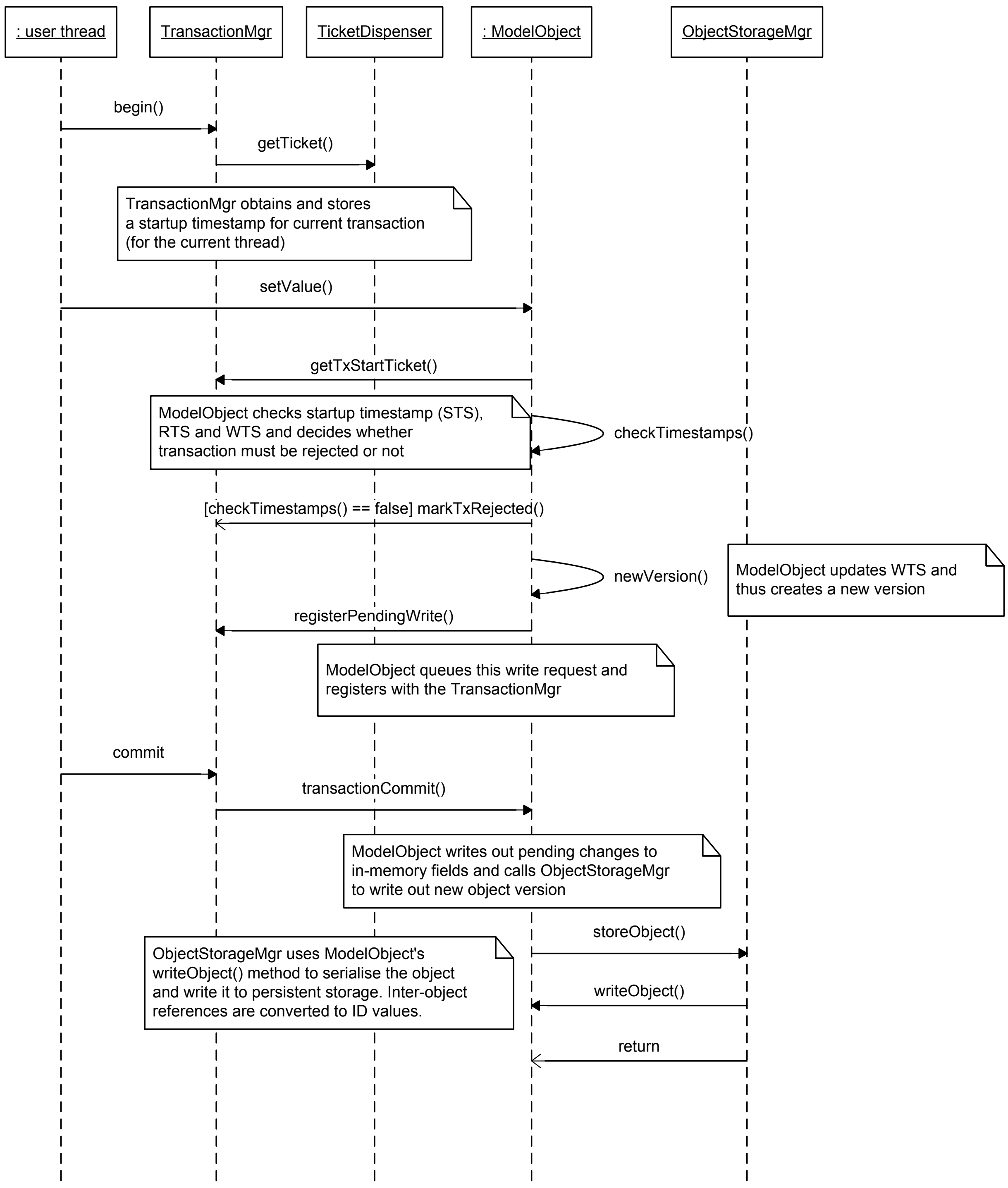

: user thread
TransactionMgr
TicketDispenser
: ModelObject
ObjectStorageMgr
begin()
getTicket()
TransactionMgr obtains and stores a startup timestamp for current transaction (for the current thread)
setValue()
getTxStartTicket()
ModelObject checks startup timestamp (STS), RTS and WTS and decides whether transaction must be rejected or not
checkTimestamps()
[checkTimestamps() == false] markTxRejected()
newVersion()
ModelObject updates WTS and thus creates a new version
registerPendingWrite()
ModelObject queues this write request and registers with the TransactionMgr
commit
transactionCommit()
ModelObject writes out pending changes to in-memory fields and calls ObjectStorageMgr to write out new object version
storeObject()
ObjectStorageMgr uses ModelObject's writeObject() method to serialise the object and write it to persistent storage. Inter-object references are converted to ID values.
writeObject()
return

Figure 5.3: *Sequence diagram of MVTO implementation.*

– and thus this object's version number –, and a read timestamp (RTS) containing the startup timestamp of the last reader of the most recent version of the object.

In each thread, at most one transaction can be active at a time (i.e. nested transactions are not supported). To start, commit, or abort a transaction, the singleton TransactionMgr exposes a set of methods that are called from the user thread. Once a transaction is started, it is assigned a startup timestamp (STS) obtained from the TicketDispenser.

Write requests are only accepted if $STS \geq RTS$ and $STS \geq WTS$ (see above), otherwise they are rejected and the transaction must be restarted. Once accepted, the write is marked as pending and not visible to other transactions until the current transaction is committed.

Objects implement this behaviour by storing a list of pending writes and notifying the TransactionMgr that they have pending writes. Transactions are committed when the user thread calls the corresponding TransactionMgr method, which in turns notifies all objects that have previously announced pending writes to commit these writes.

Once the writes are committed, a snapshot of the current object version is enqueued for serialization to the database by the ObjectStorageMgr. This class acts as an abstraction layer for persistent storage and manages both storage and retrieval with simple cache functionality (see 5.5).

By default, reads are performed as fallback reads: if reading the most recent version would result in delay, the previous version is read. However, clients may opt to use the most recent version in any case.

Handling Cyclic Restarts and Priorities: An Extension to MVTO

One problem of the original MVTO algorithm is that it does not allow any prioritisation of transactions. Transactions may be restarted frequently until successful completion. In extreme cases this can result in a potentially infinite restart cycle between transactions: MVTO's counterpart to 2PL's deadlock problem.

Just as in Carey and Muhanna (1986), the implementation uses a simple adaptive exponential backoff scheme for transaction restarts. The TransactionMgr keeps statistics about transaction lengths and the number of restarts for each transaction and delays transactions progressively with a mean of one average transaction length. Using the waitForRestart() method, a thread can therefore reduce its risk of restarts and increase overall throughput.

For real-time transactions, observing deadlines is a crucial requirement. The concurrency control algorithm does not give any timeliness guarantees: while the adaptive delay scheme just described is helpful for overall performance, it does not assist a thread in ensuring timely completion of its transactions. An extension is to the MVTO algorithm is therefore introduced that allows threads to preempt other threads in certain situations: specifically, when a write transaction is granted but delayed by a pending write from another thread.

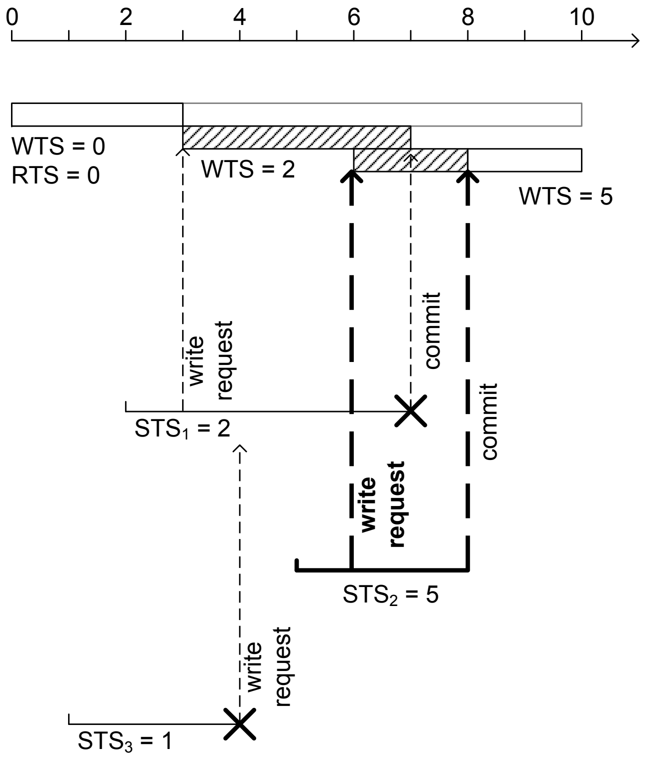


Figure 5.4: *MVTO sequences with preemption.*

Each thread has a priority level that determines whether it may preempt another thread. Preemption is only granted if the conditions for STS, WTS, and RTS are fulfilled (as discussed in 5.2.4) and the transaction with the pending object write, T_1 has a lower priority p than the one requesting preemption, T_2. It thus follows that preemption is only possible iff T_1 currently has a pending transaction on the object and $STS_1 < STS_2 \wedge p_1 < p_2$. The preempting thread then simply overwrites the object's current WTS to claim its precedence and become the new write owner of the object. When the previous write owner attempts to commit or perform another write on the same object, it finds that the object's WTS has changed in the meantime and the operation therefore fails, resulting in a transaction abort.

An execution example is shown in figure 5.4 (based on the previous example of figure 5.2). Transaction T_2 has been assigned a higher priority than T_1 and is therefore able to preempt T_2 and create a new version of the object with $WTS = 5$. Note that T_3 is unable to preempt T_1 regardless of its priority, as the basic condition $STS(T) \geq WTS(O)$ must always hold. The object version created by T_2 is discarded.

Pending writes are kept private and conflict-free by implementing them in thread-local storage instead of attaching them to each object as in the original method. This means that it is possible for two threads to maintain pending writes on the same object concurrently for some time until one of them detects that it has been preempted, which brings the MVTO implementation closer to OCC in some respects.

The following code section shows the awaitPendingWrites() method of the model object class in simplified form:

```
// the actual pending writes flag
transient boolean pendingWritesFlag = false;

// predicate: noPendingWrites (pendingWritesFlag == false)
transient final Condition noPendingWrites
        = lockPendingWritesFlag.newCondition();

// S_TS of the writer who's currently holding pending writes
transient long currentWriter;

// wait until pending writes flag is cleared
void awaitPendingWrites(long S_TS, boolean releaseLock) {
  int myPriority = TransactionMgr.getInstance().getPriority();
  while (this.pendingWritesFlag != false) {
    // preempt if possible
    if(this.currentWriterPriority < myPriority) {
      break;
    }
```

```
        // wait otherwise
        this.noPendingWrites.await();
    }
}
```

A potential downside of preemption is that the preempted thread is not immediately notified of the need to abort, but only when it attempts to access the object or the TransactionMgr the next time. For computing-intensive transactions, this could result in wasted computation effort. We have found that for our purposes, this issue did not have any measurable impact. If necessary, an explicit signal could be sent to preempted threads to ensure rapid aborts.

5.2.5 Performance Evaluation

The primary goal of this performance evaluation is to assess the relative throughput of the MVTO concurrency control implemented compared with a simple exclusive locking approach. Specifically, it is interesting to evaluate how the algorithm performs in extreme cases as well as in situations believed to be representative for the actual operation of a large-scale building model.

Unlike the simulation-based study performed by Carey and Muhanna (1986), actual implementations are run using the object-oriented implementation described above. As persistent storage is an absolute requirement regardless of the concurrency control mechanism used, it is disregarded in this comparison: versions are kept in working memory only. This quickly piles up a large number of briefly-referenced objects that are candidates for garbage collection (GC). To reduce its impact on the measurements, GC is explicitly initiated between test runs using System.gc().

Measurements are conducted for the MVTO and locking implementations with a combination of the following parameters:

- Number of Concurrent Threads: 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000. This parameter sets the number of threads that execute concurrent transactions. All threads execute at the same priority (no preemption takes place).
- Write Probability: 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0. This parameter controls the likelihood that the next transaction executed by a thread will be a write transaction.

- Transaction Delay: 0, 3, 6 milliseconds. This parameter introduces an artificial pause in each transaction before it is committed to simulate “long” transactions.

Each test run comprised 100,000 successfully committed transactions.

Measurements were conducted using Sun’s Server VM 1.5.0_07-b03 for Windows XP (32 bit) on a dual-core Athlon 64 X2 3800+ (2 GHz core clock frequency) CPU with 1 GiB of RAM. To reduce the impact of any background tasks, the process’s base priority was set to HIGH_PRIORITY_CLASS. The Java process start and maximum heap size were set to 512 MiB using the –Xms and –Xmx parameters, respectively. System memory usage was monitored during the test runs to detect possibly skewed results due to excessive paging (thrashing): the process working set never exceeded 20 MiB, with several hundred MiB listed as “free” by the operating system during the entire test run.

Results

Figure 5.5 compares the sustained throughput of short transactions for various write probabilities. Short transactions consist of a read or a read and write operation on an object without any further processing or delay. On all diagrams, the dual-core CPU results in a visible throughput increase between 1 and 2 threads. As is expected, higher thread counts generally result in lower throughputs because thread synchronisation efforts increase while the available CPU resources remain constant.

