ARISTIDES FAUSTO ROSA MIGUEL

OTIMIZAÇÃO DA APLICAÇÃO COMERCIAL - *CONCEPT SPA & LEISURE*

UNIVERSIDADE DO ALGARVE
Instituto Superior de Engenharia
2020

ARISTIDES FAUSTO ROSA MIGUEL

OTIMIZAÇÃO DA APLICAÇÃO COMERCIAL - *CONCEPT SPA & LEISURE*

Mestrado em Engenharia Elétrica e Eletrónica
Especialidade em Tecnologias de Informação e Telecomunicações

Trabalho efetuado sob a orientação de:
Prof. Doutor Roberto Lam
Prof.ª Doutora Gabriela Schütz

UNIVERSIDADE DO ALGARVE
Instituto Superior de Engenharia
2020

OTIMIZAÇÃO DA APLICAÇÃO COMERCIAL - *CONCEPT SPA & LEISURE*

Declaração de autoria de trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

I hereby declare to be the author of this work, which is original and unpublished. Authors and works consulted are properly cited in the text and included in the reference list.

(Aristides Fausto Rosa Miguel)

Abstract

The applications dedicated to the spa and leisure are in huge demand due to the latter growth in developed countries. This type of application has been designed to manage activities at hotels, health spas and leisure venues. Usually, appointments are booked quickly and efficiently via a series of flexible grid control screens or through an availability search, where the system automatically finds available times. This project presents a solution to the problem of SPA & Leisure (SPA) application. Nowadays it generates thousands of records daily, which leads to increase data on the database. Currently, the application is facing a performance challenge due to the amount of data involved in the booking process. The objective of the present work is to propose a dedicated notification framework to optimize the SPA application. The main goal of this framework model is to reduce traffic on the network between the client application and database server. This would be achieved by using as much as possible data stored locally, rather than requesting it from the database every time it is needed. The search engine process is speeded up, keeping as many data as possible within the client application. A message broadcast framework will be created to maintain local data synchronized with the database. The main idea is to have a parallel system to keep watching data changes on the database and as soon as data change, a message will be sent to processes to inform that current data held is out-of-date and must be refreshed before use in the client application.

In the meantime, the Message Broadcast Framework was implemented. The tests

performed and the analysis of their results, are presented in Chapter 5.

Keywords: Messages broadcast, Network, Optimization, Spa & leisure, Thread, Cache.

Resumo

As aplicações dedicadas ao *spa* e lazer estão em grande demanda devido ao crescimento destes nos países desenvolvidos. Esse tipo de aplicação foi projetado para gerir atividades em hotéis, spas e locais de lazer. Normalmente, os compromissos são agendados de forma rápida e eficiente por meio de uma série de ecrãs com grelhas flexíveis ou por meio de pesquisa de disponibilidade, na qual o sistema encontra automaticamente os horários disponíveis. Este trabalho apresenta uma solução para o problema da aplicação SPA & Leisure (SPA). Atualmente, a aplicação enfrenta um desafio de desempenho devido à quantidade de dados envolvidos no processo de reserva. O objetivo deste trabalho é propor uma forma de notificação dedicada para otimizar a aplicação SPA. O principal objetivo deste modelo é reduzir o tráfego na rede entre a aplicação cliente e o servidor da base de dados. Isto será obtido usando os dados armazenados localmente, sempre que possível, em vez de solicitá-los à base de dados sempre que for necessário. Uma estrutura de transmissão de mensagens será criada para manter os dados locais sincronizados com a base de dados. O intuito é ter um sistema paralelo a observar continuamente as mudanças de dados na base de dados e assim que forem alterados, uma mensagem será enviada às aplicações para informar que os seus dados estão desatualizados e devem ser atualizados antes de serem utilizados.

Entretanto, o sistema de mensagens foi implementado. Os testes efetuados e a análise dos resultados são apresentados no capítulo 5.

Palavras chave: Difusão de mensagens, Rede, Otimização, “Spa e lazer”, Thread,

Armazenamento local

Ikaponomeha

Do the right thing,

be in the right place at the right time

and live the moment in the right way.

— Hawaiian Culture

To my wife, my son

and my parents

Acknowledgments

The elaboration of this work was only possible due to the support I received from several people and institutions that I would like to thank:

To my supervisors, Professor Doutor Roberto Lam and Professora Doutora Gabriela Schütz, for their guidance, feedback, correctness and friendship.

To the Universidade do Algarve, for the monetary support through the UID/Multi/00631/2019 project of the Portuguese Science and Technology Foundation (FCT).

To Conceptek Information Systems S.A., for their consent to use their information infrastructure to execute the technical part of this work.

To Shiji Group, for the monetary support through the full tuition payment of the second year of the Master's degree.

To all who contributed in some way to the accomplishment of this work: family, colleagues, and friends.

Last but not least, I would like to thank my parents and, in particular, my wife and son, for their love, support and patience.

Thank you!

Contents

List of Tables

List of Figures

Glossary

Activity In the *spa* industry the term "activity", "treatment" refers to services provided by the hotel or spa business.

Application In this document, the terms, application, application program or simply program are treated as synonymous. An application is a collection of bytes representing code and data which are stored in a file (Glass, 1993, p. 6). When an application is running, it is called a process (Stallings, 2018, p. 132).

Booking A Booking refers to an activity appointment. Abraham Pizam (2005, p. 540), calls reservations for bookings made in advance (from a few hours to several months). However, both terms can be used. A booking contains the following (mandatory) information: the place (complex), date, time, staff, location (room), and customer. It can often contain extras (resources), like products, equipment, and hired items.

Business Rules These are statements (or conditions) that tell a person whether they can perform a specific action that relates to how the business operates (Walsh, 2020). The application software working in this area facilitates the business operations and also controls business functions in real-time (Pressman, 2009, p. 7) (e.g., SPA appointment scheduling, POS transaction processing).

Complex In the *spa* industry, the term "complex" refers to the places where spa treatments or any other type of activities are executed. For example, it can be a building,

an open space, a set of massage rooms, fitness centres, tennis courts among others.

Data Encapsulation Data encapsulation refers to sending data where the data is augmented with successive layers of control information before transmission across a network. The reverse of data encapsulation is decapsulation, which refers to the successive layers of data being removed (essentially unwrapped) at the receiving end of a network. When a network device sends a message, the message will take the form of a packet. Each OSI (open system interconnection) model layer adds a header to the packet. The packet is then covered with some information directing it onward to a destination; this is analogous to the address on a letter in which the actual message is carried inside the envelope. Similarly, the message in the packet is encapsulated with some information such as the address of next node, protocol information, the type of data and the source and destination addresses.

Event The term "event" in the network context usually refers to the message sent over the network. However, "event" and "message" can be used interchangeably because, in this context, both refer to a message.

Message See Event.

Operator The term "Operator" is used to designate the person responsible for operations related to handling of activity appointments using a computer program.

Outlier An outlier is an observation or measurement which seems to be different from other values contained in a given dataset.

Overbooking The concept of overbooking—accepting more reservations than the available capacity (Bardi, 2002, p. 137).

Polling Polling in computer science, among others, refers to an operation that actively checks the status of an input/output (I/O) operation.

Process The name Process is vague and can have different meanings depending on the context where it is being used. Stallings (2018, ch. 5) suggests several definitions for "process", including:

- A program in execution.
- An instance of a program running on a computer.
- The entity that can be assigned to and executed on a processor.
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.

In software engineering, the term "process" is also used to describe application data flow. A process is a task that changes data and produces an output. It might perform computations, sort data based on logic, or direct the data flow based on business rules (Lucid Software Inc., 2019, sec. Symbols and Notations Used in DFDs).

Property In Hospitality terms, "property" refers to the place where services are offered, e.g., building, hotel, resort. In a programming language, it refers to a variable of a structure or a class.

Reservation See Booking.

Template In C++, the template implements the concept of a parameterized type. Instead of reusing object code, as with inheritance and composition, a template reuses source code. When a template is used, the parameter is substituted by the compiler (Eckel, 2000, p. 731). Templates are functions or classes that are written for one or more types not yet specified.

Time slot The term "time slot" refers to the time interval configured for a complex in the application. It specifies the interval of time to accommodate a reservation. As an example, if a complex has a "time slot" of 15 min, it stands that a reservation must start at intervals of 15 min starting from the complex opening time.

Transaction As per Silberschatz et al. (2006), transaction is a collection of operations that performs a single logical function in a database application.

Acronyms

ACID Atomicity Consistency Isolation Durability

AES Advanced Encryption Algorithm

ANSI American National Standards Institute

API Application Programming Interface

CPU Central Processing Unit

CSV Comma Separated Values

EACAPI Event Alert Client API

EAS Event Alerter Server application

EDS Embarcadero RAD Studio

EI Event Inspector application

FCFS First come, First served

FIFO First-in-First-out

GDPR General Data Protection Regulation

GUI Graphical User Interface

GUID Globally Unique Identifier

HTTP Hypertext Transfer Protocol

IEEE Institute of Electrical and Electronics Engineers

IM Instant Messaging

ISO International Standards Organization

JSON Javascript Object Notation

LAN Local Area Network

LOG Logger application

MBDM messaging-based data management

MBF Message Broadcast Framework

MBN Message Broadcast Network

MBS Message Server

OSI Open System Interconnection

PCI Payment Card Industry

PCI DSS PCI Data Security Standard

POD Plain Old Data

POS Point of Sale

RAD Rapid Application Development tool

RDBMS Relational Database Management System

SDF System Data Format

SOAP Simple Object Access Protocol

SP Script-parser application

SPA SPA & Leisure

SQL Structured Query Language

TCP Transmission Control Protocol

TCP/IP Transmission Control Protocol over Internet Protocol

UDDI Universal Description, Discovery and Integration

VCL Visual Component Library

WINAPI Windows Application Programming Interface

XML Extensible Markup Language

We are what we repeatedly do. Excellence, therefore, is not an act but a habit.

Aristotle, (384 BC – 322 BC)

1 Introduction

1.1 Motivation and scope

The spa and leisure industry has a considerable impact on the world economy. It is the main economic engine for many countries. For example, the revenue in the South African spa industry has more than tripled since 2008 according to Spa Business Handbook (2015, p. 70). The growth of tourism combined with the constant development of technology, promoted the creation of tools and applications to respond to the growing needs of the sector.

One of the market applications dedicated to the spa and leisure is the SPA & Leisure (SPA) application. Being developed for about twenty years, and installed in more than 60 countries around the world (Shiji Group, 2020), SPA is an application that has been

designed to manage activities at hotels, health spas and leisure venues.

Appointments are booked quickly and efficiently via a series of flexible grid control screens or through an availability search, where the system automatically finds available times. The system controls an unlimited number of services that are linked to their associated resources such as locations, equipment or employees (staff). Due to the program's flexibility and booking controls, reservations can be made for a wide range of diverse activities, either for one person or for groups. Activities include spa treatments, massages, tennis and padel court rental, tour bus rental, scheduled activities (e.g., master cook classes, gym classes, karate classes), and many others (Conceptek, 2019) .

SPA application generates thousands of records daily, which leads to an increase of data on the database. Currently, the application is facing a performance challenge due to the amount of data involved in the booking process. This project has the purpose to give the best user experience, independently of the amount of data involved, through optimization of the application.

1.2 Objectives

The objective of the present work is to create a dedicated notification framework to integrate with the SPA application. The main goal of this system is to reduce communication between the application and database server. This would be achieved by using as much as possible data stored locally, in place of gathering it from the database every time it is needed.

It is expected that this framework would take an important role in the optimization of the application. To accomplish this, the project will consist of the following phases:

- Implementation of the notification framework.
- Definition of messages structure.
- Integration with the application.
- Performance tests and analysis of results.

The messaging system will be integrated into the reservation creation process, concretely in the search engine component. We opted to start with the search engine operation because: (a) the search operation involves a significant number of queries submitted to the database in a short period of time, (b) with some of them getting the same data with low change frequency, and (c) the storage of data in local memory from the different queries can share the same storage mechanism.

1.3 State-of-the-art

Distributed applications connected to a database system implies data exchange over a network. The quantity of data transmitted depends on the type of system. There are systems where clients store a full copy of data locally, and others, where clients store only a subset of the data.

Two types of client-server architectures can be distinguished: "fat servers" and "fat clients". In the architecture with fat servers, the logic is implemented in the server, hence clients send requests to the server and the server provides the results. With powerful workstations as we have today, the architecture with fat clients has gained popularity. In this architecture, clients can cache necessary objects and perform intensive data processing, distributing the computation between the system components (Bukhari, 2012).

1.3.1 Data caching

Data caching in a buffer is a technique used in database systems to reduce disk latency, by storing objects in easily accessible storage at client-side (if there is sufficient storage available), so eventually, system response time is improved because objects do not have to be retrieved from the original source (Bukhari, 2012), (Silberschatz et al., 2006, p. 790).

In mobile environments where connectivity is not always guaranteed, keeping data cached locally permits applications to keep working in offline mode. However, this situation creates problems related to data recoverability and with consistency. The

former refers to the loss of data and the latter to stale data. There are some techniques to solve these problems when mobile hosts reconnect. For example, propagation of the local updates or broadcast of invalidation reports informing of out-of-date entries. However, these solutions have also issues like miss of invalidation reports, leading to loss of consistency (Silberschatz et al., 2006, p. 926). Our framework is not designed to operate in disconnected mode, therefore such limitations do not apply. But as a consequence, if the client-server connection is broken, non-committed transactions will be lost, meaning that user operations started but not completed will be aborted.

Bukhari (2012) in his thesis refers to three options existing in the literature about caching consistency: invalidation, propagation, and mix of the two options. *Cache invalidation* is the process to remove out-of-date local copies of objects (records) as a result of persistent changes in the database. To achieve consistency, the commit of the update transaction has to be delayed until all client caches have been invalidated. Only when the client application wishes to access an object that has been removed, it must request a copy from the database. In contrast, *cache propagation* relies on the replacement of the out-of-date copy with a fresh one from the database, soon the updates have been committed at the database server.

Our proposed model uses *cache invalidation* technique to keep cached objects up to date. Moreover, it is the issuer of the commit of the updates (client application) that initiates notification process. Other clients are notified after the commit of the updates has been completed.

1.3.2 Broadcasting

Local cached objects must be synchronized between clients and server otherwise, data will be out of sync, resulting in inconsistency issues. One of the techniques to address cache coherency, is broadcasting (Silberschatz et al., 2006, pp. 925-926).

Broadcasting is a mechanism to keep local data synchronized between clients and the server(s), by propagating notifications about data that have changed. Those notifications can be sent directly by the server or by clients. In the first case, the server sends a

notification to clients after the updates have been committed (Bukhari, 2012). In the second case, the client sends a notification after updates have been committed. In our model, the Message Broadcast Framework acts as an auxiliary system to maintain the information on clients up to date. Therefore, the server is used for synchronization messages delivery only. Moreover, the server knows nothing about the composition of objects transmitted and cached by client applications.