From 1% to about 20% write probability, MVTO is at advantage: due to the small percentage of writes, restarts are rare and generate less overhead than locking. At 100% writes, the situation is clearly reversed: when conflict is almost guaranteed, locking is more efficient than conflict detection plus restarts. Even though this is the worst-case situation for MVTO, the throughput difference is limited: while locking generates almost twice as many successful transactions than MVTO at 2 threads, the advantage is reduced quickly for higher thread counts, converging to parity at about 1000 threads.

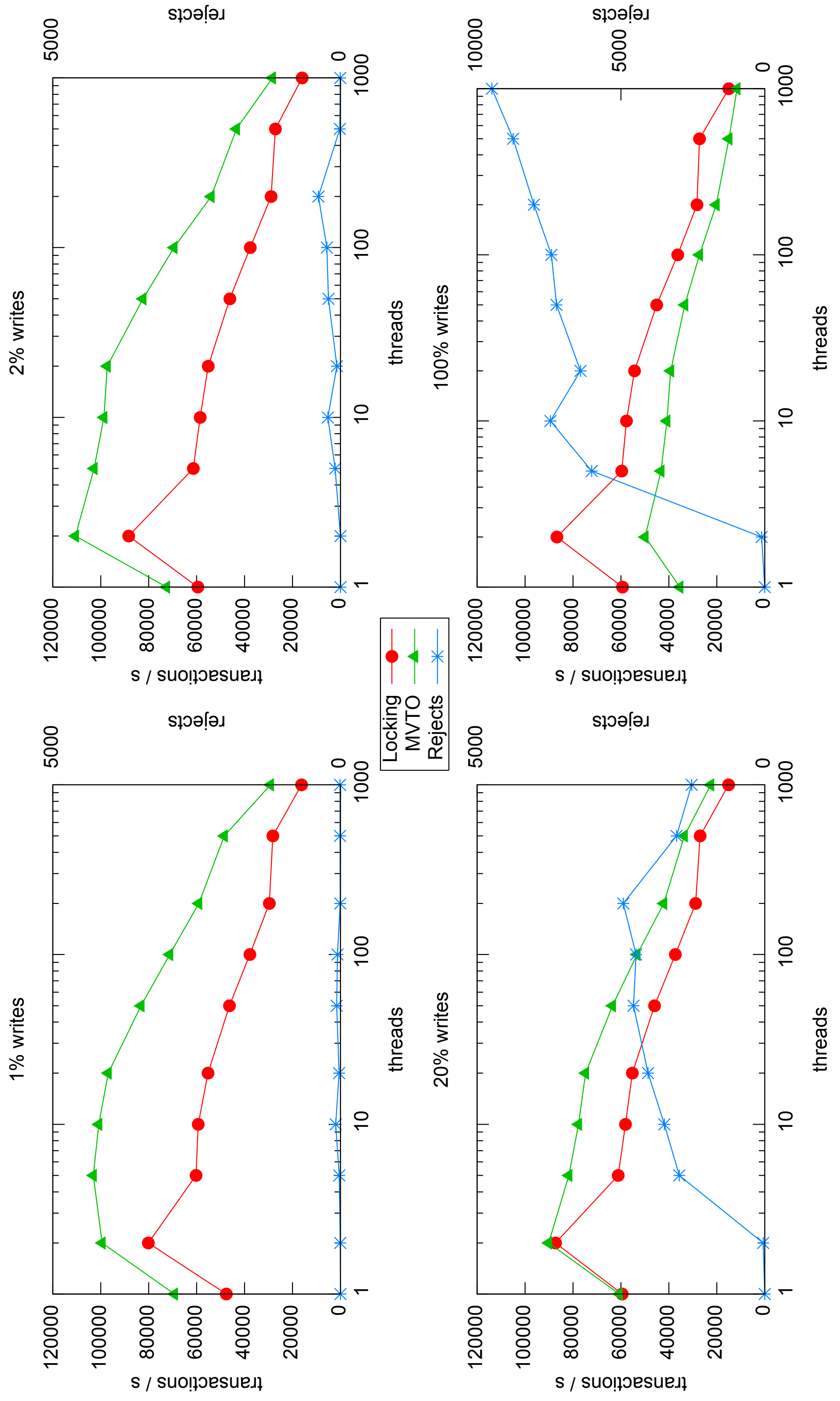


Figure 5.5: *Throughput of short transactions for various writer ratios.*

Figure 5.6 compares the sustained throughput of "long" transactions. Such transactions include a pause of approximately 3 milliseconds before committing. This is to simulate transactions that comprise more than just a series of object reads and writes, but possibly other processing (calculations, I/O). Clearly, this is an idealised setup, as the threads are actually sent to sleep, effectively reducing their CPU usage share to zero. It therefore does not represent a situation where transactions execute CPU-bound tasks in addition to model access.

In this case, the scale effects are dramatic. Maximum locking throughput is always bounded by $1/t_{transaction}$ (about 330 transactions per second) as the threads queue up for lock acquisition, taking turns to use the object. At 1% write probability, MVTO throughput scales linearly at throughputs approaching about 85% of the upper bound $n/t_{transaction}$ due to the low risk of restarts, until about 100 threads[2].

After this point the cost of concurrency control begins to outweigh the benefit, yet throughput remains well above locking until at least 1000 threads. The advantage diminishes with higher write probabilities up to 100%, where the two concurrency control methods tie at all thread counts.

[2]Note logarithmic scale on diagrams. A good linear approximation for throughput for thread numbers $1 \leq n \leq 100$ is $287.36n + 550,31$ ($R^2 = 0.994$)

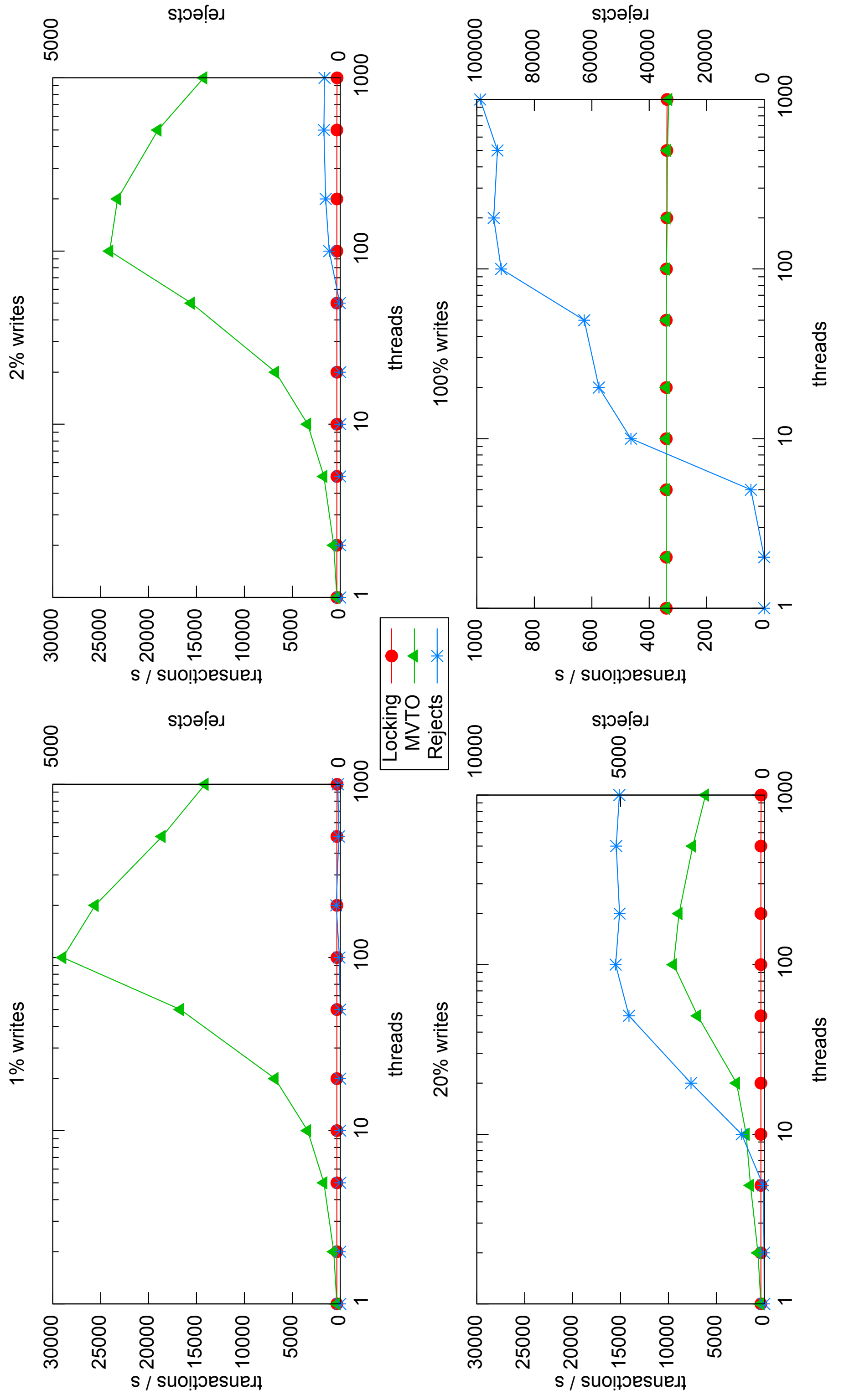


Figure 5.6: *Throughput of “long” transactions of approx. 3 milliseconds for various writer ratios.*

Figure 5.7 compares the sustained throughput of “long” transactions, this time with $t_{transaction} \approx 6$ ms. The results are principally the same as those for three-millisecond transactions.

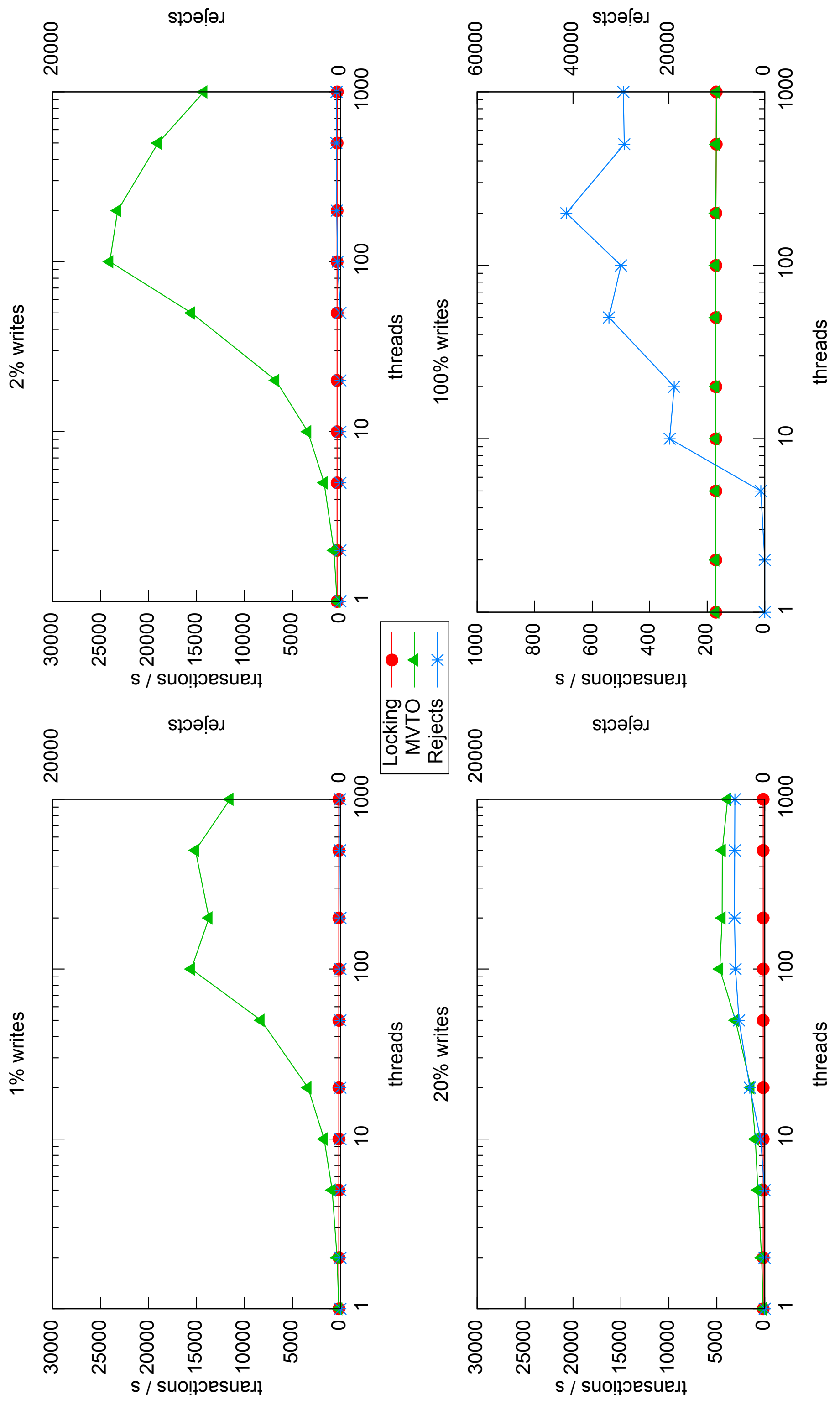


Figure 5.7: *Throughput of "long" transactions of approx. 6 milliseconds for various writer ratios.*

5.3 Temporal Ordering of Data

The building model reflects objects and events (facts) in the physical building, keeping histories of object states over time. This section discusses some challenges that arise in tracking the temporal ordering of real-world events with a temporal database and describes an approach to deal with these issues.