1.3.3 FirebirdSQL database notification approach

The FirebirdSQL open-source database (FirebirdSQL, 2019), implements a broadcast system, known as "Firebird Events", directly in its core engine. Those *events* are simple notification messages sent asynchronously by the database server to clients, through a secondary channel. This approach allows clients to complete normal tasks while waiting for notifications. The FirebirdSQL incorporates an *Event Manager* running in background, to broadcast notification events to the applications connected to the database server (Babuškov, 2005).

The main advantage of the FirebirdSQL events, resides on the fact that messages are sent directly by the database server, in response to a write operation on the database. Clients are immediately notified about changes, soon the transaction is committed. However, it has some drawbacks to work as a solution for our problem. First, it works only for clients connected to the FirebirdSQL database engine, thus not applicable for clients using another database systems, for example, Oracle, and MSSQL. Second, event notifications cannot carry other information than the event name and a useless counter[1], forcing clients to adopt the "polling" method to identify which records have been updated in the database.

[1]The counter contains the number of post-event method calls (for each type of event) in the context of a transaction. At transaction commit time, only one notification is sent per event type.

1.3.4 Conclusion

There are many books and papers written about caching objects in client-server database systems. Even so, the existing models have too much complexity, not necessary to integrate into our application. In most of the available models, the server is an active component in the system. It not only keeps the objects cached in memory but also is responsible to enforce their consistency.

In our proposed model, the server acts as a simple message delivery system. Soon all clients have been notified upon a received request, the server removes the request data from memory. The integrity of data is guaranteed by the database server and by the application, being accepted that some transactions may fail due to stale data on client at transaction commit time.

The ultimate goal of our framework is to broadcast a subset of data sufficient for clients to identify which objects need to be updated before next used. We believe that our model is the right solution for improving SPA & Leisure application performance.

1.3.5 Report structure

This report comprises six chapters. The present one introduced the theme and a brief state-of-the-art about existing related work.

The goal of this work is defined in Chapter 2, where we also describe the current system architecture.

In Chapter 3 we present our proposed scheme. We describe the new system architecture and the framework to be created.

The integration of the proposed framework in the client application is described in Chapter 4. We explain the classes created and necessary changes to made in the application to integrate the framework.

The performance tests model and results of the proposed architecture are described in Chapter 5.

In Chapter 6, the final chapter is where the main conclusions are done. We conclude with the future work to be carried on.

He who does not know how to look back at where he came from will never get to his destination.

Rizal, José

2

Current system and proposed solution

This chapter introduces the theme of the present work. We describe the current system architecture and detail some of the functionalities of the SPA application. Then we explain with some detail how the search engine component operates. Finally, we describe the existing problem we want to address and the proposed solution.

2.1 Overview

The SPA package is part of a suite containing several desktop applications focused on tourism and leisure business, particularly in golf and spa areas. These applications run on Windows environments. They were designed to work with several database systems such as Oracle, MSSQL, Interbase/Firebird, and Informix.

The SPA package is composed of two applications: the first one is the SPA Graphical

User Interface (GUI) Windows desktop application and the second one is the SPA Windows service application (Microsoft, 2019a). From here, we will use the term "SPA service" to refer to the service application. Both applications are built on a client-server database system architecture, i.e., some tasks are performed at the client-side and some tasks are executed at the server-side (Silberschatz et al., 2006, p. 783). This architecture is also known as two-tier architecture where end-user application communicates directly with the server, i.e., there is no intermediate between client and server (Stallings, 2018, sec. 18-9). The SPA (GUI) application implements a two-tier architecture. On the other hand, the SPA service provides a three-tier or multi-tier architecture. The multi-tier architecture consists typically in three layers as follows: client layer, business layer, and data layer. The client layer contains the user interface part of the application; the business layer contains all the business logic like the validation of data, calculations, data insertion, etc.; the data layer is where the database comes into the picture (Stallings, 2018, sec. 18-9). The SPA service works on the business layer.

Most often, the application runs on a Local Area Network (LAN). Some installations contain only a few workstations and others may contain several hundreds of workstations, running one or more instances of the applications, accessing the database simultaneously. Furthermore, the SPA service adds another level of complexity because it is called from an HTTP Web service, all over the world, increasing the number of active connections to the database. For these reasons, the service application has strong rules to keep the active connections to the database as low as possible by using a pool of active connections. The active connection pool allows to conserve system resources and improve performance due to the reuse of inactive connection slots whenever a new connection is required by the HTTP Web Service call. The communication between the HTTP Web service and clients, usual Web Browsers, uses SOAP format to transfer messages. In contrast, the communication between the HTTP Web Service and SPA service uses an internal message protocol over TCP/IP.

The concurrency control (see Silberschatz et al., 2006, p. 635), i.e., the rules governing the parallel access to data, is performed by the database and by the application. The

database guarantees transaction isolation and lock control. The application implements the business logic lock control using a central table for lock purposes. This table can be seen as a database temporary memory table holding the necessary information about running application instances connected to the database to prevent overbooking (Bardi, 2002, p. 137). For example, when scheduling a massage, the lock table is inquired to check if some other application instance has locked the room for the same time. The shared resources include staff, location, equipment, etc. Who first locks the resources, gains control over them, which means First-in-First-out (FIFO).There are no other policies that define priorities in the booking process. This mechanism is simple but efficient to control the sharing of the resources available.

The SPA package applications can be deployed using three different methods. The installation model depends greatly on the site's business model:

- **Installed on a client workstation.** The application is installed in all workstations that require a connection to the database. It has the advantage of spreading the processing through workstations instead of concentrating on an application server. On the other hand, it has the disadvantage of requiring workstations to have some processing power (which is no longer a problem in current days). Moreover, it requires installation and configuration of the application for all workstations. However, further updates do not require manual interaction due to the built-in update system.

- **Installed on application server.** The application is installed on a server computer (or more) running a GoGlobal or Citrix virtualization solutions (Beasley & Nilkaew, 2015, chap. 13). The application is then accessed using a Web Browser or a native tiny client. The biggest advantage is the availability of the application through a central point. Instead of the individual installation on each workstation, the application is accessed using a small specific client application or a Web Browser. Consequently, client machines do not need to be robust. Furthermore, the installation and initial setup time decrease considerably. However, if the network bandwidth is not adequate, delays may occur while running the application.

- **Installed as service application.** The application is installed on a server computer as a service. It is accessed indirectly through a Web Browser using the HTTP Web services provided. Normally it is integrated with a Website provided by the client (business place), which is developed by a third party company. The intention is to offer customers a search and booking service from the World Wide Web. The main advantage of this type of distribution is that it provides customers with on-site services using the latest technologies, not only as a way of attracting new customers but also ensuring greater comfort for existing customers. The disadvantages include the cost of the infrastructure, which needs to be designed to support higher traffic, and the rigorous management of the online services provided.

2.2 Application structure

The SPA & Leisure application principal purpose is to provide an electronic appointment agenda to manage a spa business. The application is divided into two main client business operations areas:

- Back office. The configuration is an essential part for the daily operations of the business place. Resources and rules are defined based on site business rules.
- Front office. Operators spent most of the daily time managing appointments. This includes the creation, rescheduling, cancellation, check-in, check-out of appointments and many other operations related to bookings.

The resources included in the application are listed below:

- Complexes, are those places of a site having bookings.
- Staff, are the people performing activities.
- Locations, are the places where activities take place.

- Activity groups, arrange similar activities in groups.
- Activity subgroups, serve to organize similar activities in subgroups.
- Activities, are the activities available for clients to book. They can be spa treatments or any other kind of activity. For example massage, manicure, pedicure, yoga class, fitness class, and country sports.
- Equipment, corresponds to the equipment that may be required in the activity. It can be a spinning bike, stones, and weight bars.
- Products, are optional items used in the activity. For example, massage oil, epilation wax, and gloves.
- Hired items, are optional items that are hired to be used in the activity. Towel, exercise ball, and tennis racquet are some examples of hired items.

Main activity booking rules (restrictions):

- Activity price table, defines the price over a period for a specific set of configurations (activity group, activity subgroup, activity, client type, client sub-type, etc.).
- Guest from, defines the rules for a client member to book activities for a guest over a period.
- SMS configuration, defines the text layout of communications via SMS and the rules to notify clients.
- Activity restrictions, specify which activities can be done by a client on a period. Usually, they refer to activities that cannot be performed in sequence (one after the other).
- Online restrictions, specifies which resources are available for online bookings and the rules to control operation.

- Online activity restrictions, specifies which activities can be booked over a period.
- Online staff restrictions, specifies which staff is available over a period.
- Online location restrictions, defines locations availability over a period.
- Turnaround, defines bookings gap to allow preparation and cleaning operations, before and after the booking takes place. Turnarounds are very important for daily operations because they simplify the creation of appointments by Operators as the system automatically inserts blocks on the agenda for the turnaround operations.

The activities are classified into three types: (a) Standard activities, are activities assigned to staff or a location but not to both, (b) Multilevel activities, are assigned to staff and locations and (c) Scheduled activities are activities scheduled over a period. These type of activities can be assigned to staff and locations simultaneously, although many activities have only staff or location.

The booking types are divided in:

- Single bookings. A customer having a single activity at a time.
- Add-in bookings. A customer having two or more simultaneous activities performed by the same therapist (staff).
- Concurrent bookings. A customer having two or more simultaneous activities performed by distinct therapists (staff).
- Package bookings. A package booking consists of several activities for one customer. This type of reservation has the particularity of having a sale price different from the sum of individual activities. Usually, used for multi-session treatments distributed along the day or week or to promote activities over a season.
- Classes. A class booking is the reservation of a scheduled activity. Abraham Pizam (2005, p. 79) refers to some typical activity classes like fitness centres, exercise classes, tennis, squash, racquet-ball, swimming pool, massage services, ice-skating, etc.

- Multi-bookings. Multi-bookings refer to a group of bookings with mixed activities. They can be composed of any of the previous booking types and are suitable for the management of large groups of clients having various activities. Multi-bookings are very suitable for company events, conventions, meetings, special events (weddings, parties, etc.), tours and many others (Abraham Pizam, 2005, p. 291). Typically, these reservations require preparation in advance to organize the necessary resources (staff, locations, equipment, products, etc.) to ensure their availability. Operators can specify a subset of staff, locations, equipment, hired items, and other resources and assign them to the group. Also, the partial activity payment functionally implemented in the SPA application, which provides several options to handle mass group payments, simplifies considerably the check-out process.

Application main dashboards:

- Agenda: shows the list of bookings of the day or week for a set of resources. It has the following view modes: *By staff*, shows the list of staff in the grid columns; *By location*, shows the list of locations in the grid columns; and last, *7-Day view* which shows the staff weekly schedule.
- Bookings List: shows bookings list using a grid similar to a spreadsheet.
- Classes: shows the scheduled activities with their participants.

Agenda dashboard is composed of:

- Resource, filters to limit what resources (staff, locations, activities, etc.) should be visible on screen.
- Calendar, to select the date to display bookings.
- Quick view panel shows booking summary information about the selected day's calendar, staff, and location.

- Agenda, depending on view mode, it shows the reservations of the day or week selected in the calendar.

2.3 Booking

The booking process consists of creating one or more activity reservations (bookings) using the facilities provided by the SPA application. A booking is identified by the place (complex), date, time, staff, location (room), customer, and a status. Bookings go through various statuses during their life cycle, which is divided into two sets, see Figure 2.1. The first set contains the active bookings, while the second contains the cancelled bookings.

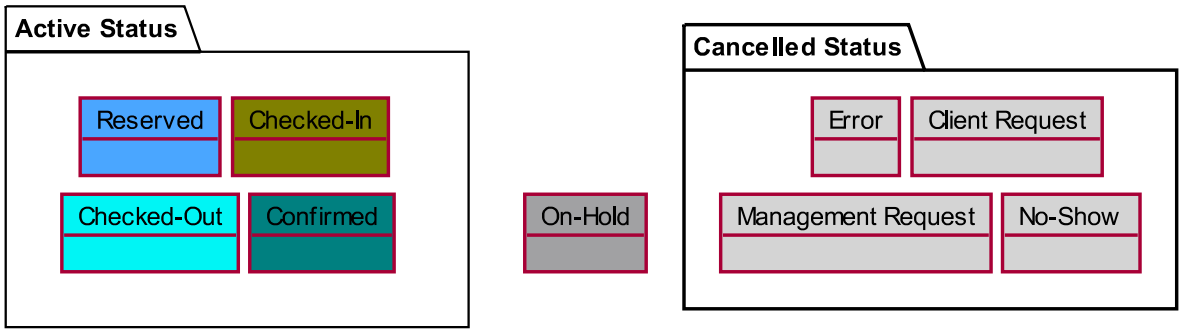


Figure 2.1: Booking status: on the left, active status; on the right cancelled status.

2.3.1 Active booking status

Each active booking status has a specific purpose, according to the business rules, and are subdivided into the following types:

- **Reserved.** This is the first status of newly created booking. Most of the times, no payment is required at this stage, but depending on the operations in place, a prepayment may be required to hold the reservation.

- **Confirmed.** This status is optional. Some business places use the confirmed status as part of their business flow; others use it depending on the activity or type of customer (members, guests, visitors, and others); while others do not use it at all.

- **Checked-In.** When the customer arrives at the place, it makes the check-in before having his activity. Usually, the length of time of the check-in status is associated with the duration of the activity.

- **Checked-Out.** After the customer has completed its activity, the operator checks-out the activity and the customer makes the payment. The amount to be paid by the customer is the difference between the activity sale price and the pre-paid amount (if any), as well as any discounts attributed. The check-out is the final status for an active booking.

2.3.2 Cancelled booking status

On the other hand, cancelled bookings have no distinct business operations except for the cancellation fees that may apply based on cancellation rules. The cancellations are subdivided into the following types:

- **Error.** It is a generic error but generally is used when the reservation is cancelled due to unusual circumstances. For example, bad weather, staff unavailable, location (room) out of service or in maintenance, staff sick, or system failure (application, database, network, and others).

- **Client Request.** The cancellation was requested by the client. There may be a payment penalty due to cancellation according to the cancellation rules in place. Generally, the amount to be paid depends on how early the reservation is cancelled. Late cancellations, highest amount to be paid.

- **Management Request.** The cancellation was dictated by the management. Depending on the circumstances, the activity may be rescheduled for another date. It is also generally common practice to offer a free activity session to the customer to overcome any inconvenience caused by the cancellation.

- **No Show.** The customer did not appear at the appointment time. Customer. may be charged for the cancellation, according to cancellation rules in place, like as for

the *Client Request* cancellation.