5.3.1 Terminology

In the terminology collected by Dyreson et al. (1994), there are two main notions of time:

- *Valid time* relates to the time when facts are true in the real world. Time here is generally understood as "wall clock time", which is observable in limited accuracy and granularity. A single point in valid time may be represented as "2006-10-15 18:15:32.198", for instance. In this example, the given granularity is in milliseconds. Valid time is primarily of importance to the external user of the database: it orders events in the real world.

- *Transaction time* relates to the order in which facts become available in the database. It is supplied by the database itself and has the important property of "marching monotonically forward" (Jensen and Snodgrass 1999) when triggered by certain events. A single point in transaction time may be represented by an integer number, such as 7637463. Transaction time is primarily of importance to the internal mechanisms of the database: it orders database activity.

The building model database is a bitemporal database as defined by Salzberg and Tsotras (1999) because it records both kinds of time. As a transaction-time database, it is able to roll back to previous states and provide consistent snapshots of the past. As a valid-time database, it models the current state of the real world – at least those facts that are known about it.

It follows that each database record must comprise at least a 3-tuple $(t_v, t_t, data)$, where t_v denotes valid time and t_t denotes transaction time. Each t_t is unique for each object (actually, for the entire database) and can be considered the object's *version*. Valid-times are not necessarily unique, i.e. multiple versions of the same object (or other objects) may carry the same t_v.

5.3.2 Time Travel: Reconstructing Model Snapshots

Considering only transaction time, retrieval of consistent historic object network snapshots (sometimes called “time travel”) is simple. Based on a given reference transaction time value t_r, obtaining a consistent snapshot for the reference time amounts to obtaining, for each object, its version with the highest available transaction timestamp $t \leq t_r$, as illustrated in fig. 5.8. In the figure, the current transaction time is ≥ 10 and the reference transaction time is 8. Obtaining a snapshot for the latter corresponds to the most recent versions of objects 1 and 2 and non-recent versions of the other objects.

Retrieval of non-recent object versions is integral functionality of the model core and transparent to the client, regardless of where the objects are stored (see 5.5 for details).

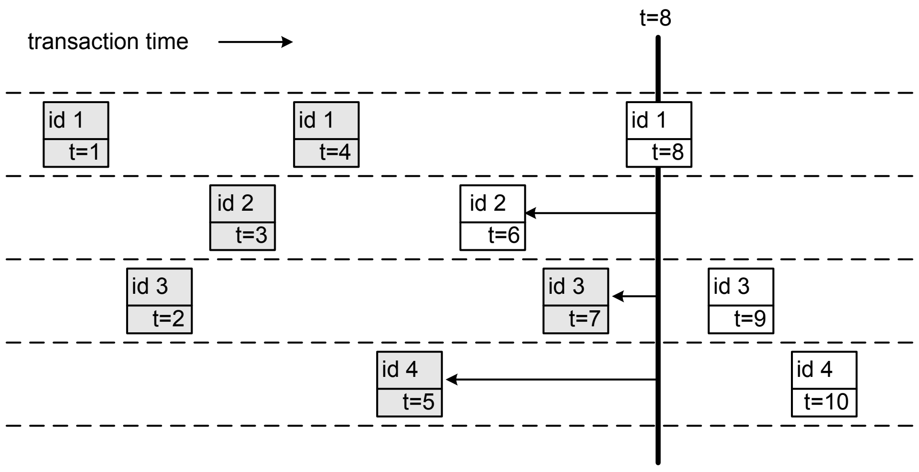


Figure 5.8: *Reconstructing object set snapshots for points in transaction-time.*

However, ignoring every notion of time other than transaction time is an idealised assumption.

5.3.3 Delay

Facts about the real world are fed into the database from external sources, such as sensors, but possibly also by manual data entry. Data input may occur either online or in batch processing. In the first case, attributes of a model object (e.g. the temperature reading of an indoor climate sensor) are updated in an automated fashion, as soon as possible after they have been recorded by the sensor. For instance, a sensor may transmit its current reading in 1-second intervals to the building model, using some form of communications channel. In the second case, data are collected independently of the building model (e.g. by an autonomous data logger device) and fed into the database as a “batch” of data some time later, e.g. in a nightly or weekly rhythm. This may happen semi-manually, typically by an operator who downloads collected sensor readings from the devices and then feeds them into the building model.

In both cases there will be a delay between the time the real-world fact is observed, and the time that this fact is recorded in the database. For online data input, this delay can be expected to be somewhere in the range of fractions of a second to a few seconds. For batch processing, delays could range from hours to weeks.

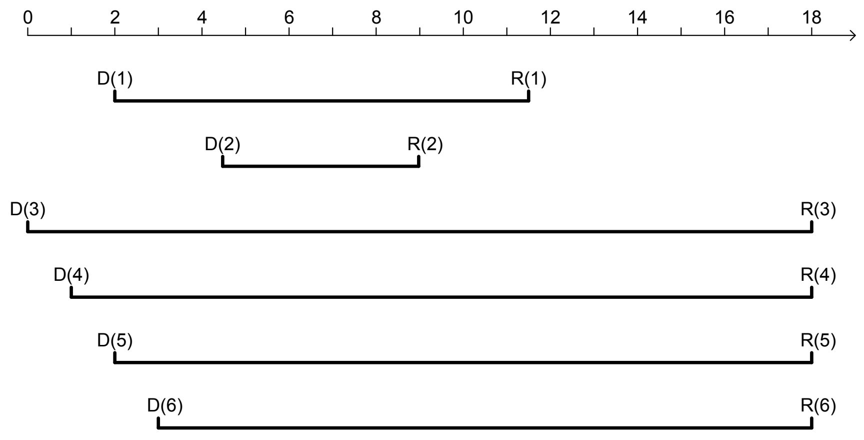


Figure 5.9: *Delays between observation (O) and recording (R) of facts. The time scale represents "wall clock time", measured in fractions or multiples of a second.*

Figure 5.9 illustrates some delay and ordering timelines beginning with observation (O) of a fact and ending with its recording (R) in the database. Facts may be related to the same model object or different model objects. The first two timelines from the top show a situation where the occurrence of fact 1 happens prior to that of fact 2, yet is recorded later than fact 2. This may be due to communication channels with different speeds, necessary pre-processing of the sensor data before it is recordable, or simply due to uncontrollable task scheduling decisions by the model server’s operating system. The remaining timelines illustrate a batch input situation: a number of facts occur one after the other, but are recorded as a “batch” significantly later at (or around) the same point in time.

Delay alone is not a significant problem for the designer or user of a database, at least as long as only a single object is considered and ordering is preserved. When updates are delayed but arrive in the same order as the real-world events they represent, an important assumption holds: namely, that valid time increases monotonically over transaction time. This assumption has two consequences for queries. First, the latest object version (with the highest transaction timestamp) is guaranteed to be the latest version in terms of valid time, i.e. the most current representation of the physical object available. This means that obtaining the most current version is trivial: one simply has to select the version with the highest transaction-timestamp. Second, the sequence of transaction times of all object versions corresponds to the sequence of valid times. This means that getting a faithful representation of the valid-time evolution of an object is equivalent to a linear traversal of its history of transaction-time.

Somewhat more formally, if an object o is defined as $(t_v, t_t, data)$ like shown above, for any two versions of the object, $t_{v1} \geq t_{v2}$ if $t_{t1} > t_{t2}$.

5.3.4 Ordering

As discussed above, an important assumption underlying many temporal databases is that the startup times of write transactions correspond to the temporal ordering of the events that caused them, or are at least correlated to some extent. In the case of a building model, this means that streams of sensor data must be received and processed in the order that they were generated.

Ensuring this condition is fairly simple for a single data source, even when the transmission channel does not guarantee arrival in the same order that data were sent. Assuming that all messages are sent with a sequence number to establish a total ordering, messages can be buffered and sorted before they are passed on for further processing following the *Resequencer* pattern (Hohpe and Woolf 2003, ch. 7).

However, ensuring ordered arrival over multiple data streams coming from different sources with different transmission delays will quickly become infeasible. First, sequence numbers or timestamps would have to be coordinated among the data sources so that an unambiguous total ordering could be determined for all data – increasing the required capabilities and therefore cost of data sources. Second, the necessary buffer size and computational effort for sorting increases with the number of data sources. Third, the expected delays incurred by waiting for out-of-order arrivals would increase, as each data stream increases the chance that the Resequencer has to wait for “straggler” messages. It follows that a total order of all incoming messages cannot be established with reasonable effort.

Even if such a total order of incoming messages exists, there is no guarantee that the messages result in an equivalent ordering of database updates. As discussed earlier, the transaction control mechanism does not guarantee strict serializability: restarts of concurrent write transactions result in reordering of updates.

For queries, the result is that the “latest” object version (with the highest t_t of all versions) may not be the version with the latest valid-time, which may prove inefficient for queries if only the transaction-time is indexed. This problem can be alleviated by keeping a separate index for valid-time. Within an object’s history, the ordering of t_t does not necessarily correspond to the ordering of t_v (Figure 5.10). This introduces some ambiguity as the ordering of object versions with the same t_v cannot be resolved by comparing t_t.

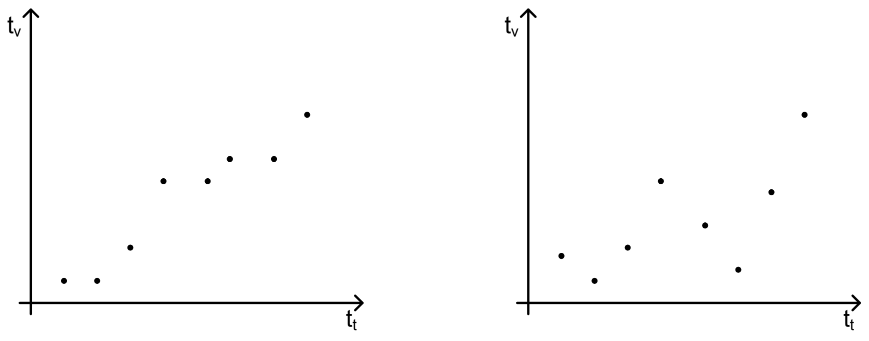


Figure 5.10: *Ordered vs. un-ordered arrival of updates and its effect on the monotonicity of t_v over t_t.*

The situation is complicated further for multiple objects. To obtain a consistent snapshot of multiple objects, one can generally use a reference transaction-time t_{ref} as a basis and for each object, obtain the version with the highest available $t_t \leq t_{ref}$. While this results in a snapshot that is consistent with regard to transaction-time, it can result in a snapshot that is not consistent in terms of valid-time. Consider figure 5.11, which shows two objects with multiple versions and their respective t_v and t_t values. Until $t_t = 5$, things appear normal, and snapshots are valid for both transaction-time as well as valid-time. At $t_t = 6$, the symmetry breaks: the inserted object version’s valid-time is greater than the valid-time of its last version, as before. However, it is less than the valid-time of the other object's latest version. Similarly, in $t_t = 7$, an out-of-sequence update for O_1 appears.

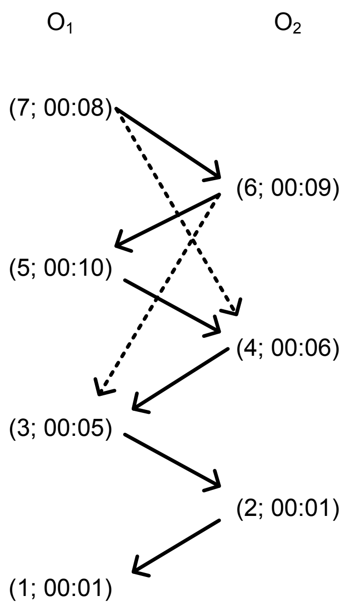


Figure 5.11: *Two objects (t_t, t_v) with delayed and out-of-sequence updates*

5.3.5 Requirements Summary and Solution Approach

The database is connected to a number of sources S_i. Each source i emits a sequence of messages $m = \{data, s, t_v\}$ where $s \in \mathbb{Z}$ is a sequence number and t_v is a timestamp. The sequence number is local to the source. Using the relation $<_s$, the messages populate a total order $(M_i, <_s)$ for each source.

The timestamp represents a global "wall clock" time of limited granularity, which means it is actually a time interval. Multiple messages can therefore carry the same timestamp, resulting in a total order $(M, \leq_{t_v})$ for all messages from all sources.

One or more incoming messages eventually result in a database update. The ordering of startup transaction times t_t of successfully committed update transactions can be understood as a *known-before* relation, while the ordering of timestamps t_v is a *happened-before* relation. It is desirable that *happened-before* relations correspond to *known-before* relations:

Condition 5.3.1 *For any pair of recorded messages* (m_1, m_2)*,*

$$t_{v1} \geq t_{v2} \Longleftrightarrow t_{t1} > t_{t2}.$$

However, the discussion above has shown that this condition does not hold if t_t can only increase monotonically with each recorded message, as messages are not guaranteed to arrive in order of increasing t_v, and recorded messages must not be changed. The proposed solution is to allow insertion – or "sandwiching" – of transaction times between existing transaction times.