2.3.3 On-Hold status

The *On-Hold* status is a special case. It can be considered part of the active status or part of the cancelled status, depending on the operations being performed. An on-hold booking is a reservation that has not been completed, but instead of being cancelled, it has been placed on hold, to be resumed and completed later. The bookings on-hold prevent other reservations from being created at the same time for the same staff or the same location.

2.3.4 Booking status transitions

Figure 2.2 shows the state diagram of a booking. The thick blue arrows show the most common transition status of a reservation from its creation to its completion at check-out. The narrow blue dashed arrows show the alternate path that is also common in some business places. Finally, the red dotted arrows indicate the other possible paths that reservations can take.

To close the reservation, SPA calls the Point of Sale (POS) application to proceed with the payment. Upon completion of payment, the POS sets the final status of the reservation to *Checked-Out*.

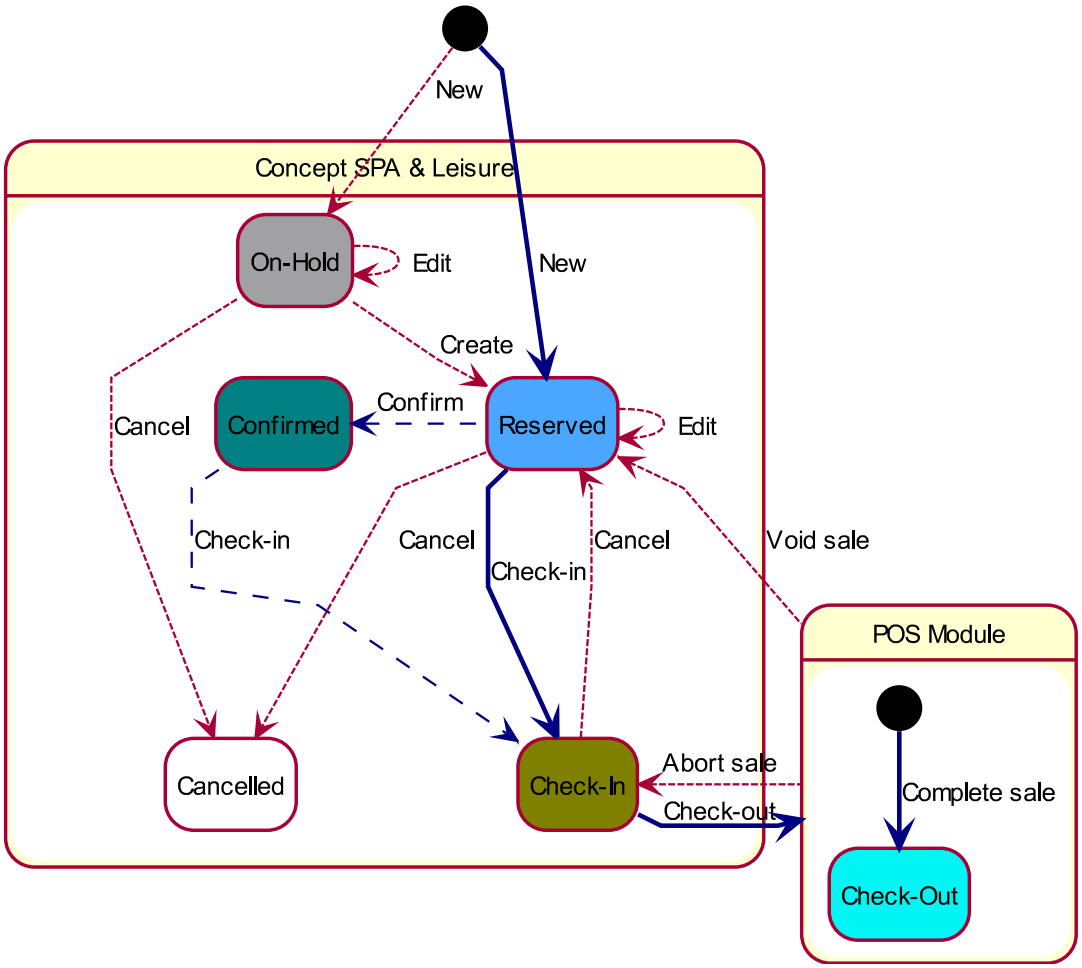


Figure 2.2: Booking status transition.

2.4 Search Engine

The purpose of the search engine is to (a) Provide facilities to help the operator to satisfy customer requests using processes well defined; (b) Find activity availability times distributing them fairly by staff and (c) Reduce operator interaction during the availability search process.

The search engine is used in many areas of the application including the booking creation and modification forms; agenda functionalities like cut & paste, drag & drop, change date & time, change staff or location. The search engine can handle multiple activities in the same session, which can be assigned to one or more customers. Activities can be individual (unrelated) or grouped (related) such as add-in, concurrent, and package bookings.

2.4.1 Availability times

To create a reservation, a set of validations must be executed to obtain the necessary data to initiate the booking process. The availability calculation will determine the reservation time, staff and location that can be assigned.

A reservation is identified in the SPA application by the mandatory attributes activity, date, start time, duration, and client; and the optional attributes staff, location, equipment, products and hire items. An availability time is essentially a time available to make a reservation. We define an availability time, the attributes activity, date, start time, staff and location, that together are available for the reservation to take place. The availability of each attribute is obtained according to the business rules defined in the system. For an example of availability time, we may have: activity "Acupuncture", on "20-April-2019" starting at "10 AM" and ending at "11 AM", executed by staff "John Smith" in location "Room 1". For each activity (reservation), the search engine calculates various availability times, providing several options for making the reservation, allowing the customer to choose the best option that suits best for him.

The calculation of the availability of the attributes is grouped into four groups: activity, staff, location, and client. Each group contains a specific set of validations and restrictions performed sequentially. Figure 2.3 shows the categories of validations and restrictions that the search engine uses to find availability times.

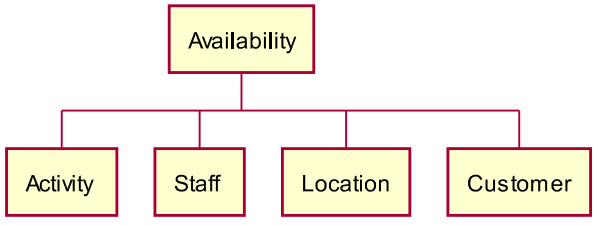


Figure 2.3: Availability calculation categories.

- **Activity availability** refers to the validation of data related to activities, for example, complex opening and closing time, activity season, and activity reservation order.

- **Staff availability** consists in validation of the staff. One of the most important

conditions related to staff, is the staff timetable, not only because an activity cannot be booked to an absent staff, but also because it has impact on the performance due to the fact it has to be verified constantly.

- **Location availability** accomplish the validations related to locations. For example, the maximum location capacity, current location availability.
- **Customer availability** contains the customer validations. These validations includes, for example, multiple activities sequence order, simultaneous reservations.

2.4.2 Availability modes

The search for availability can be *Passive Search* or *Active Search*. The use of one or other depends on the operation context. The *Passive Search* mode is suitable when the operator wants a list of possible availability times combinations, while *Active Search* auto-locks the activity with the first availability time found.

- **Passive Search.** The operation consists of searching for a specified number of possible availability times for each activity. No blocking is done during the search, therefore the same availability time can be suggested for more than one activity. The operation is executed for all activities and is restarted when the operator selects an availability time from the previous search operation. In the subsequent searches, the previous availability times blocked are no longer presented in the results. This mode is useful when the intention is to obtain all possible availability times for all activities in one run. Figure 2.4a shows the algorithm used to calculate the availability times. The search always starts from the initial conditions introduced by the operator. Then, the algorithm tries to find available times adding them to the list of results. After all the activities have been processed, the operator selects one activity and chooses one availability time to block; then, the search process starts again for all unlocked activities.
- **Active Search.** The operation consists of searching and blocking the first availability time found for each activity without operator intervention. The process is

performed for all activities using the following algorithm shown in Figure 2.4b: (1) select the first unlocked activity; (2) find the first availability time; (3) block the activity with the availability time found; (4) repeat steps (1), (2) and (3) until there are no more unlocked activities.

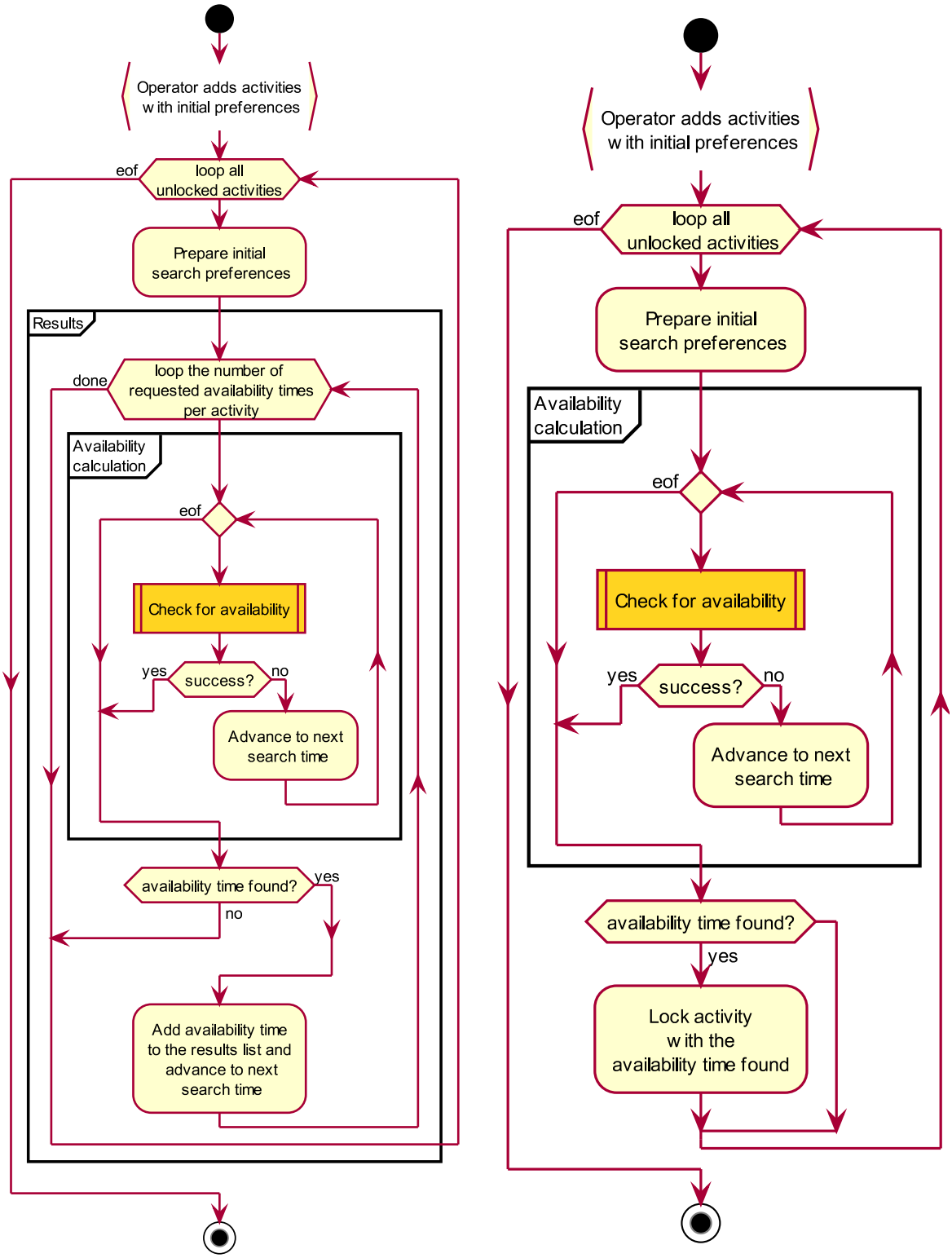
The search and lock process is interrupted when the algorithm is unable to find any availability time. All activities are blocked in one run (in contrast to Passive Search, in which the operator manually chooses the availability time to block). Despite this, the user can still change the availability time suggested, by instructing the application to return a list of availability times.

Looking at the two algorithms from Figure 2.4, we can see that they are very similar. In fact, the base algorithm is the same, but with the difference that the first one returns the possible availability times for all activities (to be blocked manually by the operator), while the second blocks all activities (in sequence) with the first availability time found (for each one).

To summarize, the booking process using *Passive Search* mode is slower, allowing the operator to choose the most suitable availability times for the customer, whiles the booking process using *Active Search* is faster, allowing the system to choose the most suitable availability times to optimize the schedule of staff members.

2.4.3 Availability calculation

The search engine searches for availability times, checking the conditions and restrictions which are grouped by activity, staff, location and customer. There are two methods to assign customer to activities. On the first method, the customer is added at the beginning of the booking process, when the operator is preparing the initial search conditions, before availability calculation. On the second method, the customer is added at the end of the search process, after availability calculation. The method chosen affects how the availability calculation is performed. Thus, if the customer is added at the beginning, the availability calculation will take into account the validations of the customer's group. If, on the other hand, the customer is assigned after the availability



(a) Passive Search

(b) Active Search

Figure 2.4: Availability search methods.

calculation, the customer's validations may result in a conflict, thereby invalidating the previously chosen availability time (e.g., a customer may already have another activity

at the same time or have other activities whose defined rules do not allow its execution). The sequence of the validations remains unchanged, whatever the method used to add the customer to the activity, that is, activity → staff → location → customer.

Figure 2.5 shows the algorithm used by the search engine to obtain availability times. From the initial preferences, the search process starts by checking the conditions and restrictions of the activity (see Figure 2.5a). If all validations were successful, the process will move to the validation of the staff's conditions and restrictions. The list of staff who performs the activity is obtained and sorted according to the activity rules defined in the system. Staff are checked one by one until the first one that meets the conditions necessary to execute the activity (see Figure 2.5b). In case of success, the process moves to the validations of the location. Like the staff, the locations assigned to the activity are obtained and sorted according to the rules defined in the system. The locations are checked one by one until one of them meets the conditions necessary to receive the activity (see Figure 2.5c). The last step is to validate the customer's conditions and restrictions (see Figure 2.5d). If a customer has not yet been added to the activity, these validations will be postponed, otherwise, they will affect the calculation of availability time.

If all the previous validations were successful, the selected time, staff and location correspond to the availability time to be added to the activity results. Now there are two possible situations: (a) with *Passive Search* operation, before moving to the next activity, the search process repeats until it reaches the desired number of results per activity; (b) with *Active Search* operation, the activity is locked with the availability time found.

To contribute to a fair distribution of activities by staff members, the search process attempts to equitably assign activities, taking into consideration their specific experience, the number of activities they have and how many points they have gained on the day. For each activity assigned, staff gains a number of points configured for the activity. This sort method is known in the SPA application as "staff rotation".

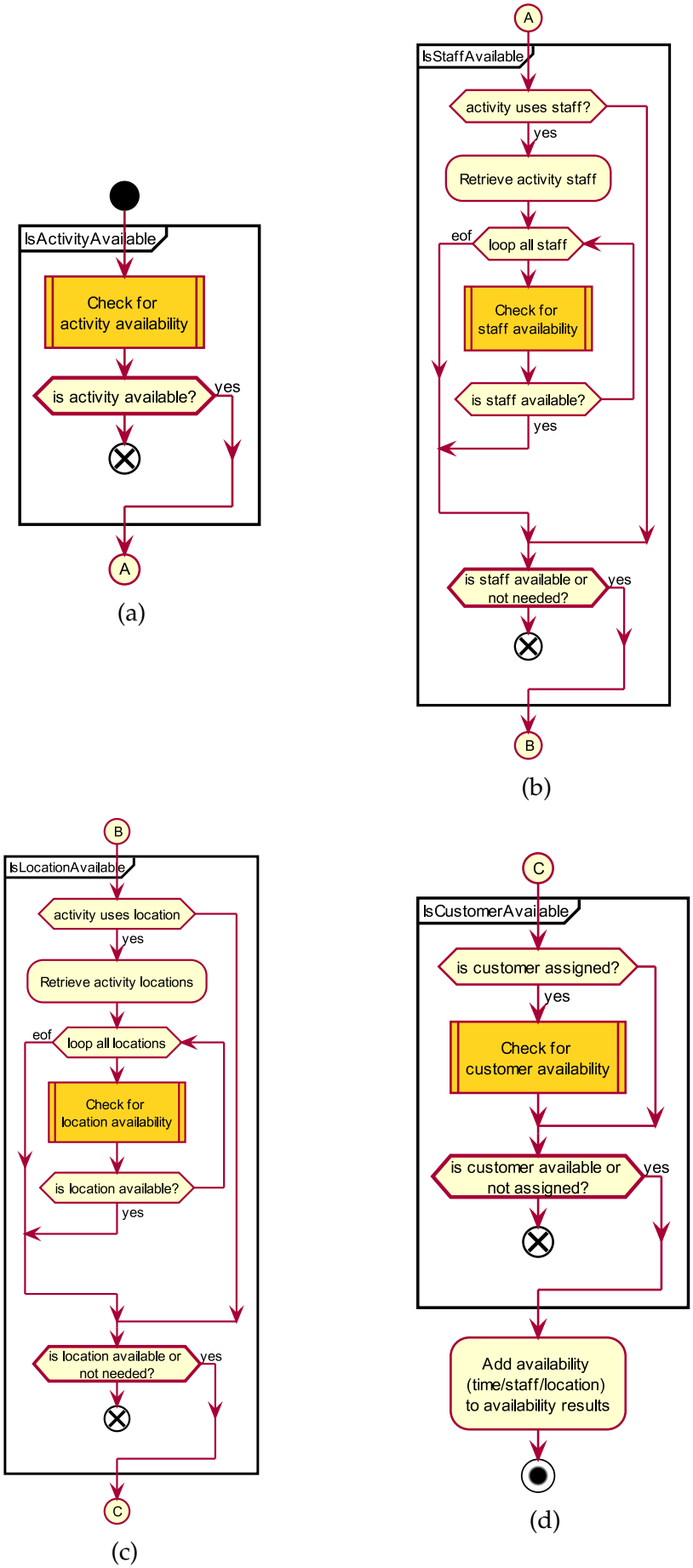
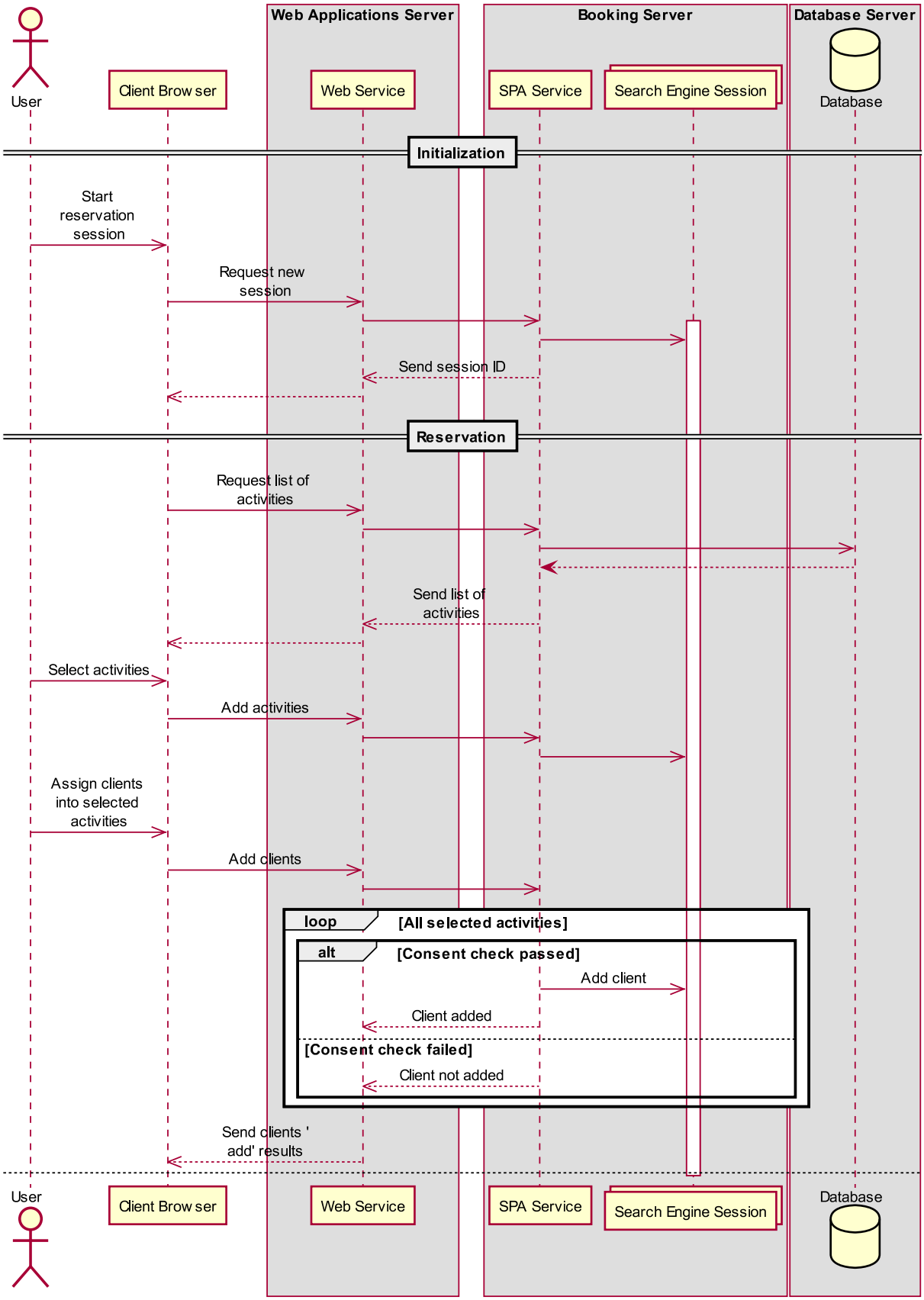


Figure 2.5: Availability workflow.

2.5 Online Search Engine

The proliferation of the Internet and the revolution of technologies have introduced a wide range of new marketing tools. The Internet allowed hotels to develop their websites to facilitate online bookings. Hotel chains receive a significant percentage of their reservations through their websites, free of commissions and other charges, reducing their distribution costs significantly by expanding their e-commerce (Abraham Pizam, 2005, p. 338).

The function of the Online Search Engine is to serve HTTP Web services requesting availability times. Figure 2.6 shows the typical sequence for an online booking using the *SPA Service*. As per the figure, we can see that there are four layers: *Client Browser*; *Web Applications Server*; *SPA Service* and *Database Server*. Each layer only interacts with the adjacent layer. The *Booking Server* plays the most important role in the booking process containing the rules which control the reservation process. The *Booking Server* corresponds to the *SPA Service*. Each client connection requires a *Search Engine Session* structure in memory. This structure implements the functionalities shown in Section 2.4, providing the necessary features for clients to search for available times and to book activities. In the sequence diagram of Figure 2.6, the messages "[1] Search for availability times" and "[2] Lock activities" require the most intensive computations in the whole booking process, in particular, affecting the *Search Engine Session* and *Database* components.


Web Applications Server
Booking Server
Database Server
User
Client Browser
Web Service
SPA Service
Search Engine Session
Database
Initialization
Start reservation session
Request new session
Send session ID
Reservation
Request list of activities
Send list of activities
Select activities
Add activities
Assign clients into selected activities
Add clients
loop
[All selected activities]
alt
[Consent check passed]
Add client
Client added
[Consent check failed]
Client not added
Send clients ' add' results
User
Client Browser
Web Service
SPA Service
Search Engine Session
Database

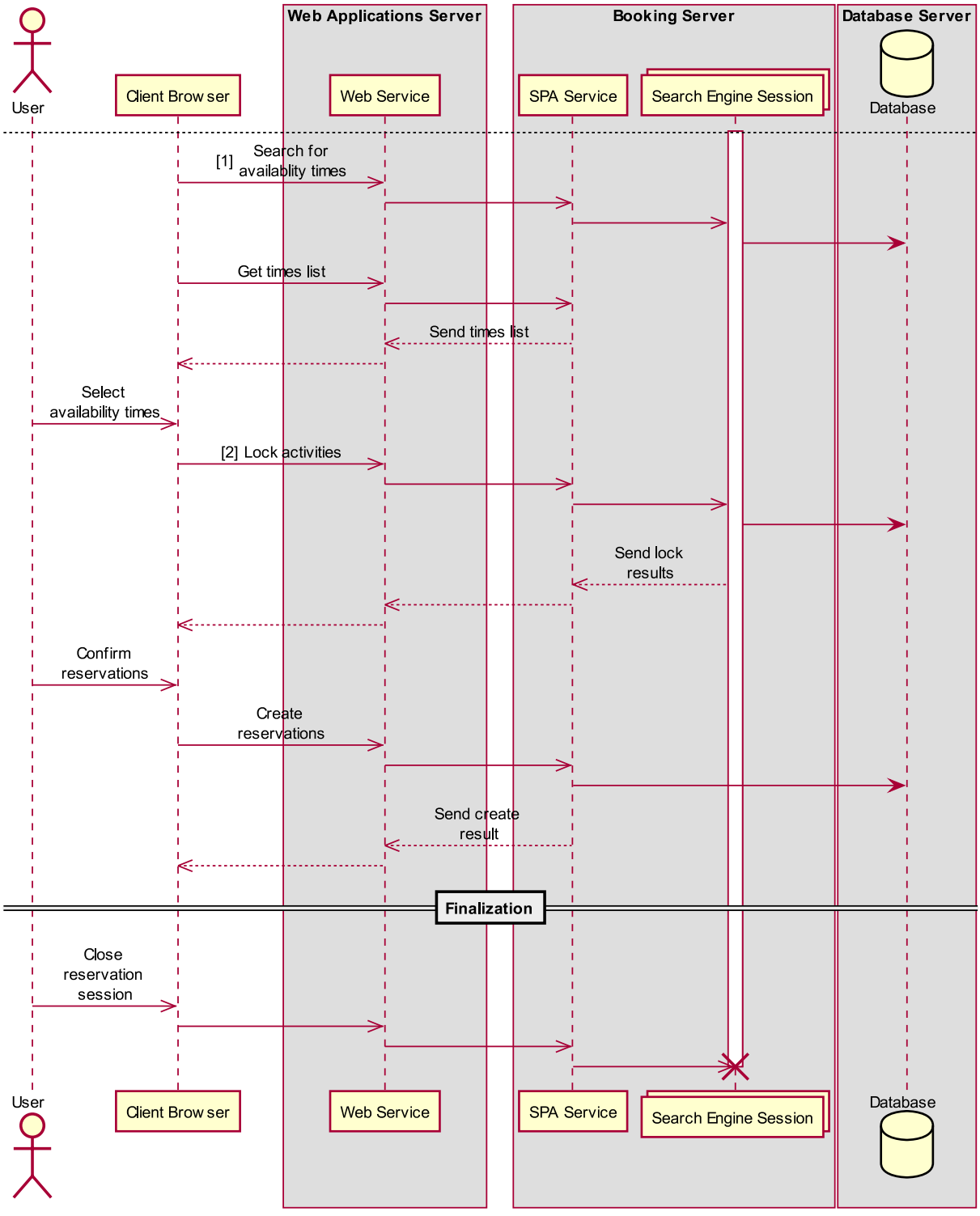


Figure 2.6: Online booking process overview.

2.6 Daily operations

Daily operations in a business place consist mainly of operations related to activity appointments (bookings). Typically there are two kinds of operations involved. The first kind of operations is related to the creation and modification whilst the second kind is related to check-in and check-out processes. Some sites have front desk operators dedicated only to do these operations, while on other sites, the operations are executed directly by the staff.

The creation and modification of booking appointments are generally executed in advance, i.e., activities are scheduled hours or days before they take place. Usually, the customer calls or arrives into the business place and asks for specific activities he intends to book and specifies at least preferred date and time. Based on the customer preferences, the operator offers a list of available options for the reservation. This process can be manual or automated using the booking engine to assist in the reservation process. In the first case, the operator manually selects all the appropriated options including date, time staff, and location. In the second case, the booking engine will return the first available time, staff, and location, based on the search criteria and restrictions that may apply. It is also common for the customer or spa manager to request changes to the existing reservation. For example, move the booking to other date or time, other staff or a different location. Depending on the requested changes, the operator may choose to execute the changes manually or use the booking engine to suggest alternative times if necessary.

The check-in and check-out operations are associated with the actual booking day. When the customer arrives, before the activity to take place, the operator marks the booking as check-in, indicating the customer is on-site. After the activity has been completed, the operator invokes the check-out process, so that the customer can make the payment to end the booking life cycle.

2.6.1 Manual booking process

From the application's agenda, the operator selects a staff or location and launches the manual reservation creation form. After that, the operator chooses the activity, staff, location, and date and time. At this point, the chosen activity, staff, and location are locked on the database at the requested date and time. As a result, the current selection is not available for use in other workstations trying to book at the same date and time. In the last step, the operator assigns the customer to the activity and completes the creation of the booking.