When an incoming message is determined to be a late arrival, i.e. its t_v is less than the t_v of the most recent record, it is not assigned the next transaction-time increment. Instead, the correct location between two existing $t_{t,lower}$ and $t_{t,higher}$ must be determined, and the message must be recorded with a new $t_{t,middle}$.

The correct insertion transaction-time that fulfils condition 5.3.1 can be determined from the message's t_v easily. Due to the limited granularity of t_v, multiple valid insertion points may be available (see figure 5.12). The sequence number s can help narrowing the set of choices, but is not guaranteed to do so, as sequence numbers are local to a single source.

This retroactive insertion approach potentially changes the meaning of t_t for establishing the known-before relation. The implementation should ensure that the actual order of message arrival can still be reconstructed. Moreover, it should not be necessary to change records that have already been written to persistent storage.

5.3.6 Implementation Options

The main problem for implementing retroactive insertion of records is finding a solution for efficient allocation of and access to inserted transaction-times. Some apparent solution approaches include:

- Increasing the default increment for transaction-time, leaving gaps between values for later insertion. For instance, transaction-time could be set to increase by 100 instead of 1, leaving 99 intermediate transaction times for later use. Even in the optimal case, this limits the number of possible retroactive inserts between two regular transaction-times to "increment minus 1", but potentially wastes lots of available transaction-times when few or no late insertions are needed.

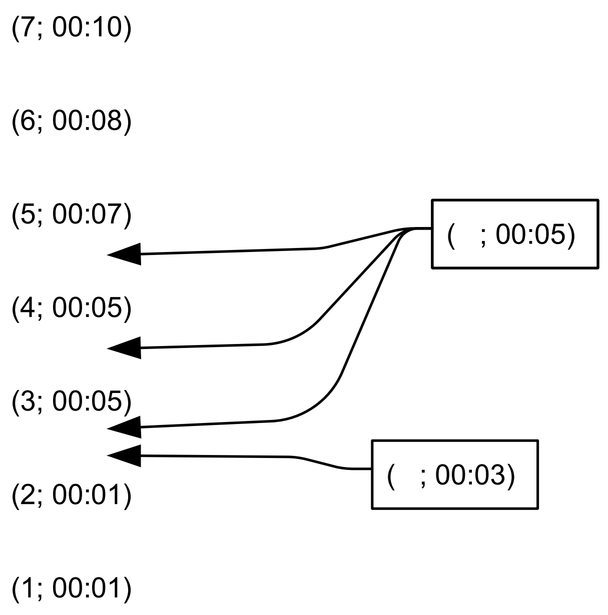


Figure 5.12: *Finding the correct point for late insertion of messages: single and multiple valid insertion points.*

- Using a “virtual tree” approach, as in the classic paper by Dietz and Sleator (1987) and a recent adaptation by Bender et al. (2002), called the “amortized order-maintenance algorithm”. These algorithms ensure efficient allocation of available transaction-times and do not need an explicitly represented tree structure or any other data besides the list, which makes them very space-efficient. However, they do require re-labelling of existing records.

- Using a separate ordering table or tree-based index. Instead of comparing the transaction-times arithmetically as in the original algorithm, an ordering lookup table or b-tree is maintained. In this case, the transaction-times can even be arbitrary (but non-repeating) values. This requires additional storage and maintenance for each inserted record and complicates SQL queries significantly.

In choosing a suitable solution, it is assumed that retroactive insertion is not a very frequent operation, i.e. most updates occur in sequence (by appending). The solution must also take into account that the actual storage and retrieval is handled by an RDBMS through SQL queries.

5.3.7 Implementation

The *valid-time* and *transaction-time* remain as before. Two additional fields are introduced:

- The *insertion-time* t_i is an extension of the *transaction-time*. It is implemented as a text field of varying length (“character varying” or “text” in SQL). The lower bound of the interval $t_{t,l}$ (the lower transaction-time) is always assigned the lowest possible value of t_i. Any record that is inserted between two existing transaction-times $t_{t,lower}$ and $t_{t,higher}$ is assigned the lower bound’s transaction-time. The insertion-time is then determined from the position of the new record (see figure 5.13).

- The *recording-valid-time* is similar to the valid-timestamp in that it is a “wall clock” value. However, it tracks the actual time when a record is written to the database and therefore represents the *known-before* relation.[3]

[3]Alternatively or additionally, a monotonically increasing transaction-time for insertions could be used if the exact order of writes – as opposed to their valid-times – must be available.

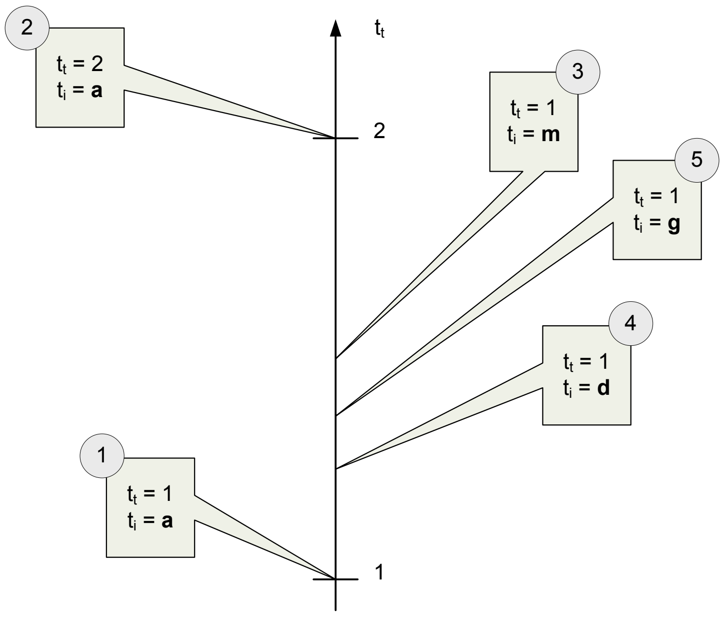


Figure 5.13: *Sample insertion sequence.*

Choosing Insertion-times

Figure 5.13 shows some insertions using the basic latin alphabetical range (26 characters) for insertion-time. The algorithm basically treats t_i as if it represented the fractional part of transaction-time. To select an appropriate value of t_i between two existing records, the middle point between their insertion-times is chosen. When the value range has been exhausted, i.e. when a new record is to be inserted between two records with successive values of t_i, the "precision" is increased by appending a new digit. As an example, to insert a record between two records with equal t_t and $t_{i,1} = b$ and $t_{i,2} = c$, the insertion-time $t_i = bm$ is chosen. This is analogous to inserting the decimal number 1.25 between the numbers 1.2 and 1.3.

Value Ranges and Storage for Insertion-time

In the optimal case, the value range of each digit is used up completely. Given a value range of r usable characters per digit, up to $r^n - 1$ values can be inserted between two transaction-times when using a string of length n for the insertion-time in this optimal case.

In the worst case, subsequent insertions always occur before or after the last inserted value. The maximum number of insertions then derives from the number of times the value range can be divided by 2 before the next digit must be used, i.e. $(\lfloor \log_2 r \rfloor)^n$.

Assuming a very conservative $r = 26$ (characters from a to z), this results in a range of $[4^n \mathinner{..} 26^n - 1]$. For $n = 16$, at least $4^{16} \equiv 2^{32} \approx 4.3\text{E}9$ insertions can be stored between two adjacent transaction-times. This conservative estimate is expected to work with even the simplest SQL database.

Recent versions of the PostgreSQL DBMS provide the data type *bytea* for variable-size binary strings, which allows to use all 8 bits per octet ($r = 256$) with full support for sorting (Pos 2006, ch. 8). Hence, the range becomes $[8^n \mathinner{..} 256^n]$. For $n = 16$, at least $8^{16} \equiv 2^{48} \approx 281\text{E}12$ insertions are then possible.[4]

Discussion

The proposed approach is efficient and simple to implement. In the absence of insertions, the only storage overhead is an additional two fields for each record. In-memory concurrency control is not affected at all, as it is only relevant for current object versions. Database concurrency control is delegated to the DBMS, using its provided transactionality mechanisms. As no additional tables or indexes are required (although they may be used for query optimisation), the known-before and happened-before relations can be deduced from the records themselves.

The advantage of using a string data type is that it is of variable length and allows the RDBMS to use its built-in lexical sorting algorithms, allowing simple SQL queries for object retrieval. In particular, the frequently-needed query for an object's latest version less than or equal to some reference value (see 5.3.2) remains trivial. No renumbering or additional lookups in separate tables are needed.

A possible issue with retroactive insertion arises when older versions of objects are successively stored to write-only media (WORM) for long-term archival. In this case, query performance will degrade as object histories may be scattered among multiple non-contiguous volumes, especially when insertions occur very late. Unless an index is available, sequence reconstruction requires a full table scan to find late insertions, possibly across many volumes.

[4]PostgreSQL version 8.2 requires a fixed overhead of 4 bytes for storing a binary string, plus the actual length of the string in octets (Pos 2006, 8.4). Insertion-times therefore add at least 5 bytes to each record and up to 20 bytes if $n \leq 16$.

5.4 Application Interface

The most current version of each object is always kept in memory; the singleton graph of all interrelated current object versions constitutes “the model”. Only this most recent, online version of an object will receive updates from building systems and allow to send commands back. Anyone holding a reference to such an object must be aware that this is a public, shared object and may change at any time.

Applications wishing to obtain private object copies of the most recent object version or of previous versions can obtain offline objects. An offline object is completely decoupled from the model and represents a snapshot object state. It can be changed, but this has no effect whatsoever on the model; it does not react on any incoming building system data (compare 3.2.1).

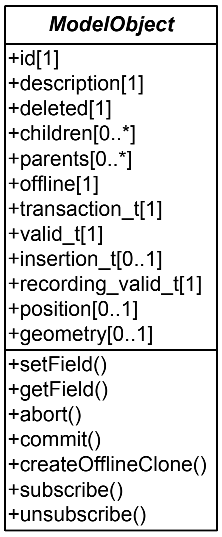


Figure 5.14: *The object model root class (simplified).*

To support this functionality, all model objects are derived from a root class, ModelObject, as shown in figure 5.14. Historic object versions are provided by the Object-StorageMgr component (see 5.5).

5.4.1 Object Creation and Destruction

The model core provides factory methods for creating new online or offline objects; direct instantiation by agents is not supported. Deletion of an object is handled by setting a “deleted” flag on the object that creates a final non-readable “dead” version of the object and blocks any further updates – this approach ensures referential integrity.

5.4.2 Notifications

Agents must be able to register for notification of changes on specific objects (see table 3.7). These notifications are not delivered by synchronous method calls (as in the Java standard library’s java.util.Observer), but through message queues. This loosens not only the temporal coupling between sender and receiver, but also prevents problems that arise when the receiver agent disappears. An additional benefit is that it allows receivers to share the workload of notification processing.

After registration, notifications are sent by the model core on any object change or – using a simple string matching filter – on changes of certain fields only. Notifications are only sent when a write transaction has been committed successfully. A sample sequence is shown in figure 5.15. Care is taken to keep only queue IDs, not object references, for subscriptions. This is to reduce referential coupling between agent threads and the model (see 4.6.1).

5.5 Persistent Storage

Persistent storage is managed by an ObjectStorageMgr component. This component is responsible for storage of and retrieval from a relational database. It offers a quasi-asynchronous interface for object storage to allow low-latency updates and a synchronous interface for object retrieval. The component design is shown in figure 5.16.

Due to the characteristics of the MVTO concurrency control approach, read accesses to recently-written objects occur frequently when the current recent object version is often in a pending-writes state. In such situations, readers are compelled to fall back to non-recent versions. However, these versions may still be in the pipeline to persistent storage. It is therefore helpful for performance to provide access to not-yet-written objects. This would be difficult to achieve using a standard FIFO message queue. Instead, the ObjectStorageMgr maintains an internal queue that is accessible in FIFO as well as random access. Put requests copy the passed object reference to the write queue

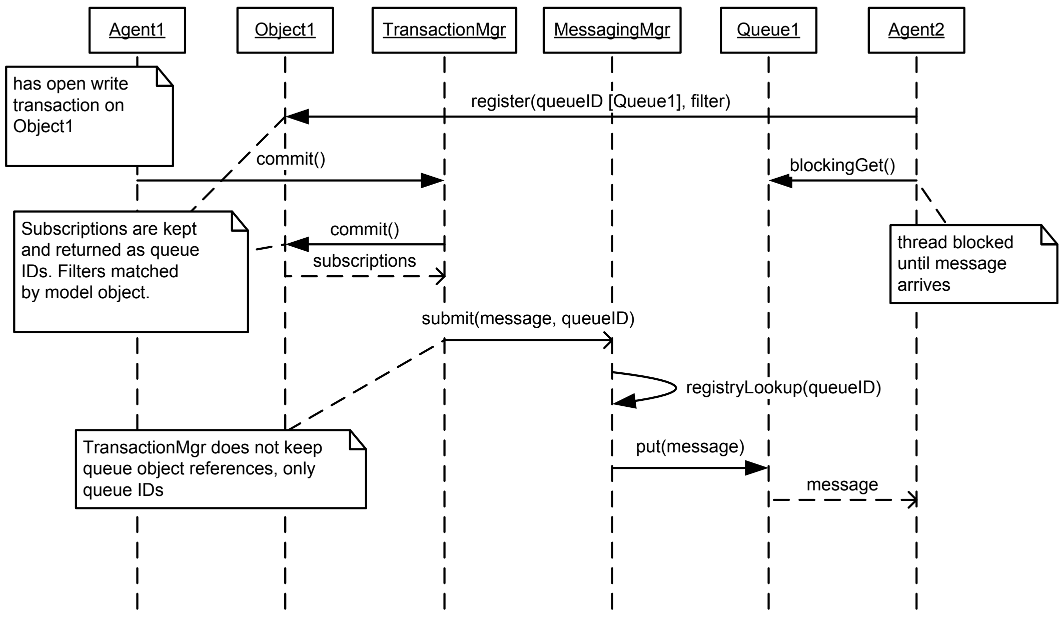

Agent1
Object1
TransactionMgr
MessagingMgr
Queue1
Agent2
has open write transaction on Object1
register(queueID [Queue1], filter)
commit()
blockingGet()
Subscriptions are kept and returned as queue IDs. Filters matched by model object.
commit()
subscriptions
thread blocked until message arrives
submit(message, queueID)
registryLookup(queueID)
TransactionMgr does not keep queue object references, only queue IDs
put(message)
message

Figure 5.15: *Object update notification: a sample sequence.*

and return immediately. The actual writing of the objects is performed by a fixed-size, leader-followers thread pool (see also 4.4).