2.6.2 Creation of multi-bookings using the booking engine

The advanced reservation form uses a wizard to simplify the reservation of multiple activities step by step. From the application's agenda, the operator launches the advanced reservation form. From there, he adds one or more activities specifying the initial search conditions like date, time, staff, and location. After the selections have been made, the operator can choose to manually the lock of the activities (see Figure 2.4a) or let the application handle the search and lock operations without operator interaction (see Figure 2.4b).

2.7 Problem description

The problem can be described as follows: How can SPA & Leisure improve the performance of the booking process in response to the continuous increase of data in the database?

The booking process is composed of several restrictions that must be checked to complete the booking. The creation of a single booking requires a few calls to the database server to check restrictions and rules. However, the creation of a multi-booking containing several activities can increase this number to thousands of calls. Waiting a few seconds to complete availability checks is not an issue for one activity. On the other hand, waiting a few seconds on each activity of a multi-booking composed of various

activities will raise a major performance problem in the booking process.

WORKSTATIONS	ROOMS	THERAPISTS	BOOKINGS
138	329	388	2271
138	347	385	2251
145	340	376	2246
155	320	368	2134
143	344	386	2105
156	352	357	2062
141	352	364	2003
143	344	355	1999
150	342	357	1933
147	341	350	1909

(a) Client A

WORKSTATIONS	ROOMS	THERAPISTS	BOOKINGS
40	38	24	371
37	41	24	350
38	37	26	343
40	40	26	327
41	38	26	326
36	38	26	323
35	44	29	314
43	38	29	312
35	41	27	310
32	35	20	309

(b) Client B

Figure 2.7: Top 10 day bookings.

Figure 2.7 shows a top 10-day booking reports taken from two distinct clients with different business reservations volume. These two reports show the 10-days with more bookings including staff, locations and the workstations used to perform the operations related to the booking process. In the first report, from Client A (Figure 2.7a), the most busiest day shows a total of 2271 bookings executed by 388 therapists using 329 treatment rooms across 138 workstations. In the second report, from Client B (Figure 2.7b, the numbers are more modest, but even so, the busiest day has 371 bookings.

As we saw in Section 2.4, the Search Engine provides the available times by starting from the initial start time (i.e., the requested start time) and then increment it until a free time slot is found. If the first time slot space is near the requested time, the system quickly finds it using a few cycles. On the contrary, if the first free time slot is far from the requested time, the search must continue until finds it. The worst scenario is when there are no free time slots at all. In this situation, the search continues until it reaches the end search range. As a result, on a busy day, i.e., a day with few free time slots, the number of steps necessary is by far greater than on an empty day. Consequently, performance will decrease as the number of bookings increases.

Another point is the security of communications between client applications and the database server. Because the data processed by the SPA application may contain sensitive information subject to the General Data Protection Regulation (GDPR) (GDPR, 2020)

and to the PCI Data Security Standard (PCI DSS) (Council, 2020), its transmission over the network must be encrypted. The encryption layer requires additional processing on the client and server to encrypt and decrypt data. As a consequence, communication time increases. The Oracle `tnsping` command is a utility to determine whether or not an Oracle service can be successfully reached (Burleson Consulting, 2019). Essentially the `tnsping` command tests whether the Oracle Listener at the target IP is responding - and the time it measures is that response. The Figure 2.8 shows the results of the `tnsping` command using an encrypted TCP/IP channel and using an unencrypted TCP/IP channel in a existing client production environment. The encrypted TCP/IP channel returned 730 ms whereas the unencrypted TCP/IP channel returned 10 ms to respond. In other words, the response time of the encryption transmission takes 7300 % of the time to respond, compared to the response time of the unencrypted transmission.

The results obtained show that the data exchanged between client applications and the database server using an encrypted transmission channel considerably deteriorates the system's responsiveness, which cannot be ignored.

```
TNS Ping Utility for 32-bit Windows: Version 12.1.0.2.0 - Production on 17-JUN-2019 10:50:15

Copyright (c) 1997, 2014, Oracle.  All rights reserved.

Used parameter files:
C:\ORACLE\product\12.1.0\Client\network\admin\sqlnet.ora


Used TNSNAMES adapter to resolve the alias
Attempting to contact (DESCRIPTION = (ADDRESS_LIST = (ADDRESS = (PROTOCOL = TCPS)(HOST = 172.16.1.61)(PORT = 1521))) (CO
NNECT_DATA = (SERVICE_NAME = [redacted])))
OK (730 msec)

H:\>tnsping [redacted]

TNS Ping Utility for 32-bit Windows: Version 12.1.0.2.0 - Production on 17-JUN-2019 10:50:32

Copyright (c) 1997, 2014, Oracle.  All rights reserved.

Used parameter files:
C:\ORACLE\product\12.1.0\Client\network\admin\sqlnet.ora


Used TNSNAMES adapter to resolve the alias
Attempting to contact (DESCRIPTION = (ADDRESS_LIST = (ADDRESS = (PROTOCOL = TCP)(HOST = 172.16.1.61)(PORT = 1523))) (CON
NECT_DATA = (SERVICE_NAME = [redacted])))
OK (10 msec)

H:\>
```

Figure 2.8: Oracle TNS ping utility communication test results.

2.8 Proposed solution

For the reasons appointed above, we propose to optimize the search engine process by keeping as many data as possible on client-side. This technique is known as local cache (Naderializadeh et al., 2017; Vakali, 1999).

A message broadcast framework will be created to maintain local data synchronized with the database. The main idea is to have a parallel system to keep watching data changes on the database and soon as data change, a message will be sent to processes to inform that current data held is out-of-date and must be updated before use. The proposed schema, see Figure 2.9, is based on Instant Messaging (IM) applications (Wikipedia, 2019). Most of them use a server to register, control and serve the messages communications between clients (Shipley & Bowker, 2014). Some applications also use local cache to allow a peer-to-peer communication (González & Thiruvathukal, 2006). Using local cache in SPA application will: (a) reduce the traffic between client applications and server database and (b) decrease computational times in the server because the local cache will reduce the number of queries sent to the database server.

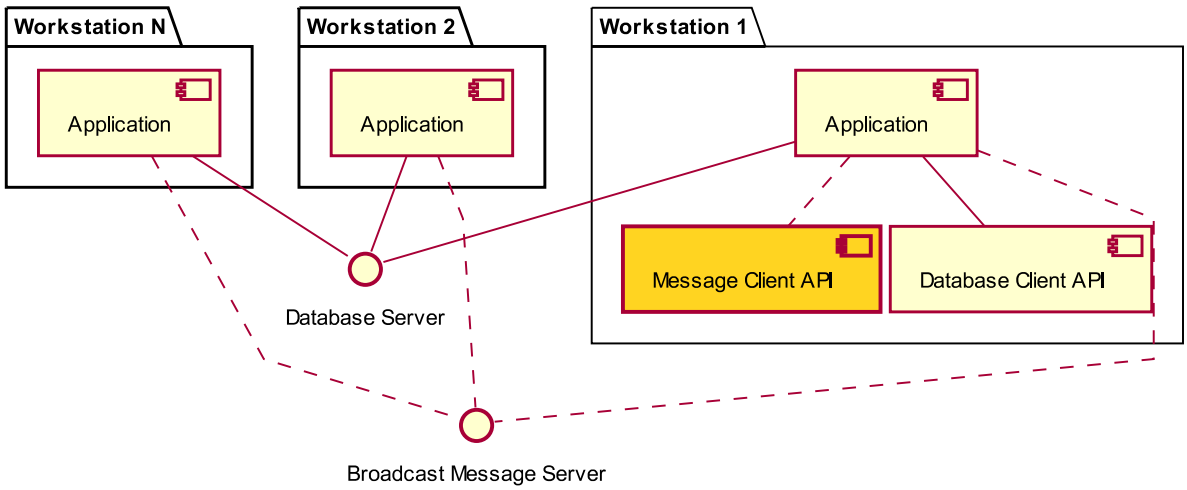


Figure 2.9: Network structure with the proposed Broadcast Message Server.

Figure 2.9 shows the database network and message notification network. In the current implementation (solid red lines), the interaction between client and database server are independent of each other. The database network comprises the information system and the message notification the status of the local data.

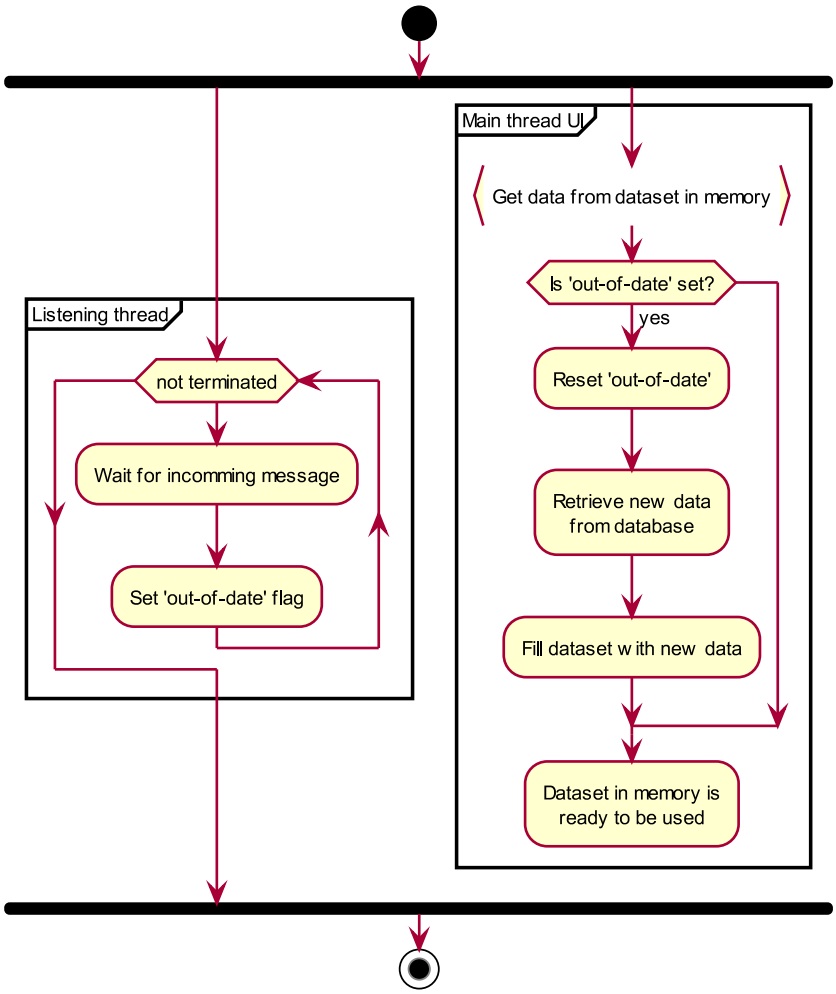


Figure 2.10: Cache implementation.

Figure 2.10 shows the cache implementation overview on the client side. When the application starts, a dedicated thread to listen for incoming messages is started and keeps waiting for messages until the application terminates. In the main thread, when a function needs to access some data, it first checks if the out-of-date flag[1] is set. If the flag is set, the function retrieves a fresh copy of data from the database and resets the out-of-date flag. If the flag is not set, it uses the data locally stored (if available, otherwise it must retrieve it from database). On the other hand, when the listening thread receives a message stating that some data in the database has changed, it sets the out-of-date flag to indicate to other threads that the related data locally stored is not up to date, hence requiring an update before use. This is the generic method used to synchronize locally stored data with server counterpart data.

[1]Each set of data has it own flag

We choose to use cache because:

- Application has been written for about twenty years having thousands of source code lines in its codebase.
- Search engine uses classes and functions shared by other areas of application.
- Rules and restrictions are too complex to be rewritten.
- There is already data cached in some sections of search engine.
- Caching only requires changes to specific parts of the search engine without touching in the core workflow.
- It is less prone to bugs in the current search algorithm.
- Other solutions would require more changes in source code and more time to implement them.

The optimization using local cache will consist of the following:

- **Load of data.** Data is retrieved from the database and kept in memory using local variables.
- **Usage of data**. When it is required, data is accessed directly from local memory. After use, memory is not destroyed. As a result, it is available for use next time.
- **Maintenance of data**. The message broadcast system notifies connected applications to inform that the data they stored in local memory is no longer synchronized with the database, hence needs to be refreshed before next use.

In conclusion, to address the performance problem we will use:

- Local cache of data.
- Messages broadcast framework to ensure local data is always synchronized with data in the database.

The indispensable first step to
getting what you want is this:
Decide what you want.

Ben Stein

3

Implementation Model

This chapter details the proposed solution to solve the SPA application performance problem. We introduce the Message Broadcast Framework, Message Server, and Event Alert Client API components and flows are presented: we show the internal structures of the server and client components. We describe the Event Alert Client API in detail with a particular focus on listener components and how they handle the notification (invalidation) messages. The last part, demonstrates the created messages structures with some illustrative examples of their use.

3.1 Overview

The proposed Message Broadcast Framework (MBF) described here consists in transmitting messages between applications in a LAN connected through a logical star

topology network. Client applications communicate with each other indirectly through the message server application. Applications do not know nothing about other applications they communicate with. They simply connect to the Message Server. It is the responsibility of this server to keep track of connected applications and manage the communications between them.

In this model all remote clients connect to a single server process. The server process receives requests, processes them and returns a response. To avoid blocking other clients while processing a client request, the server process is multithreaded. It executes some code on behalf of one client for a while, then it saves the internal context and switches code to another client. The overhead of switching between threads is low, typically only a few microseconds (Silberschatz et al., 2006, pp. 934-935).

To protect shared resources from multiple access from clients, critical sections are used. The synchronization of critical sections in Windows environments are lightweight (Stallings, 2018, pp. 322-329). They can only be used within one process, but have the advantage they don't need to switch to kernel mode (Microsoft, 2019d) (except when used with a synchronization primitive like an event or semaphore). The server process enters the critical section every time it needs to read or write to the shared resources. When there is no more work to do with the shared resources, it leaves the critical section. The current thread trying to acquire the lock will be blocked until the critical section is released by its owning thread. As soon as the shared resources are no longer needed, the critical section is released, to avoid blocking other threads for a long period of time.

The applications that form part of the MBF system include:

- **Event Alerter Server application (EAS)**: This application plays a leading role in the system. It manages the communications between client by maintaining a list of applications and the messages they are interested in receiving. Incoming messages are forwarded to all clients that have registered those messages.

- **Logger application (LOG)**: This application logs the messages received from the server process into a log database. It is mainly used for diagnostic and debugging purposes. In production environment it is not required.

- **Client applications**: The applications that connect to the EAS to use its functionalities to send and receive notification messages.