An additional fixed-size object cache with LRU replacement policy is used to improve the performance of old object version read accesses. Get requests are served from either the write queue or the read cache, which triggers database accesses as necessary.

By design, the ObjectStorageMgr provides a sequential stream of model updates. This could be used as an extension point to update remote slave replicas of the model for fast fail-over in order to increase its availability (cf. 4.2.3).

5.5.1 Serialization and Deserialization: Managing Object References

As outlined before (4.6.1), Java serialization has deep-copy semantics. For any model object, this means that serializing the object will result in serializing the entire object graph through following its parent and child references; such external references to other model objects must therefore be broken before serialization and stored in a way that allows the reconstruction of references upon later retrieval. This is achieved by marking external references as transient and converting them to arrays of object IDs.

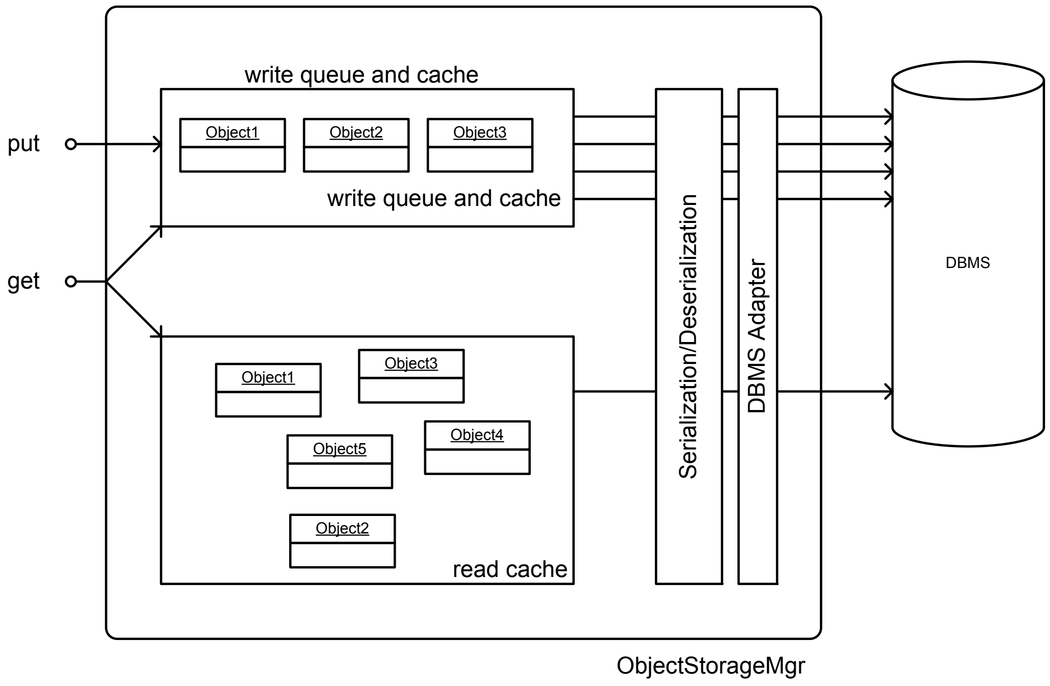


Figure 5.16: *The ObjectStorageMgr component*

References to internal objects (those that constitute the object's data and are only accessible by itself) are kept as is and serialized accordingly.

Upon retrieval, external object references are not restored automatically, but can be restored on request. The policy for object graph reconstruction from persistent storage is generally based on a target timestamp value WTS_t: to retrieve one or more objects from persistent storage, the versions with the largest timestamp less than or equal WTS_t are restored.

5.5.2 Storage Format

The used database schema is intentionally simple: a single table with columns containing the unique object ID, the various timestamps, and a BLOB (binary large object) field containing the serialized object data.

The main advantage of this approach – as opposed to a complete field-by-field object-relational mapping – is a simplified data access layer and vastly improved flexibility, as the database schema does not have to be changed in accordance with object model evolution. The disadvantage is that this makes certain types of queries on the model history very inefficient (e.g. "find all times in the past 7 days when the reading of temperature sensor A was below 15 degrees"). In such cases, many versions of the object must be deserialized to memory and checked for the search criteria. For the scope of this work, it is expected that such queries are relatively rare and generally not time-critical.

As discussed earlier, a notable characteristic of the multiversion mechanism is that database records, once written, are never changed. This has potential benefits for performance, security, and auditing: it simplifies optimisation (e.g. caching, indexing) and allows older data to be archived on write-only mass storage media.

In the current implementation, no additional tables are used – as a result, only queries based on object ID and write timestamp can be supported with database indexes for fast queries without full table scans. In future scenarios, additional tables for maintaining various indexes (e.g. R-trees for spatial queries) should be considered for faster queries.

5.5.3 Performance

To give an indication of the ObjectStorageMgr's sustained write throughput, figure 5.17 shows the results of a performance test for thread counts from 1 to 100. For each thread count, 100,000 objects were initialised with random data. These objects, each taking

657 bytes storage size in serialized form, were then sent to persistent storage using the PostgreSQL 8.2.3 DBMS running on the same computer. Each writer thread allocated a separate JDBC connection using the type 4 JDBC3 driver over a local TCP connection and utilised a prepared statement for all its INSERTs. The DBMS was running in its default configuration except for a raised client limit; two different settings of the *forced synchronization* flag (Pos 2006, 17.5) were tested. The hardware and software setup was the same as described in 5.2.5. Measured times include Java object serialization and complete execution of the database INSERT transaction.

Notably, throughput scales well with thread count until about 40–60 threads (peaking at about 2900 written objects per second), even though all writes affect the same table in the same database.

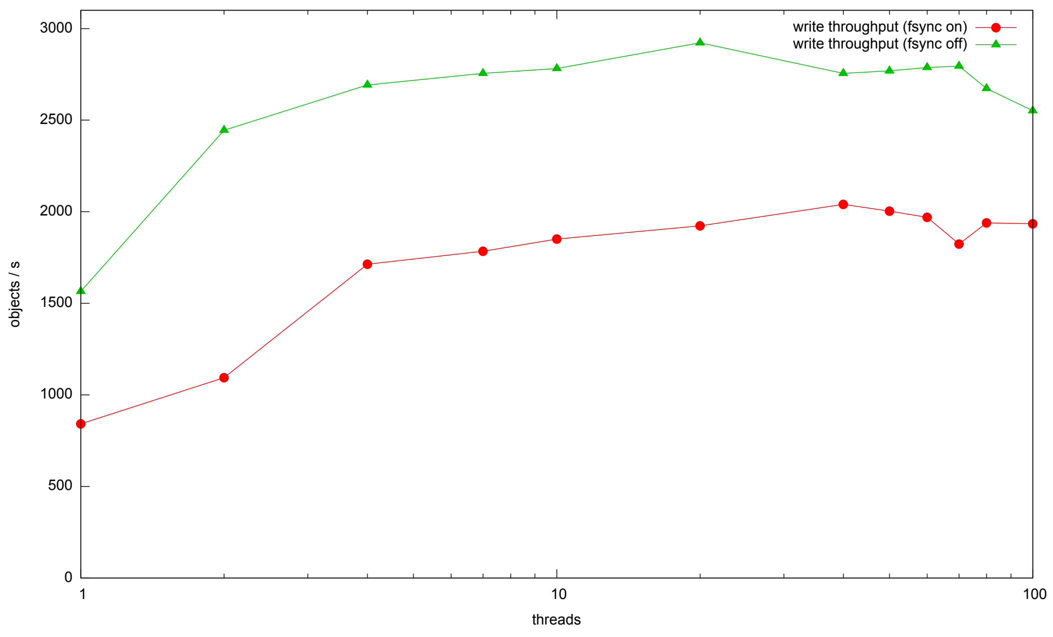


Figure 5.17: *Object storage performance for various thread counts.*

5.6 Related Work: OODBMS

The model server core's feature set is very similar to that of object-oriented database management systems (Atkinson et al. 1989). However, its design differs significantly from that of most available OODBMS systems (Greene 2006, Objectivity Inc. 2006) in some respects:

- In contrast to usual OODBMS systems, it is not based on a client-server or peer-to-peer architecture. This is a deliberate decision driven by its requirements. However, it can be extended easily to provide both generalised as well as application-specific client/server functionality through agents.
- The model server acts not just as a storage component, but as an agent-based execution environment. Unlike classic Active Databases (Paton and Díaz 1999), user-supplied code is not restricted to an event-reactive pattern.
- The model server has been designed for object-level versioning from the ground up, including built-in support for happened-before and known-before relations, and retroactive insertion.
- While many OODBMS designs rely on locking for concurrency control, it uses multi-version concurrency control and has some support for transaction priorities.
- It delegates the actual persistent storage and retrieval to a separate DBMS.

5.7 Chapter Summary

In this chapter, the design of the model server core was described. Key points in summary:

- The building model is accessible as a graph of plain objects.
- All objects are versioned; the most recent version of each object is available in main memory, older versions are kept in persistent storage provided by a DBMS. Access to either is transparent for clients.
- Transactionality is provided by using a concurrency control scheme based on multi-version transaction ordering. An extension to the original MVTO algorithm is introduced to support different priorities for write access through preemption.
- The MVTO algorithm performs very well compared to a simple locking scheme. In typical usage scenarios, its performance and scalability are significantly better.
- The model server provides full support for happened-before and known-before ordering of data even when the data arrive unordered or very late. This is achieved by allowing insertion of old object versions, while maintaining simple querying and the important property that written objects are never changed.
- Using an open-source RBDMS without specific optimisations for persistent storage, sustained write rates of several thousand objects (versions) per second are achievable on standard desktop-PC-class hardware.

6 System Application: A Simulation-Based Lighting Control System

The building model service implementation was driven by the needs of a research project that aimed to produce a working prototype of simulation-based control based on a sensor-supported building model. This chapter describes the experimental setup of the project and the context in which the model service prototype was initially designed and operated.

The full system setup was successfully used to evaluate the functional integration of all system components.

6.1 System Overview

The system architecture was set up to support a simulation-based lighting control system for an experimental office space containing remotely controllable light fixtures and window blinds (Figure 6.4). The model service was connected to the following building systems, as shown in figure 6.1:

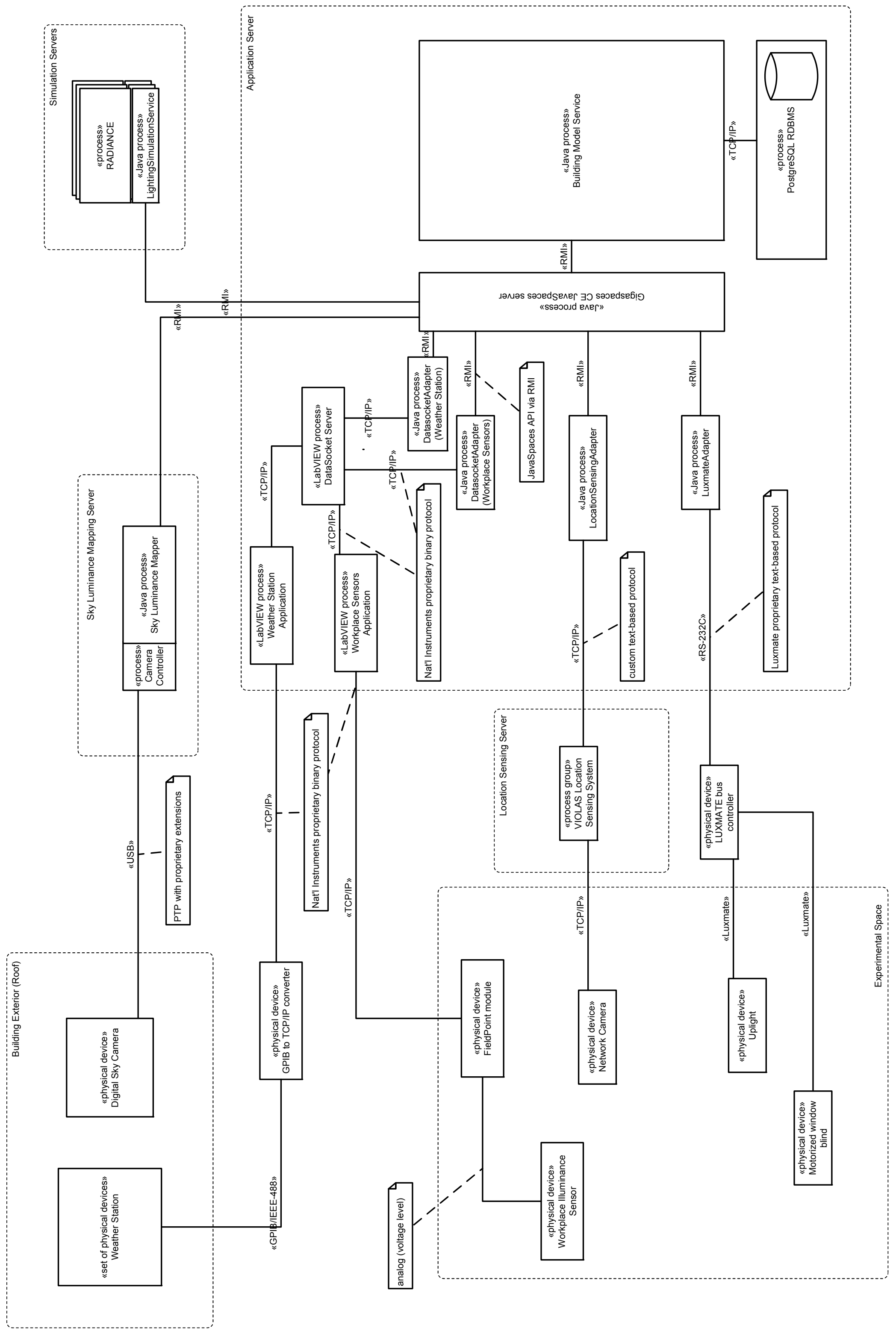

Simulation Servers
«process» RADIANCE
«Java process» LightingSimulationService
Application Server
«Java process» Building Model Service
«TCP/IP»
«process» PostgreSQL RDBMS
«RMI»
«Java process» Gigaspaces CE JavaSpaces server
«RMI»
«RMI»
«RMI»
«Java process» DatasocketAdapter (Weather Station)
«RMI»
JavaSpaces API via RMI
«RMI»
«RMI»
«Java process» DatasocketAdapter (Workplace Sensors)
«TCP/IP»
«LabVIEW process» DataSocket Server
«TCP/IP»
«TCP/IP»
«TCP/IP»
Nat'l Instruments proprietary binary protocol
«Java process» LocationSensingAdapter
«Java process» LuxmateAdapter
Sky Luminance Mapping Server
«Java process» Sky Luminance Mapper
«process» Camera Controller
«LabVIEW process» Weather Station Application
«LabVIEW process» Workplace Sensors Application
custom text-based protocol
«TCP/IP»
Luxmate proprietary text-based protocol
«RS-232C»
Location Sensing Server
«process group» VIOLAS Location Sensing System
«physical device» LUXMATE bus controller
Nat'l Instruments proprietary binary protocol
«TCP/IP»
«USB»
PTP with proprietary extensions
«TCP/IP»
«TCP/IP»
«Luxmate»
«Luxmate»
Experimental Space
Building Exterior (Roof)
«physical device» Digital Sky Camera
«set of physical devices» Weather Station
«physical device» GPIB to TCP/IP converter
«GPIB/IEEE-488»
«physical device» FieldPoint module
«physical device» Network Camera
«physical device» Uplight
«physical device» Motorized window blind
«physical device» Workplace Illuminance Sensor
analog (voltage level)

Figure 6.1: *An overview of system components and connections in the experimental setup.*

- A weather station (Figure 6.2) measuring outside air temperature, relative humidity, air pressure, illuminance, global irradiance, precipitation, and wind speeds. Measurements were taken in one-second intervals by a LabVIEW application, converted, and sent as messages by a Java adapter program.
- A sky luminance scanning system based on analysis of sky images (Spasojević and Mahdavi 2005) periodically taken by a digital camera (Figure 6.3). It supplied 1–2 arrays of sky luminance data (each containing 256 floating-point numbers corresponding to sky patches) per minute through asynchronous messaging.
- A location sensing system providing building geometry information through the use of optical markers ("tags") and digital cameras (Icoglu and Mahdavi 2005, Icoglu 2006). The system was connected using a custom text-based TCP protocol and sent bulk data updates in intervals of a few minutes, which were picked up by a Java adapter and submitted as messages to the model service.
- Indoor illuminance sensors used for validation of the simulation-based control algorithm, also connected using LabVIEW and an adapter program.
- A LUXMATE bus system controlling two dimmable uplights and motorised window blinds. The LUXMATE system was connected through a serial interface to a Java adapter program (see 8.1 for details).

6.2 Application: Control Cycle

The control methodology is described in detail in other publications (Mahdavi and Spasojević 2006, Mahdavi et al. 2005); the following description focuses on the implementation of the workflow as shown in figure 6.5.

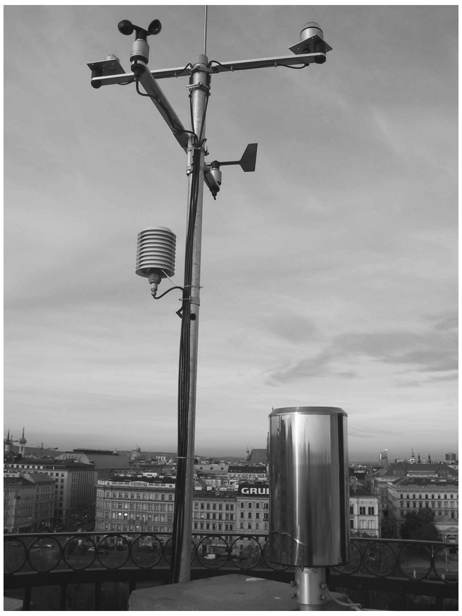


Figure 6.2: *Weather station mounted on main building roof.*

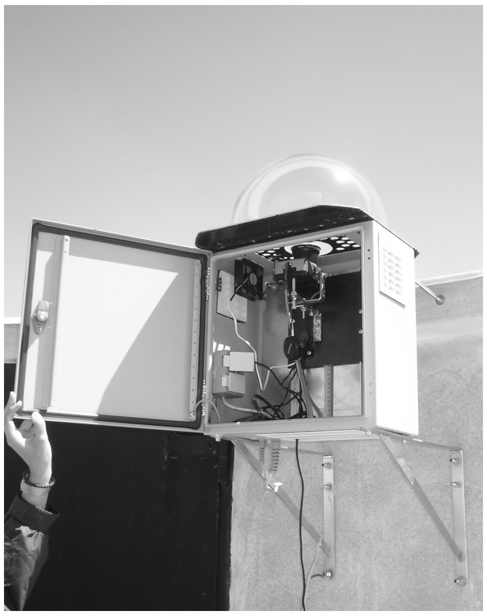


Figure 6.3: *Sky-scanning camera mounted on main building roof.*

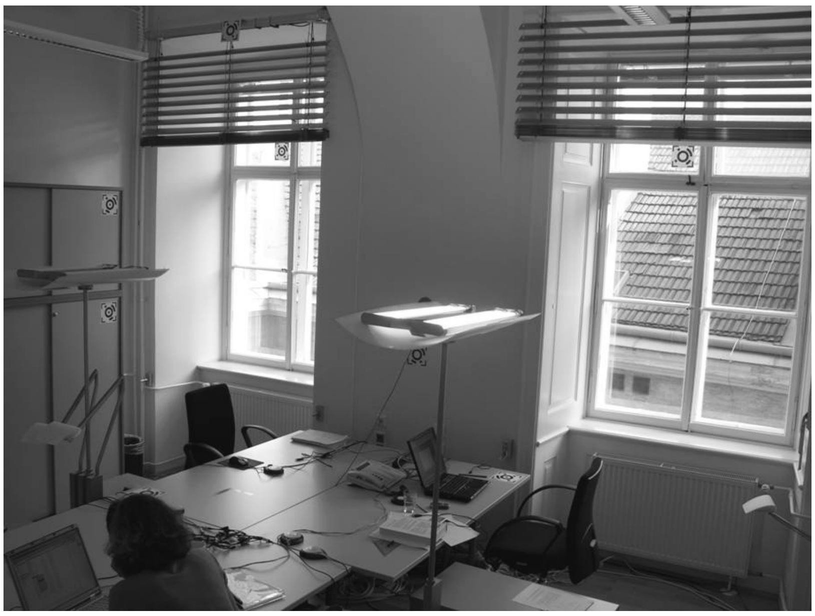


Figure 6.4: *Test space with uplights and window blinds.*

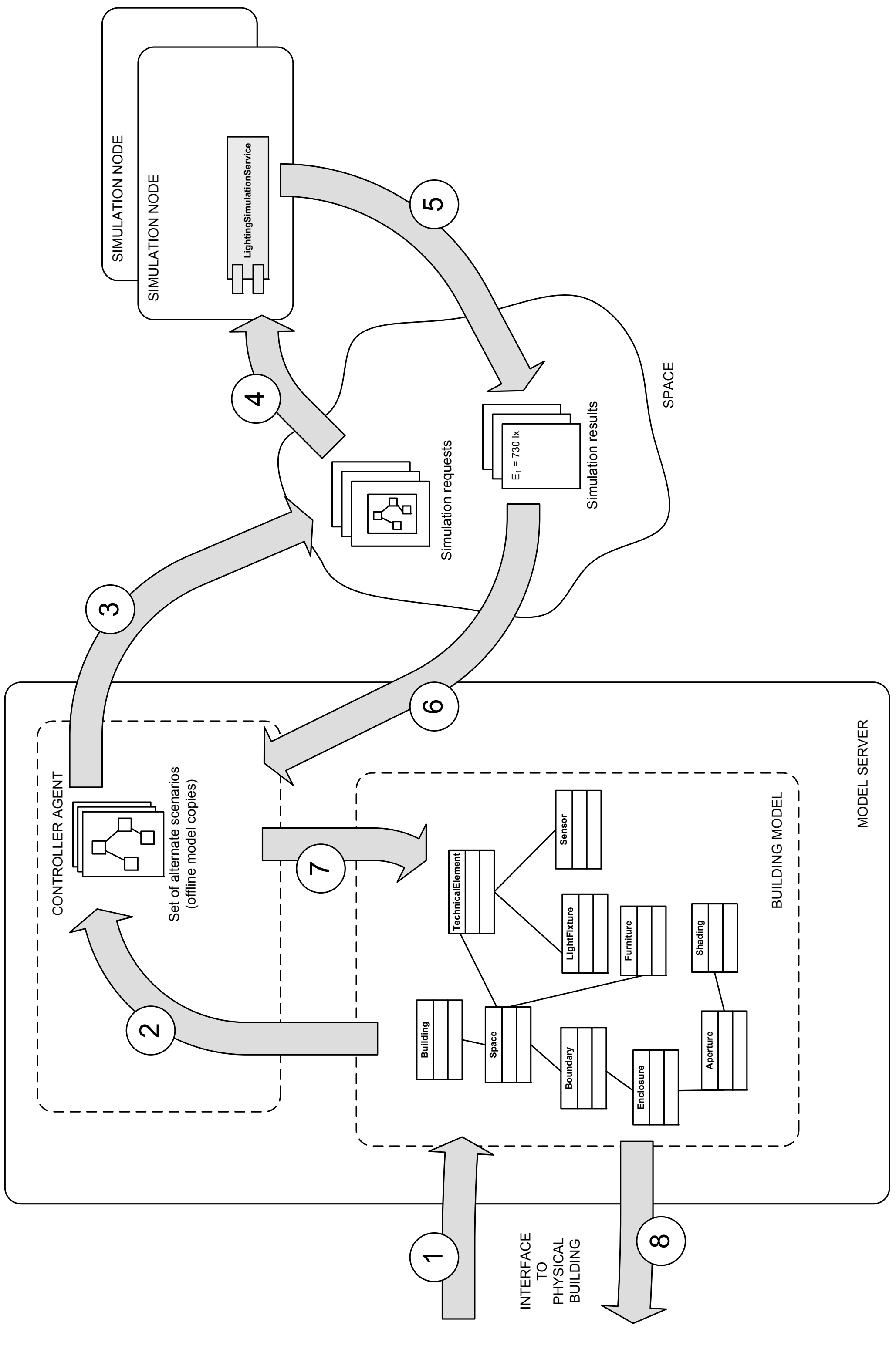


Figure 6.5: *Simulation-based lighting control workflow.*

The model is updated regularly through the building data interface (1). To run simulation-based control, a controller agent is set to operate on the controlled entity; i.e. the workplace illuminance in the space, with the goal of keeping the illuminance level within a preferential bandwidth. Control decision cycles are triggered periodically by a timer; they could also be started by significant changes in environmental conditions.

When a control cycle is started, the controller agent derives an off-line snapshot of the room's model including objects representing the relevant environmental parameters, such as the sky luminance distribution. From this as-is model, a set of prospective alternate states is derived by changing the states of controllable objects: in this application, the positions of window blinds, as well as the dimming levels of uplights (2). The controller now has a set of object networks, each representing a different state of the space.

The controller's task then is to compare these different scenarios and select the most desirable one. The comparison is based on a utility function that takes into account not only the expected illuminance level, but also power consumption due to electric lighting or cooling loads. While power consumption can usually be estimated using a simple function or table lookup, calculating illuminance requires physically accurate simulation. The controller therefore submits each model scenario, as well as a specification of the coordinates for which illuminance values are required, as a simulation request to the service space (3) and waits for all result responses to come back.