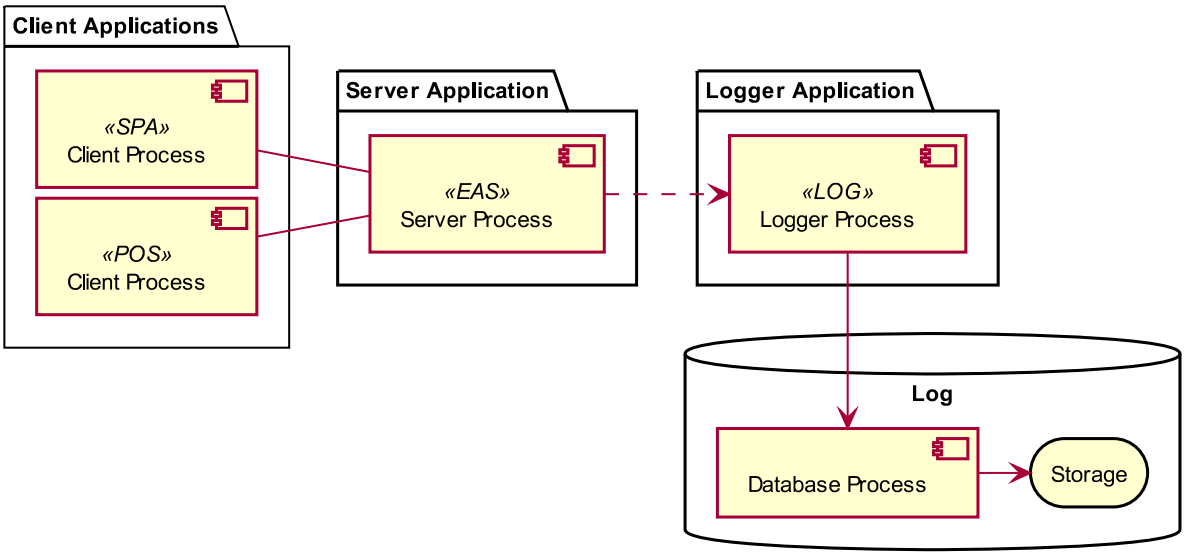


Figure 3.1: Message framework architecture.

Figure 3.1 shows the pipeline layout of the message system. The EAS communicates with clients over TCP/IP and with the LOG application through inter-process communication using a message queue.

Clients open a socket to the server and optionally register a set of messages to which they shall be notified. Clients send messages without worrying about who will receive them. The EAS then resends the message to each client that has registered the message. When a client starts, it creates a dedicated thread to read data from the socket.

The communication between clients and the server is asynchronous, i.e., clients send a message to the server and do not wait for the response about who had received it. The purpose is to allow the server to notify the other clients connected as fast as possible, in response to a message triggered by the business rules defined on the application. Because of the asynchronous nature of the communication, clients do not wait for acknowledgement of the messages sent. As a result, there is no difference in sending messages with registered clients or without registered clients, because clients do not block waiting for a response. If such a response is necessary, it must be addressed at the business layer of the application.

The MBF allows clients to send any message with any attached data, validating only the syntax, that is, if messages contain valid characters and if they do not exceed a defined maximum size. The semantics and meaning of each message is part of the defined client operation business rules. When the EAS receives a message without any clients interested on it, it simply discards that message.

3.2 Communication flow

Communication between client and server consists of three parts:

- **Initialization**: Client connects to the server and then registers which type of messages it wants to receive. Some of these messages may be private or also registered by other clients. The framework does not impose restrictions on what type of messages can be registered by the clients.

- **Work operations**: Client sends messages to the server and receives messages from other clients through the server. During normal working operations, it is also possible to register or unregister messages.

- **Finalization**: Client disconnects from the server. During the disconnect process, the server removes messages registered by the client.

Figure 3.2, represents the general communications flow. At the beginning, the client opens a connection to the server and registers the messages it wants to receive. For each new client connection, the server keeps track of its messages by adding them to internal records. After the initialization task, both client and server are ready to send and receive messages. The notification process starts when one client sends a message to the server. As soon as the server receives one message, it performs a search in the internal records to find which clients need to be notified. The message is then sent to those clients. This process is repeated whenever a client sends a message to the server. In the end, the client initiates the finalization step and disconnects from the server. After

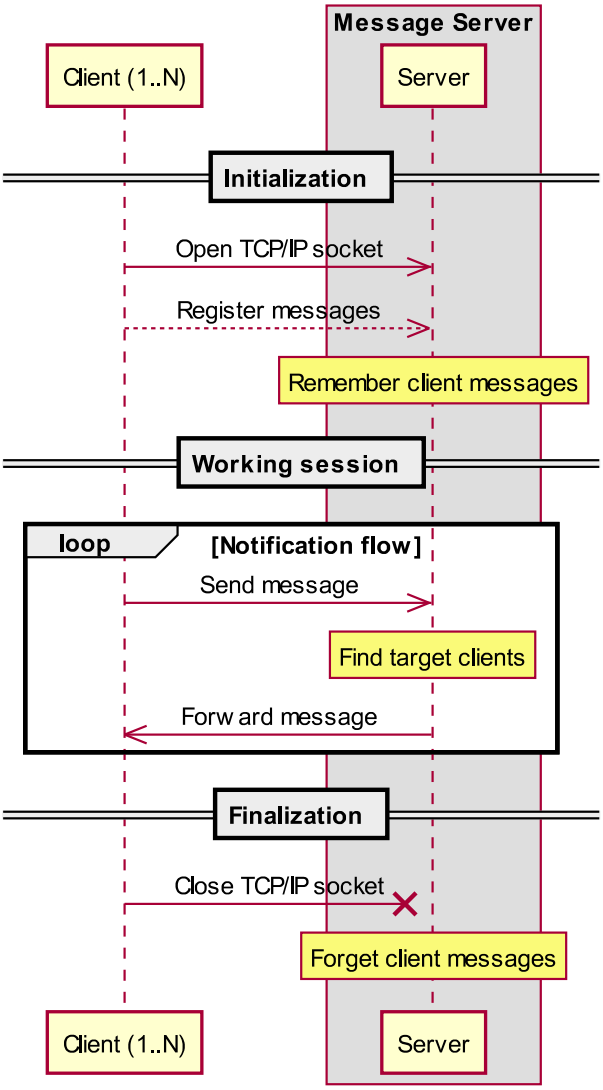


Figure 3.2: Communication between client and server.

the client has disconnected, the server releases resources that have been allocated for the client connection.

Figure 3.3 illustrates how the server's logger application integrates the communications flow between the client and the Message Server (MBS). The confirmation message is sent by the client after receiving a broadcast message with a flag requesting acknowledgement. The acknowledge message is requested by the server based on configuration, mainly for debugging purposes. The notification and acknowledgement processes independent and asynchronous, i.e., the server may continue sending messages to the client without receiving acknowledge from previous messages.

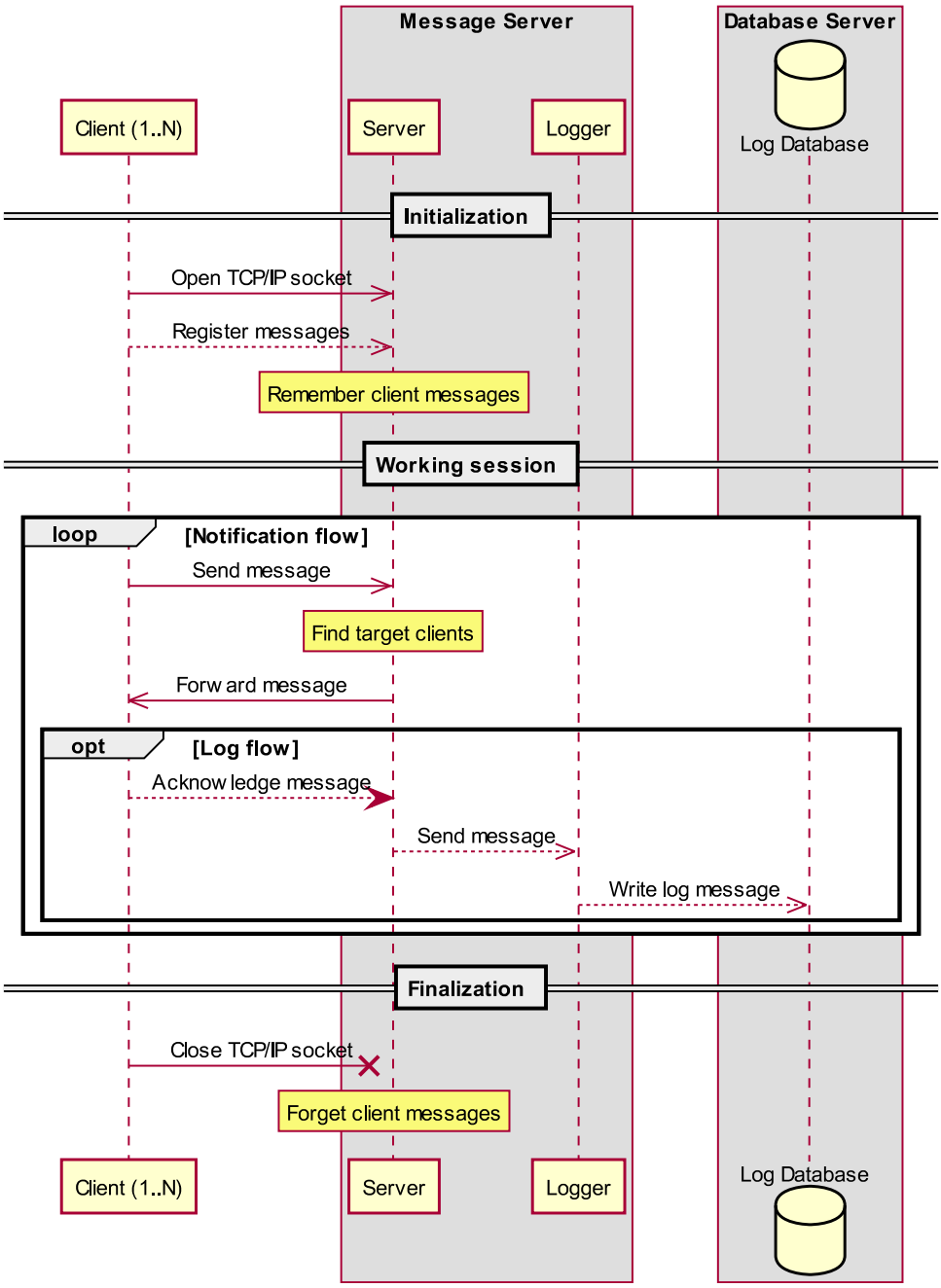


Figure 3.3: Communication between client, server and logger.

3.3 Server Applications

The framework server part is composed by the Event Alerter Server application (EAS), Logger application (LOG) and Database server, as can be seen in Figure 3.4. Each of them performs specific tasks: (a) EAS is the key application of the framework. This application implements a TCP server to which clients connect and perform the requests; (b) LOG is responsible to log the messages received and sent by the server. Messages

are decomposed and saved in a database to easier to analyse; and (c) Database server is responsible for storing the data received from the LOG, for later use in diagnostics or performance problems of the MBF.

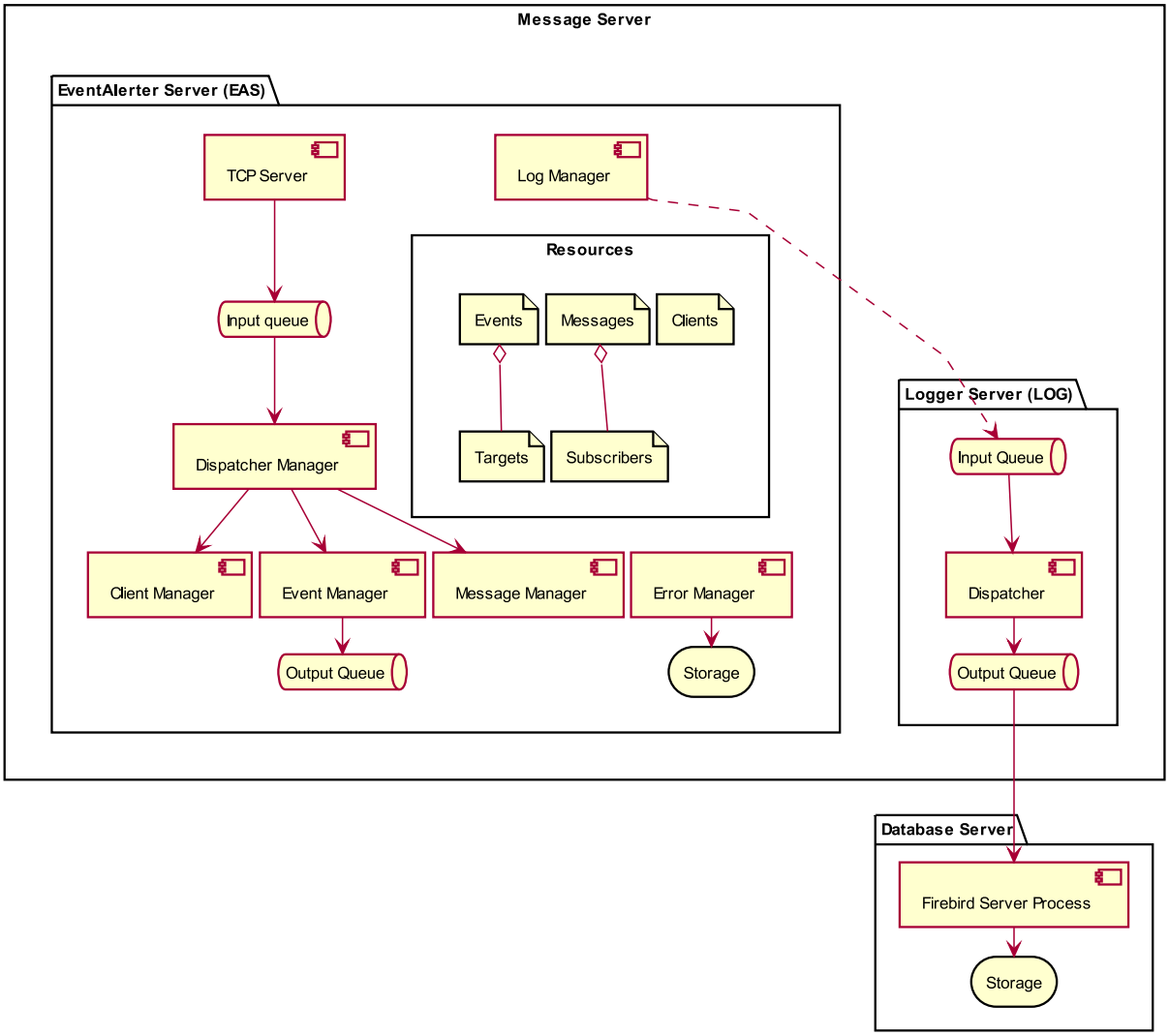


Figure 3.4: Server applications with internal components.

3.4 EventAlerter Application

The server application is a windows service program and aggregates the following components:

- **TCP Server**, is the component responsible for the communications using the TCP/IP protocol. The TCP server component used to perform this task is the

TIdTCPServer component available in the Embarcadero RAD Studio development tool (Embarcadero, 2019). The TIdTCPServer component encapsulates a complete, multi-threaded TCP server (Hower, 2006, sec. TIdTCPServer Class). The TIdTCPServer component allows multiple simultaneous client connections allocating a separate unit of execution for each client connecting to the server. Each client connection represents a task that is managed by the scheduler for the TCP server component. Listener threads use the scheduler to create an executable task for each client connection. Each client connected runs on a dedicated thread on the server. On Windows, the number of threads is limited by the available memory. By default, every thread gets one megabyte of stack space and implies a limit of around 2000 threads on 32-bit processes (Microsoft, 2009). Reducing the default stack size will allow more threads to be created, but will adversely impact system performance. In addition, threads are pre-emptively scheduled by the host operating system and allow no control over execution of the thread process (Hower, 2006, sec. TIdTCPServer Class). In the Figure 3.4, the *Input Queue* reads a single line from the input buffer, i.e., retrieves data from the input buffer until the end-of-line sequence is located or the maximum line length is exceeded. The *Output Queue* writes a single line to the output buffer. The TCP server then decides when flush data on the network. Each client connection has dedicated instances of *Input Queue* and *Output Queue*.

- **Dispatcher Manager**, decodes the received data from clients and calls the appropriated processes to perform the necessary tasks with the data.

- **Client Manager**, handles the task of maintaining a list of clients connected to the TCP server. Whenever a new connection is made, a new record entry is added to the *Clients* structure. On the contrary, when a client disconnects, the assigned record is removed from the *Clients* structure.

- **Message Manager**, manipulates the messages registered by clients. For each unique message registered, a new record entry is added to the *Messages* structure.

Each message entry contains local instance of the *Subscribers* structure whose purpose is to record all clients that have an interest in the message. When a client unregisters a message, the corresponding record in the message's *Subscribers* structure is removed. When a message has no more clients associated, it is removed from the *Messages* structure. Consequently, if the server receives a message with no clients waiting for it, it will simply discard it.

- **Event Manager**, forwards a received message to all clients who registered it. Once a message is received, the *Event Manager* creates a record on the *Events* structure to track the message state. Next, it locates the corresponding message record in the *Messages* structure and it forwards the message for all clients contained in its *Subscribers*[1] structure. For each target client, a record is added to the *Targets* structure allocated for the invalidation message. Both *Events* and *Targets* structures are used by the *Event Manager* to keep a trace of the message sent to clients. As soon as a client receives a message, it sends a confirmation to the server. The *Event Manager* then locates the client record in the invalidation message's *Targets* structure and marks it has confirmed. The *Event Manager* is tied with *Message Manager* but performs distinct operations. While the *Message Manager* maintains the list of clients associated with each message, the *Event Manager* uses that list to forward received messages to the proper clients. Clients have a maximum amount of time configured to confirm a received message. Failure to confirm will result in an error that will be handled by both the *Error Manager* and *Log Manager*.

- **Error Manager**, handles the errors triggered in the server process and on communications. The errors are divided into several error level categories. When an error occurs, it is categorized according to its severity and then written to a log file. The manager ensures exclusive write on the log file using a critical section.

- **Log Manager**, runs continuously on a dedicated thread monitoring the state of

[1] It is important to clarify that each message type record in *Messages* structure, contains a list of clients (*Subscribers*) that shall be notified when a message of that type arrives. In other words, clients receive only the message types they have previously registered.

the invalidation messages forwarded to clients (*Events* and *Targets* structures). This thread uses the WinAPI wait function *WaitForSingleObject()*, to allow block its own execution (Microsoft, 2019c). The wait function does not return until the specified criteria has been met. It this case, the criteria is when the time-out interval of one minute elapses. When the *WaitForSingleObject()* function returns, it loops the list of the messages sent to clients (*Events* structure) for confirmation. When the *Log Manager* finds a message in the *Events* structure whose time to confirm has elapsed, it performs the following operations:

1. Searches on the *Targets* structure assigned to the message, for clients that did not confirm the reception of the message and calls the *Error Manager* to handle the error.
2. If the LOG Service is running, then the message information is packed and sent to it using the WinAPI *SendMessage()*.
3. The invalidation message record is removed from the *Events* structure.

After processing all messages sent to clients, the *Log Manager* enters the wait state again. This cycle repeats until the EAS closes.

3.4.1 Memory Data Structures

To keep information necessary to work, several data structures are used in memory. Some of these structures are shared by different processes, requesting concurrent access. Because of this, all processes must acquire exclusivity access to read or write on these structures through the use of critical sections. All structures are implemented as linked lists, each one containing a list of records. To be fast, they all use indexes to allow fast access to individual records. Each record entry can be referred as a record or as an item. There is no difference using one term or the other.

The Visual Component Library (VCL) framework from Embarcadero RAD Studio (Embarcadero, 2019), contains several classes dedicated to work with lists. The following classes are the most common used:

- **TList.** *TList* is a class that implements a linked list. This class is the base class of all the other classes provided by the VCL. It provides the basic properties and methods necessary to work. The common features include insert, move, delete and compare items. Each item of the list is a pointer to some data allocated somewhere. The *TList* class does not deal with the details of the data contained in its items. Because of this reason, to add an item to list, first the memory for the item must be allocated and filled with data. Before delete an item from the list, the memory block associated must be explicitly released. Fail to do this, will fall in a memory leak.

- **TStrings.** *TStrings* class is derived from *TList* and is the base class of other classes to deal with strings. This class provides properties and methods to work with strings. This class takes care of memory management. In this case each item contains text characters and a pointer to a memory block (inherited from *TList*). It should be used in function as parameter declaration. It is not recommended to instantiate an object directly of *TStrings* because it is incomplete, i.e., some of the methods are abstract.

- **TStringList.** *TStringList* is the class specialized to work with strings. It derives from *TStrings* and provides the implementation to manipulate a list of strings. *Add()*, *Insert()*, *Delete()*, *Clear()*, *IndexOf()*, and *GetCount()*, are the most useful methods when adding and removing items from the list. TStringList has the capability to maintain automatically the items ordered to improve access time to a specific item. In addition there is a *Sort()* function that implements the quick sort algorithm which allows the programmer to custom sort the list based on a comparison callback function. A key feature is that each item of the list, can also point to an external block of data next to the string. The memory of the external block of data must be handled explicitly as in the *TList* class.

The MBF uses text to represent data. Therefore, the *TStringList* class, from VCL would be the natural choice. However, the data required for each record can not be

represented by text only. Each record has a specific structure according to the data involved.

To use the features offered by the *TStringList* class minimizing memory management, a class that essentially implements the functionality of the *TStringList* class will be used, but will automatically free the memory associated with each item. The same effect could be achieved with the *TList* class, but at the cost having to require explicit memory management and explicit conversions between data types. The new class to be created, *TListContainer*, makes use of C ++ language templates to implement a generic class for use to handle different data types. With this new class we achieved three important features:

1. One class which implements a linked list for any type of variable, either a primitive type (i.e., `int, double, char*, ...`), a Plain Old Data (POD) structure or a complex structure with data and methods. A POD structure is represented only by passive collections of field values (instance variables), without using object-oriented features (Wikipedia, 2018).

2. No need for explicit casts of container objects because the list contains the specific implementation for the object type used.

3. Memory garbage collector. The class frees memory from the internal list and the external object linked to each item. As a result, it is not necessary to explicitly free the memory of the linked object, because it will be released implicitly by deleting the item from the internal list (Eckel, 2000, p. 46).

Figure 3.5 shows the structures with the relevant properties used to control the Message Server workflow. These structures hold a list of records in memory providing fast access to records. *Clients*, *Messages*, and *Events* structures have global scope. *Subscribers* and *Targets* structures are local to each record of *Messages* and *Events* respectively.

Below we describe the contents of each structure using a text representation. We denote the variable that uniquely identifies a record in the structure by underline it.

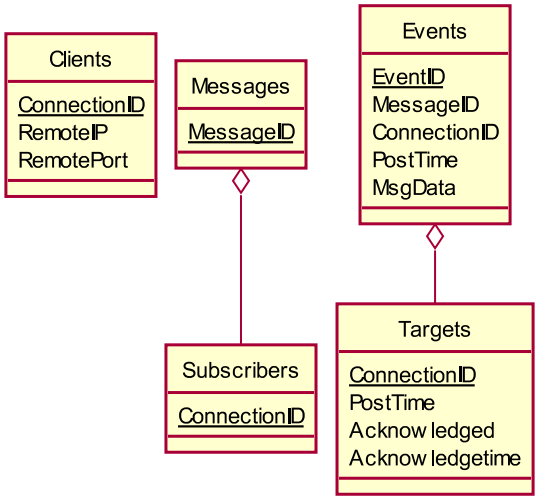


Figure 3.5: Message Server control structures.
(See text for more details.)

- **Clients**. This structure holds the list of clients connected to the EAS. Each record contains information about the connection that uniquely identifies the client application.

  Clients = (ConnectionID, RemoteIP, RemotePort)

  - ConnectionID, is the identification of the client.
  - RemoteIP, is the IP address of the client connection.
  - RemotePort, is the Port number of the connection.

- **Messages**. This structure contains the list of all unique messages registered by clients. A message is identified by a case sensitive string with a maximum of 64 characters length. The first character must be a letter or underscore. The characters following may be underscores, letters, or digits.

  Messages = (MessageID, Subscribers)

  - MessageID, is the message code.
  - Subscribers, is the structure containing the list of clients to forward the message.

- **Subscribers**. This structure contains the list of clients interested in the message. Each message record has a Listener structure assigned.

  Subscribers = (ConnectionID)

  - ConnectionID, is the identification of the client.

- **Events**. This structure holds a list of the messages that have been forwarded to clients, allowing the server to know the state of forwarded messages. An important property is a maximum time for the message target clients to confirm they have received it.

  Events = (EventID, MessageID, ConnectionID, PostTime, MsgData, Targets)
  - EventID, is the unique identifier for the message.
  - MessageID, is the message type of the message.
  - ConnectionID, is the sender of the message.
  - PostTime, is the time that server receives the message.
  - MsgData, is the data associated the message.
  - Targets, is the structure containing the list of clients to which the message was forwarded.

- **Targets**. The structure contains the list of clients for whom the message was forwarded. Clients must confirm they have received the message before the maximum confirmation period time has been reached. This information is useful for diagnosing connectivity problems or incorrect data cached by applications.

  Targets = (ConnectionID, PostTime, AcknowledgeTime, Acknowledged)
  - ConnectionID, is the identification of the client.
  - PostTime, is the server time when the message was forwarded to client.
  - AcknowledgeTime, is server time of the message confirmation by client.
  - Acknowledged, is a flag indicating that client has confirmed the receive of the message.

3.5 Logger Application

The Logger application (LOG) is a Windows service responsible for logging into the database, the result of the communications between clients and the EAS. In the EAS, the *Log Manager* sends a WM_COPYDATA message (Microsoft, 2019e) to the LOG input queue window procedure. When the internal message WM_COPYDATA is received, the LOG allocates memory to hold the received data, then it calls the *PostMessage()* WinAPI to forward that message to the main windows procedure thread. The internal message used to forward the message is an application-defined message created for this purpose. This message is labelled as WM_EVENTALERT[2] (Microsoft, 2020). When

[2]WM_EVENTALERT is defined as WM_EVENTALERT = WM_USER + 1.

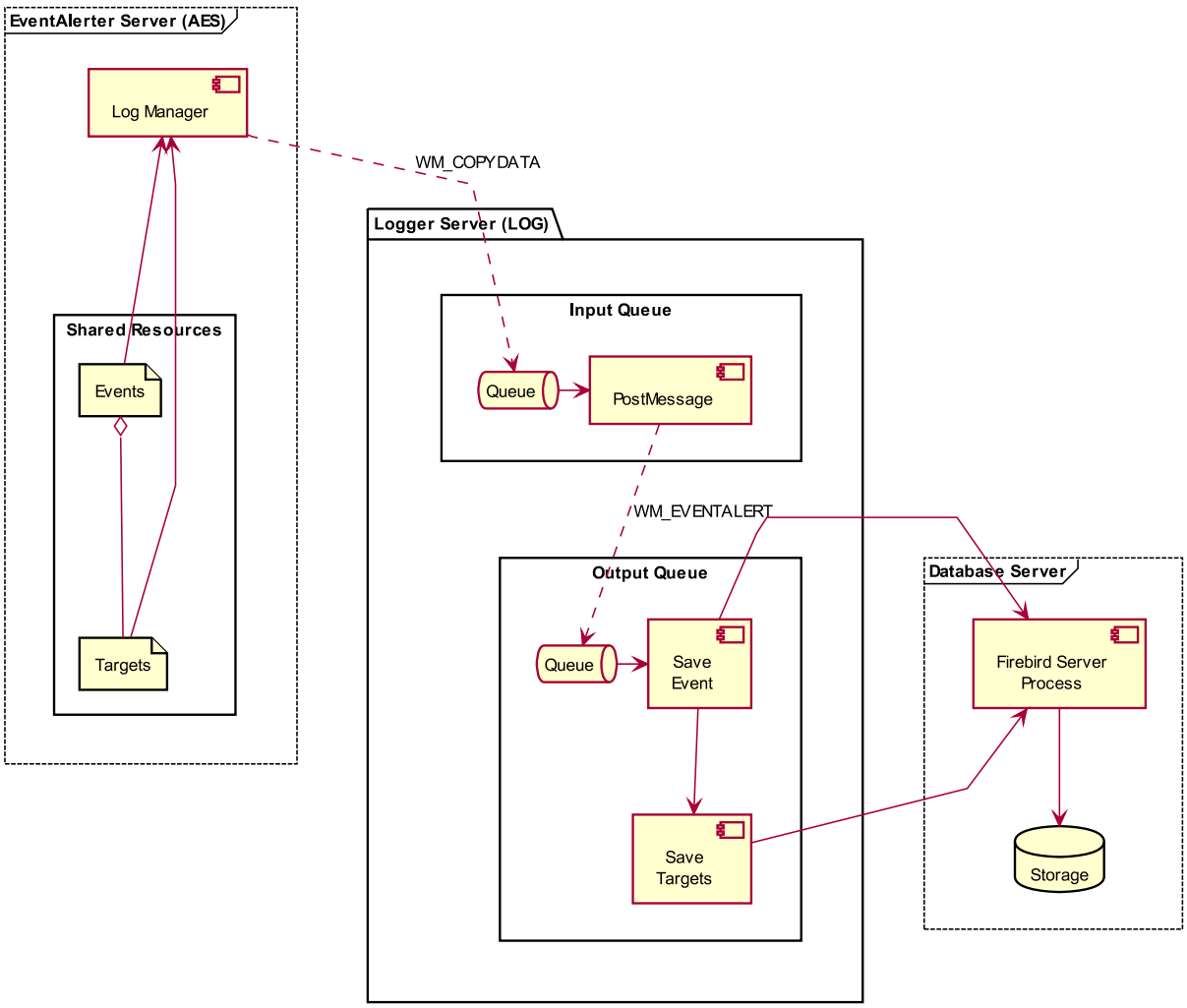
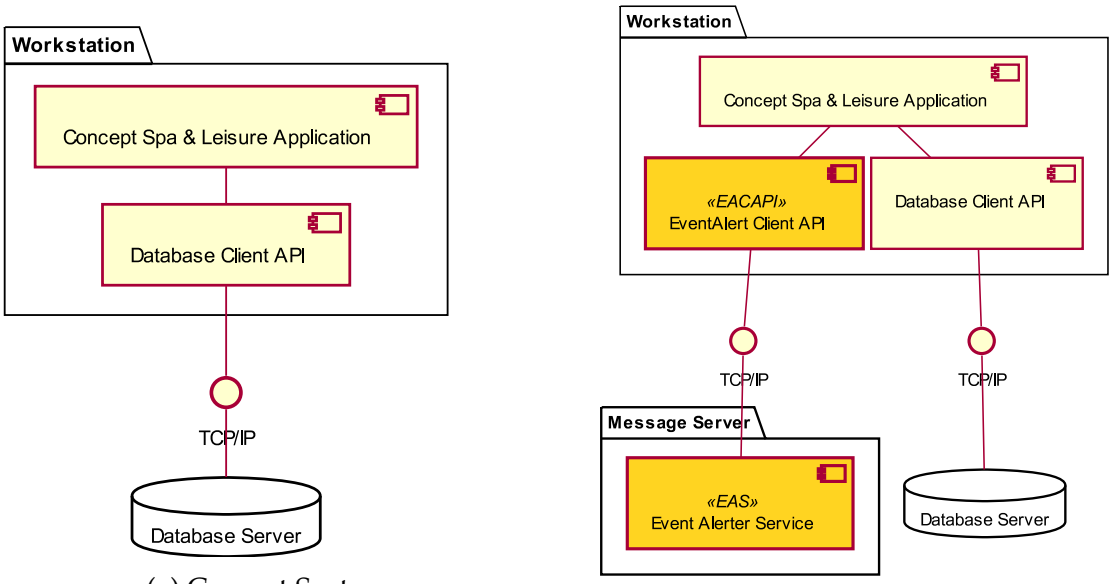