Simulation requests are picked up by distributed Java adapter processes (4) that convert the received object network to a textual representation and spawn the necessary simulation processes (see 8.2). The results – illuminance figures for the requested coordinates – are wrapped into response objects and placed back into the service space (5), where they are picked up by the original requestor (6).

Once all responses have been gathered, the controller can calculate their utility functions and use this to determine the best scenario. To trigger the required control decisions, the controller locates the respective objects in the live building model and calls specific methods – e.g. commandDimmingLevel() for an uplight object – on them (7). The model server then – transparently and asynchronously – sends the necessary commands to the modelled physical objects (8). This ends the control cycle: the controller agent thread suspends itself until the next cycle is triggered.

7 Conclusion

This chapter lists the contributions of this dissertation to the state of knowledge and identifies some areas for further work.

7.1 Contributions

- Existing model server architectures have been analysed to assess their capability for building operations support with a view toward simulation-based control (2.4.2). It was found that their designs are closely tied to design support, making them less suitable for other life-cycle phases.
- A complete set of key requirements for model servers for building operations support has been developed (chapter 3), with emphasis on both functional and non-functional requirements.
- A suitable system architecture for such a model server has been developed (chapter 4). The architecture is designed to provide high degrees of performance, modifiability, and availability. This is achieved by a introducing a flexible distribution design for different kinds of client applications (agents vs. remote clients/services), reliance on asynchronous communications, and runtime-changeable code.
- The design of a model server core has been described, based on a proven concurrency control method with novel extensions for write prioritisation and retroactive insertion of records (chapter 5). The concurrency control's performance has been evaluated experimentally.
- An integrated prototypical system setup for simulation-based lighting control has been described (chapter 6).

7.2 Further Work

7.2.1 Constraint Enforcement

In the current implementation of the model service, there is no mechanism to detect or prevent model changes inconsistent with the model definition (SOM, IFC) beyond some very basic checks based on data types. To prevent corruption of the model by faulty or malicious clients, it should be possible during runtime to specify integrity constraints to be enforced by the model service. This should include higher-level application-specific constraints that go beyond the basic model definition constraints, and it should be possible to specify complex scopes for constraints (class, object, contained-in-area, . . .).

Such a mechanism could have multiple uses. During the design phase, regional building regulations could be specified as model constraints to catch design errors (e.g. “exit doors must open to the outside”, “rooms must be at least 2.30 metres high”). During the operational phase, constraints could be used as a generalised alert mechanism to detect anomalies in building systems.

7.2.2 Security

The proposed architecture was designed under the assumption that only well-meaning users have access to it, and that there is no need to differentiate between users. This is a major omission that stands between research and application in a real-world scenario. Design for security is not trivial, even more so in distributed systems – and security-related risks spring from many unexpected sources (cf. Neumann (1995)). Adding functions to record which action was initiated by which user is expected to be fairly simple, with no disruption to the system architecture. Adding a full-fledged mechanism for restricting access, however, requires detailed security analysis that goes beyond the scope of this dissertation.

7.2.3 Support for Multiple Models and Merging

Maintaining multiple different, but more or less coupled models (Kiviniemi et al. 2005) in the server is not currently possible, although the architecture does not entirely preclude it.

Merging of models or model fragments is a typical challenge of design collaboration (Scherer et al. 2003). A component to aid in this task could increase the model server’s applicability to support of the design phase.

7.2.4 Model Navigation Support

Currently, the burden of searching for objects of interest is on the agents themselves. The model should be easily navigable in terms of spatial queries (cf. 3.2.1) as well as by queries on object relations. The latter could be based on a query expression language, as in the JXPath library.[1]

7.2.5 Distribution and Replication

As discussed in 4.2.3, there are good reasons to distribute the entire model across several nodes, while keeping a unified view to clients. Interlinking between such model partitions could be based on proxy objects that stand in for remote objects; however, it is a challenge to keep the number of proxies as well as the communications load low.

Replication is an important design tactic for increased availability. In its current form, the model server offers a starting point for 1:n master–slave replication (see 5.5) to provide “warm failover”. Automatic and possibly seamless migration of agents under such circumstances poses some interesting design questions.

7.3 Publications

As of this writing, various portions and reports on earlier stages of this work have been published in the following articles:

Icoglu, O., Brunner, K. A., Mahdavi, A., and Suter, G. 2004. A distributed location sensing platform for dynamic building models. In *Ambient Intelligence: Proceedings of the Second European Symposium*, number 3295 in Lecture Notes in Computer Science, pages 124–135. Springer-Verlag. doi:10.1007/b102265

Brunner, K. A. and Mahdavi, A. 2005a. A software architecture for self-updating life-cycle building models. In Martens and Brown (2005), pages 423–432

Suter, G., Brunner, K., and Mahdavi, A. 2005. Spatial reasoning for building model reconstruction based on sensed object location information. In Martens and Brown (2005), pages 403–412

[1]Apache Jakarta JXPath project, http://jakarta.apache.org/commons/jxpath/

Brunner, K. A. and Mahdavi, A. 2005b. The software design of a dynamic building model service. In Scherer et al. (2005), pages 567–574

Mahdavi, A., Spasojević, B., and Brunner, K. A. 2005. Elements of a simulation-assisted daylight-responsive illumination systems control in buildings. In Beausoleil-Morrison, I. and Bernier, M., editors, *Building Simulation 2005: Proceedings of the Ninth IBPSA Conference*, volume 1, pages 693–699

Brunner, K. A. and Mahdavi, A. 2006. Software design for building model servers: concurrency aspects. In Martinez, M. and Scherer, R., editors, *ECPPM 2006 – eWork and eBusiness in Architecture, Engineering and Construction: Proceedings of the European Conference on Product and Process Modelling*, pages 159–164. Taylor & Francis

References

Adachi, Y. 2002a. Technical overview of IFC model server. VTT-TEC-ADA-11, SECOM Co Ltd and VTT Building and Transport. URL: http://cic.vtt.fi/projects/ifcsvr/tec/VTT-TEC-ADA-11.pdf.

Adachi, Y. 2002b. Overview of IFC model server framework. In Turk, Ž. and Scherer, R., editors, *Proceedings of ECPPM 2002: eWork and eBusiness in Architecture, Engineering and Construction*. URL: http://cic.vtt.fi/projects/ifcsvr/.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. 1989. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan.

Barghouti, N. S. and Kaiser, G. E. 1991. Concurrency control in advanced database applications. *ACM Comput. Surv.*, 23(3):269–317. doi:10.1145/116873.116875.

Bass, L., Clements, P., and Kazman, R. 2003. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, second edition. ISBN 0321154959.

Bender, M. A., Cole, R., Demaine, E. D., Farach-Colton, M., and Zito, J. 2002. Two simplified algorithms for maintaining order in a list. In Goos, G., Hartmanis, J., and van Leeuwen, J., editors, *Algorithms – ESA 2002: Proceedings of the 10th Annual European Symposium*, number 2461 in Lecture Notes in Computer Science, pages 152–164. Springer.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. 1995. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM Press. doi:10.1145/223784.223785.

Bernstein, P. A. and Goodman, N. 1983. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483. doi:10.1145/319996.319998.

BLIS Project. SABLE – simple access to the building lifecycle exchange, 2005. URL: http://www.blis-project.org/˜sable/.

Bober, P. M. and Carey, M. J. 1992. Multiversion query locking. In Yuan, L.-Y., editor, *Proceedings of the 18th International Conference on Very Large Data Bases, Vancouver, Canada*, pages 497–510. Morgan Kaufmann.

Brügge, B., Pfleghar, R., and Reicher, T. 1999. OWL: An object-oriented framework for intelligent home and office applications. In *Cooperative Buildings: Integrating Information, Organizations and Architecture*, number 1670 in Lecture Notes in Computer Science, pages 114–126. Springer. doi:10.1007/10705432.

Brown, A., Rezgui, Y., Cooper, G., Yip, J., and Brandon, P. 1996. Promoting computer integrated construction through the use of distribution technology. *ITcon*, 1:51–67. URL: http://www.itcon.org/1996/3.

Brunner, K. A. and Mahdavi, A. 2005a. A software architecture for self-updating lifecycle building models. In Martens and Brown (2005), pages 423–432.

Brunner, K. A. and Mahdavi, A. 2005b. The software design of a dynamic building model service. In Scherer et al. (2005), pages 567–574.

Brunner, K. A. and Mahdavi, A. 2006. Software design for building model servers: concurrency aspects. In Martinez, M. and Scherer, R., editors, *ECPPM 2006 – eWork and eBusiness in Architecture, Engineering and Construction: Proceedings of the European Conference on Product and Process Modelling*, pages 159–164. Taylor & Francis.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-Oriented Software Architecture: A System Of Patterns*. Wiley.

Carey, M. J. and Muhanna, W. A. 1986. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst.*, 4(4):338–378. doi:10.1145/6513.6517.

Carriero, N. and Gelernter, D. 1989. Linda in context. *Comm. ACM*, 32(4):444–458. doi:10.1145/63334.63337.

Cayci, F., Callaghan, V., and Clarke, G. 2000. A distributed intelligent building agent language (DIBAL). In *Proceedings of the 6th International Conference on Information Systems Analysis and Synthesis, Orlando, Florida, 2000*.

Chan, A., Fox, S., Lin, W.-T. K., Nori, A., and Ries, D. R. 1982. The implementation of an integrated concurrency control and recovery scheme. In *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 184–191. ACM Press. doi:10.1145/582353.582386.

Citherlet, S. and Hand, J. 2002. Assessing energy, lighting, room acoustics, occupant comfort and environmental impacts performance of building with a single simulation program. *Building and Environment*, 37(8–9):845–856. doi:10.1016/S0360-1323(02)00044-6.

Clarke, J. A., Cockroft, J., Conner, S., Hand, J. W., Kelly, N. J., Moore, R., O'Brien, T., and Strachan, P. 2001. Control in building energy management systems: the role of simulation. In Lamberts, R. et al., editors, *Building Simulation '01: Proceedings of the Seventh International IBPSA Conference*, volume 1, pages 99–106.

Dahl, H. K. *EDMserver course slides*. EPM Technology, Oslo, 2005.

Davidsson, P. and Boman, M. 2005. Distributed monitoring and control of office buildings by embedded agents. *Information Sciences*, 171(4):293–307. doi:10.1016/j.ins.2004.09.007.

DeMichiel, L., editor. 2003. *Enterprise Java Beans Specification, Version 2.1*. Sun Microsystems.

DeMichiel, L. and Keith, M., editors. 2006. *JSR 220: Enterprise Java Beans, Version 3.0: Simplified API*. Sun Microsystems.

Dietz, P. and Sleator, D. 1987. Two algorithms for maintaining order in a list. In *STOC '87: Proceedings of the 19th Annual ACM Conference on Theory of Computing*, pages 365–372. ACM Press. doi:10.1145/28395.28434.

Dyreson, C. et al. 1994. A consensus glossary of temporal database concepts. *SIGMOD Rec.*, 23(1):52–64. doi:10.1145/181550.181560.

Eastman, C. M. 1999. *Building Product Models : Computer Environments Supporting Design and Construction*. CRC Press. ISBN 0-8493-0259-5.

EPM Technology, 2005. URL: http://www.epmtech.jotne.com/.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. 1976. The notions of consistency and predicate locks in a database system. *Comm. ACM*, 19(11):624–633. doi:10.1145/360363.360369.

Eurostep. ModelServer for IFC. Flyer, 2003. URL: http://ems.eurostep.fi/emsdoc/.

Faraja, I., Alshawi, M., Aouad, G., Child, T., and Underwood, J. 2000. An industry foundation classes web-based collaborative construction computer environment: WISPER. *Automation in Construction*, 10(1):79–99. doi:10.1016/S0926-5805(99)00038-2.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528. doi:10.1145/1071610.1071615.

Finch, E. 2001. Is IP everywhere the way ahead for building automation? *Facilities*, 19 (11):396–403. doi:10.1108/02632770110403365.

Freeman, E., Arnold, K., and Hupfer, S. 1999. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex. ISBN 0201309556.

Garcia-Molina, H. and Salem, K. 1992. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516. doi:10.1109/69.180602.

Goetz, B. September 2005. Java theory and practice: Urban performance legends, revisited. *IBM DeveloperWorks*. URL: http://www-128.ibm.com/developerworks/java/library/j-jtp09275.html.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. 2006. *Java Concurrency in Practice*. Addison-Wesley.

Greene, R. 2006. OODBMS architectures: An examination of implementations. White Paper 028.01, ODBMS.ORG. URL: http://www.odbms.org/.

Guillemin, A. and Morel, N. 2001. An innovative lighting controller integrated in a self-adaptive building control system. *Energy and Buildings*, 33(5):477–487. doi:10.1016/S0378-7788(00)00100-6.

Hapner, M., Burridge, R., Sharma, R., Fialli, J., and Stout, K. 2002. *Java Message Service Specification, Version 1.1*. Sun Microsystems.

Hassanain, M. A., Froese, T., and Vanier, D. 2003. Implementation of a distributed, model-based integrated asset management system. *ITcon*, 8:119–134. URL: http://www.itcon.org/2003/10.

Hemiö, T. and Noack, R. 2002. EMS developer guide. Technical report, Eurostep. URL: http://ems.eurostep.fi/emsdoc/.

Herlihy, M. P. and Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492. doi:10.1145/78969.78972.

Hohpe, G. and Woolf, B. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 0-321-20068-3.

Icoglu, O. 2006. *A Vision-based Sensing System for Sentient Building Models*. PhD thesis, Vienna University of Technology, Austria.

Icoglu, O. and Mahdavi, A. 2005. A vision-based sensing system for sentient building models. In Scherer et al. (2005), pages 559–566.

Icoglu, O., Brunner, K. A., Mahdavi, A., and Suter, G. 2004. A distributed location sensing platform for dynamic building models. In *Ambient Intelligence: Proceedings of the Second European Symposium*, number 3295 in Lecture Notes in Computer Science, pages 124–135. Springer-Verlag. doi:10.1007/b102265.

International Alliance for Interoperability. Industry Foundation Classes – release 2x: IFC Technical Guide, Oct 2000.

International Alliance for Interoperability. Industry Foundation Classes: IFC 2x edition 3, 2006. URL: http://www.iai-international.org/.

Jensen, C. S. and Snodgrass, R. T. 1999. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44. doi:10.1109/69.755613.

Kastner, W., Neugschwandtner, G., Soucek, S., and Newman, H. 2005. Communication systems for building automation and control. *Proceedings of the IEEE*, 93(6):1178–1203. doi:10.1109/JPROC.2005.849726.

Kiviniemi, A., Fischer, M., and Bazjanac, V. 2005. Integration of multiple product models: IFC model servers as a potential solution. In Scherer et al. (2005).

Kung, H. T. and Robinson, J. T. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226. doi:10.1145/319566.319567.

Lam, K. P., Mahdavi, A., Brahme, R., Kang, Z., Ilal, M. E., Wong, N. H., Gupta, S., and Au, K. S. 2001. Distributed web-based building performance computing: a Singapore–US collaborative effort. In Lamberts, R. et al., editors, *Building Simulation '01: Proceedings of the Seventh International IBPSA Conference*, volume 2, pages 807–814.

Lange, D. B. and Oshima, M. 1999. Seven good reasons for mobile agents. *Comm. ACM*, 42(3):88–89. doi:10.1145/295685.298136.

Schnittstelle BMS (BMS v2.3). Luxmate Controls, Dornbirn, Austria, 2001.

Mahdavi, A. 1997. Toward a simulation-assisted dynamic building control strategy. In Spitler, J. D. and Hensen, J. L. M., editors, *Building Simulation '97: Proceedings of the Fifth International IBPSA Conference*, volume 1, pages 291–294.

Mahdavi, A. 2001. Simulation-based control of building systems operation. *Building and Environment*, 36(6):789–796. doi:10.1016/S0360-1323(00)00065-2.

Mahdavi, A. 2004. Self-organizing models for sentient buildings. In Malkawi, A. M. and Augenbroe, G., editors, *Advanced Building Simulation*, pages 159–188. Spon Press.

Mahdavi, A. and Spasojević, B. 2006. An energy-efficient simulation-assisted lighting control system for buildings. In *Proceedings of PLEA 2006 – The 23rd Conference on Passive and Low Energy Architecture, Geneva, Switzerland*, volume 1, pages 565–570.

Mahdavi, A., Brahme, R., Kumar, S., Liu, G., Mathew, P., Ries, R., and Wong, N. H. 1996. On the structure and elements of SEMPER. In McIntosh, P. and Ozel, F., editors, *Design Computation: Collaboration, Reasoning, Pedagogy. Proceedings of the 1996 ACADIA (Association for Computer Aided Design in Architecture) Conference*, pages 71–84.

Mahdavi, A., Ilal, M. E., Mathew, P., Ries, R., Suter, G., and Brahme, R. 1999. The architecture of S2. In Nakahara, N. et al., editors, *Building Simulation '99: Proceedings of the Sixth International IBPSA Conference*, volume 3, pages 1219–1226.

Mahdavi, A., Suter, G., and Ries, R. 2002. A representation scheme for integrated building performance analysis. In *Proceedings of the 6th International Conference on Design and Decision Support Systems in Architecture*, pages 301–316. Ed H. Timmermans.

Mahdavi, A., Spasojević, B., and Brunner, K. A. 2005. Elements of a simulation-assisted daylight-responsive illumination systems control in buildings. In Beausoleil-Morrison, I. and Bernier, M., editors, *Building Simulation 2005: Proceedings of the Ninth IBPSA Conference*, volume 1, pages 693–699.

Martens, B. and Brown, A., editors. 2005. *Computer Aided Architectural Design Futures 2005 : Proceedings of the 11th International CAAD Futures Conference, Vienna, Austria*. Springer, Dordrecht. ISBN 1-4020-3460-1.

Building Information Models – Overview. National BIM Standard Project Committee, November 2006. URL: http://www.facilityinformationcouncil.org/bim/publications.php.

Neumann, P. G. 1995. *Computer-Related Risks*. Addison-Wesley. ISBN 020155805X.

Objectivity Inc. 2006. Objectivity technical overview: Release 9. Technical Report 9-OTO-0.

O'Sullivan, D. T., Keane, M. M., Kelliher, D., and Hitchcock, R. J. 2004. Improving building operation by tracking performance metrics throughout the building lifecycle (BLC). *Energy and Buildings*, 36(11):1075–1090. doi:10.1016/j.enbuild.2004.03.003.

Papadimitriou, C. H. 1979. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653. doi:10.1145/322154.322158.

Pargfrieder, J. and Jörgl, H. 2002. An integrated control system for optimizing the energy consumption and user comfort in buildings. In *Proceedings of the 2002 IEEE International Symposium on Computer Aided Control System Design*, pages 127–132. doi:10.1109/CACSD.2002.1036941.

Paton, N. W. and Díaz, O. 1999. Active database systems. *ACM Comput. Surv.*, 31(1): 63–103. doi:10.1145/311531.311623.

Philippsen, M., Haumacher, B., and Nester, C. 2000. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518. doi:10.1002/1096-9128(200005)12:7<495::AID-CPE496>3.0.CO;2-W.

PostgreSQL 8.2 Documentation. PostgreSQL Global Development Group, 2006. URL: http://www.postgresql.org/docs/8.2/static/.

Rawlings, J. 2000. Tutorial overview of model predictive control. *IEEE Control Systems Magazine*, 20(3):38–52. doi:10.1109/37.845037.

Reed, D. P. 1978. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology.

Reed, D. P. 1983. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23. doi:10.1145/357353.357355.

Rindel, J. H. 2000. The use of computer modeling in room acoustics. *J. Vibroengineering*, 3(4):219–224.

Salzberg, B. and Tsotras, V. J. 1999. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221. doi:10.1145/319806.319816.

Scherer, R. et al. 2003. Integriertes Client-Server-System für das virtuelle Bauteam: Gemeinsamer Abschlussbericht. Technical report, TU Dresden, Lehrstuhl für Computeranwendung im Bauwesen.

Scherer, R. J., Katranuschkov, P., and Schapke, S.-E., editors. 2005. *Proceedings of the 22nd CIB W78 Conference on Information Technology in Construction*, Dresden. ISBN 3-86005-478-3.

Schmidt, D. C., Mungee, S., Flores-Gaitan, S., and Gokhale, A. 2001. Software architectures for reducing priority inversion and non-determinism in real-time object request brokers. *Real-Time Systems*, 21:77–125. doi:10.1023/A:1011195304563.

Sethi, R. 1982. Useless actions make a difference: Strict serializability of database updates. *J. ACM*, 29(2):394–403. doi:10.1145/322307.322314.

Sharples, S., Callaghan, V., and Clarke, G. 1999. A multi-agent architecture for intelligent building sensing and control. *International Sensor Review Journal*, 19(2): 135–140. URL: http://iieg.essex.ac.uk/papers/multiagent.pdf.

Silberschatz, A. and Kedem, Z. 1980. Consistency in hierarchical database systems. *J. ACM*, 27(1):72–80. doi:10.1145/322169.322176.

Spasojević, B. and Mahdavi, A. 2005. Sky luminance mapping for computational daylight modeling. In Beausoleil-Morrison, I. and Bernier, M., editors, *Building Simulation 2005: Proceedings of the Ninth IBPSA Conference*, volume 3, pages 1163–1170.

Java 2 Platform Standard Edition 5.0 API Specification. Sun Microsystems, 2004.

Suter, G., Brunner, K., and Mahdavi, A. 2005. Spatial reasoning for building model reconstruction based on sensed object location information. In Martens and Brown (2005), pages 403–412.

Tschantz, M. S. and Ernst, M. D. 2005. Javari: adding reference immutability to Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 211–230. ACM Press. doi:10.1145/1094811.1094828.

Venners, B. Designing distributed systems: A conversation with Ken Arnold. Artima Developer, October 2002. URL: http://www.artima.com/intv/distrib.html.

Ward, G. J. 1994. The RADIANCE lighting simulation and rendering system. In *SIGGRAPH '94: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, pages 459–472. ACM Press. doi:10.1145/192161.192286.

Weiser, M. 1991. The computer for the 21st century. *Scientific American*, 265(3):66–75.

Zhu, Y. and Weng, R. aecXML framework. aecXML Technical Committee, 2001. URL: http://www.iai-na.org/aecxml/.

8 Appendix

8.1 The Luxmate BMS Interface

Luxmate BMS is a proprietary interface for the control and monitoring of various lighting-related devices, such as light fixtures and motorised shading, and HVAC devices (Lux 2001). A Luxmate system consists of devices that accept commands (e.g. dimming modules, light sensors), devices that send commands (input modules, control modules) and a communication bus (LM-Bus). A typical small-scale setup would consist of a number of lighting devices in a room connected to an input module and a small control pad with a touch-screen display. Various configurations of the devices can be pre-programmed on the control device and invoked later on. Configurations may include a presentation configuration with dimmed lights and closed blinds, a work configuration with opened blinds, active fluorescent lights and additional workplace lights, or a party configuration with only halogen lamps switched on to create a 'warm' ambience.

Additionally, a hardware interface module is available to control the connected devices from outside the Luxmate system through a 9600 bps RS232C serial connection[1]. The interface accepts simple ASCII command strings to address devices and set and get their specific values, such as a dimming level or blind positions. Responses are also returned as ASCII strings. A sample conversation between a client and the Luxmate system looks as follows:

```
> TLR1R2B6T2?
< TLR1R2B6T2S1W130E0
```

This sequence shows a client asking for the current state of a light fixture addressed as bus 1, room 2, device 6. Specifically, the device's value for type 2 (light intensity) is requested. The light fixture responds with its address followed by its current ambience setting ("Stimmung") 1 and the requested intensity value, in this case, 130 (on a scale ranging from 0 to 255).

[1]A TCP/IP-connected module exists as well, but requires an additional computer dedicated to Luxmate control and was therefore not used in our work.

The interface is very limited in its scalability, partly due to the slow serial connection, partly due to the Luxmate BMS protocol's design. Integration with the Java environment was achieved by implementing a functionality subset based on the protocol specification, using Sun Microsystems's Java Communications API to access the serial interface.

8.2 The RADIANCE Lighting Simulation System

The lighting simulation software RADIANCE (Ward 1994) provides physically accurate lighting simulation for model descriptions provided in a simple text format. While it is typically used for rendering photorealistic images, it can also provide numeric output to determine illuminance values for given coordinates.

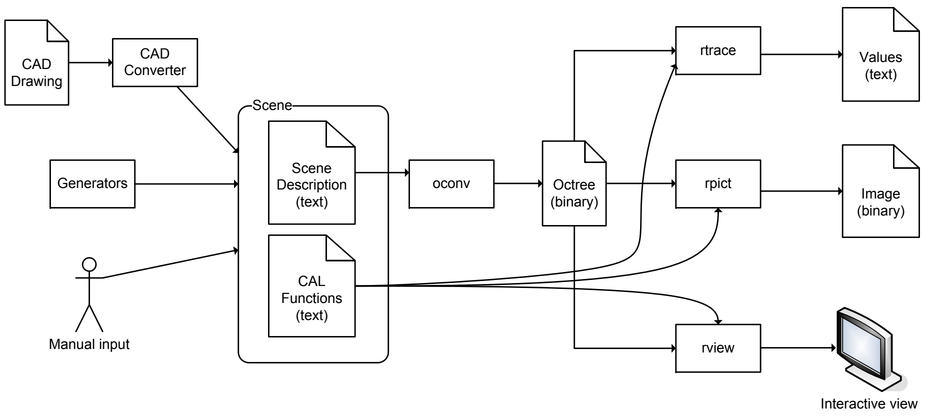


Figure 8.1: *RADIANCE components and the data flows between them.*

Following the typical "toolbox" architecture of UNIX-based software, RADIANCE comprises a collection of specialised programs that are connected in a pipes-and-filters combination to achieve the desired results (Figure 8.1). Scenes are described in a textual format in terms of geometry and material properties. A simple functional language can be used to model complex patterns of geometry and physical properties such as colour or luminance.

The scenes are converted to a proprietary octree file format, which can be used as input for programs that either provide rendered images (rpict, rview) or calculated values for a set of points and vectors (rtrace). The latter is particularly suitable for simulation-based lighting control, where illuminance values for a given area of interest must be calculated.

Integration with the Java environment was achieved by spawning the various RADIANCE processes from a Java adapter, feeding all input data through the spawned processes' standard input streams and receiving all output data from their standard output streams.