Figure 3.6: Logger process workflow.

the message is received on the main thread, the content of the message is copied to a local temporary buffer and the origin buffer data is released. Afterwards, the local buffer content is decoded to extract the message and target clients information. After the data extraction, the application prepares to save the information to the database. The first step it does, is to check if there is already an existing connection to the database. If no connection is found, the application attempts to connect to the database. After establishing the connection, the message data is saved into the database; followed by the identification of the target clients who received the message. Note that we allocate memory in one thread and then release it in another thread. This is a tricky way to allow the usage of the non-blocking *PostMessage()* function. The queue is implemented as FIFO (Stallings, 2018, ch. 5).

3.6 Client Application

The SPA application (client application or simply client), is composed of several components. One of those components is the *Database Client API* which provides the interface to communicate with the database server. Another component is the Message Client API, formally the Event Alert Client API (EACAPI), whose responsibility is to give access to the Message Broadcast Network (MBN). We will discuss in more detail the EACAPI in the following sections.



(a) Current System.

(b) New system with cache.

Figure 3.7: Client Application (SPA & Leisure), interaction of a workstation with database server and Message Server.

Figure 3.7 presents a brief view of the components of the system. However, it shows the fundamental logical networks. The connection with the database is handled by the *Database Client API*. The *Database Client API* consists of a set of classes that abstracts the database provider, allowing the application to connect to different database engines. The database holds information about the client business, which is mostly managing booking appointments, see Figure 3.7a. With the proposed system, the EACAPI provides a meaning to improve the application interaction with the database, Figure 3.7b. The EACAPI exposes the functions and classes to communicate with the Message Server. The EAS application and database server can re-

side on the same remote computer, but usually, they are installed on dedicated machines.

The MBF on client-side is basically composed of three main components, as shown in Figure 3.8. The main thread UI comprises the visible functionalities of the application. All user interactions with the application are realized in the Main Thread. Most of the business logic of the messages are executed here. The Ping Thread, ensures the connection between client and the EAS is not closed due to inactivity on the network, i.e., when there are no messages sent or received within a certain amount of time. The Read Thread is a vital component on the Message Broadcast Framework client-side, as it is the thread that listens for messages sent by the EAS. Messages are received in a dedicated thread to ensure that all messages are processed quickly, even if the application is blocked in another thread.

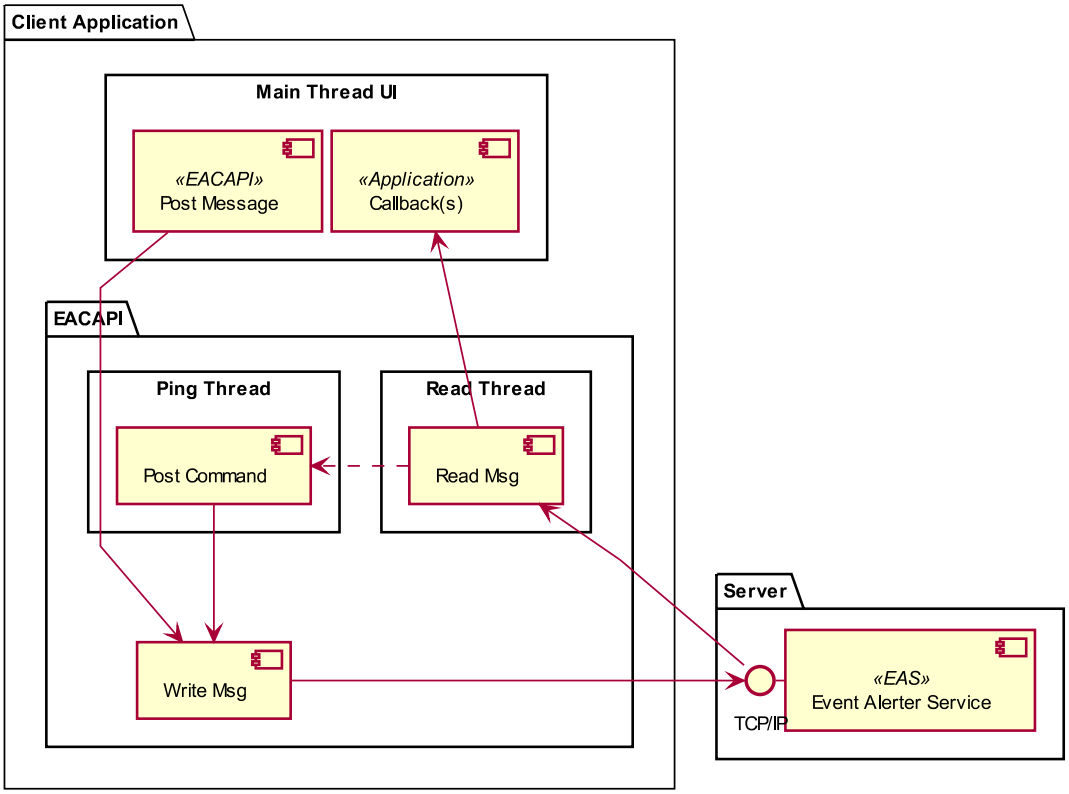


Figure 3.8: Overview of interaction of Event Alerter Server application (EAS) and Client components.

When a message is received by the EACAPI, the application is notified through a *Callback* function in the context of the listener thread. Exclusive access to shared

resources is guaranteed through the use of critical sections. A message can arrive while the main thread is performing an intensive task or blocked waiting for an I/O operation. Since the out-of-date flag is adjusted in the context of the listener thread, it implies that any thread depending on the message would know that the message was received, so it can do the necessary work to refresh data. The out-of-date flag can be updated several times without being accessed by processes on the main thread or other threads. The out-of-date flag will only be accessed by operations when the data depends on the message or messages controlled by the out-of-date flag.

We are referring to the Main Thread as the final part that has the business logic to manipulate the data received from the received messages. However, there are other situations where data is processed in secondary threads. For example, the SPA online search engine service performs all the operations in the context of the secondary threads. The Main Thread is the first thread of the application and has the responsibility of controlling the user interface.

The key actions related to the EACAPI are:

- Listening messages from EAS and take the necessary actions to set the out-of-date flag(s) to inform the application about new data available on the database server.
- Post messages, based on application business logic, to the EAS to inform the other connected applications that they have new data available on the database. The application sends a message and returns control to the user immediately. The sender does not require acknowledgement of the message by the other applications who receives it. If such acknowledgement should be required, it is a functionality to be added on the business logic layer of the application and not on the MBF.
- Send control messages to EAS to maintain the connection alive in situations where no messages are neither received or sent. The inactivity time duration is determined by the OS.

There are no special requirements for posting a message on the MBF. Clients fill in the type of message to be sent with the appropriate data and call a function of the

EACAPI to post the message. On the other hand, to receive messages, clients, must register the types of messages they wish to receive from the MBF.

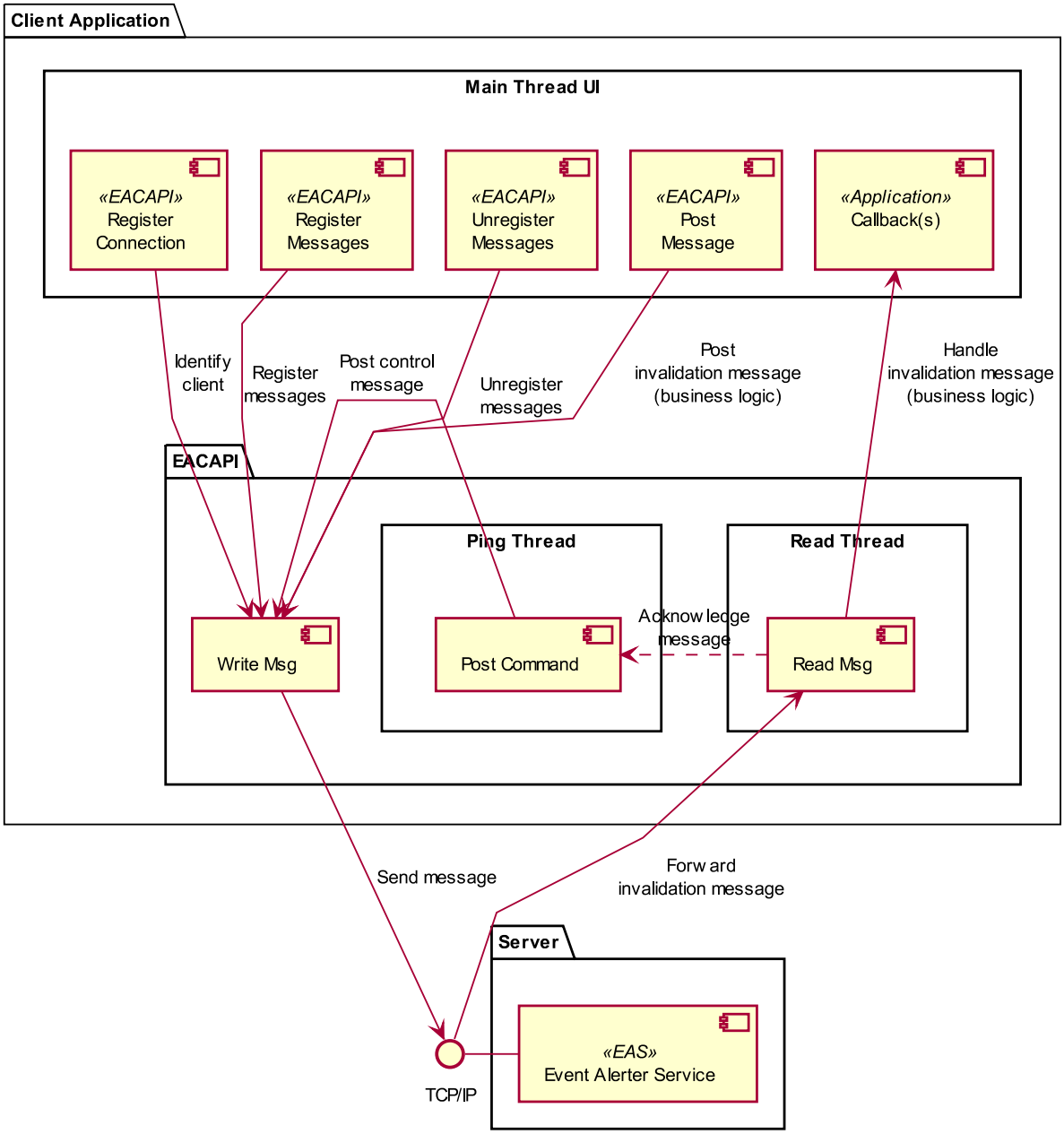


Figure 3.9: Messages flow between client components and server.

Figure 3.9 shows an overview of the interaction between components of the client and can be summarized as:

- **Register Connection** registers the application into the MBN. The operation informs the EAS that the application wants to receive messages. Each connection is identified by a unique id.

- **Register Messages** submits to the EAS, the specific messages that the application wants to receive.

- **Unregister Messages** submits to the EAS, the specific messages that the application no longer wants to receive.

- **Post Message** sends a message to the EAS to be forwarded to the applications that registered the message. (Including the sender of the message if applicable.)

- **Post Command** implements a queue of control messages to be sent to the EAS, as quickly as possible. The first control type message is the "keep alive" message to keep the connection with the server active. The second control type message is the acknowledge message sent back to the EAS, in a response to a request for confirmation of the received message.

- **Callback(s)** refer to the application endpoints that handle every message received from the EAS. Callbacks are associated with application logic connections (listeners). As a result, distinct callback functions (in different components of the application), can manipulate the same message received.

- **Write Msg** is responsible for sending data (securely) over the MBN to the EAS. This component is thread-safe, which means that the application can invoke the "Post Message" whenever required, without the need to explicit implement critical sections to protect access to shared resources.

- **Read Msg** receives messages sent by the EAS in a response of "Post Message" message from any application connected to the MBN. Soon as a message is received, it is sent to all the registered listeners. Finally, an acknowledge control message is queued, to be sent back to the server, if the received message requires feedback.

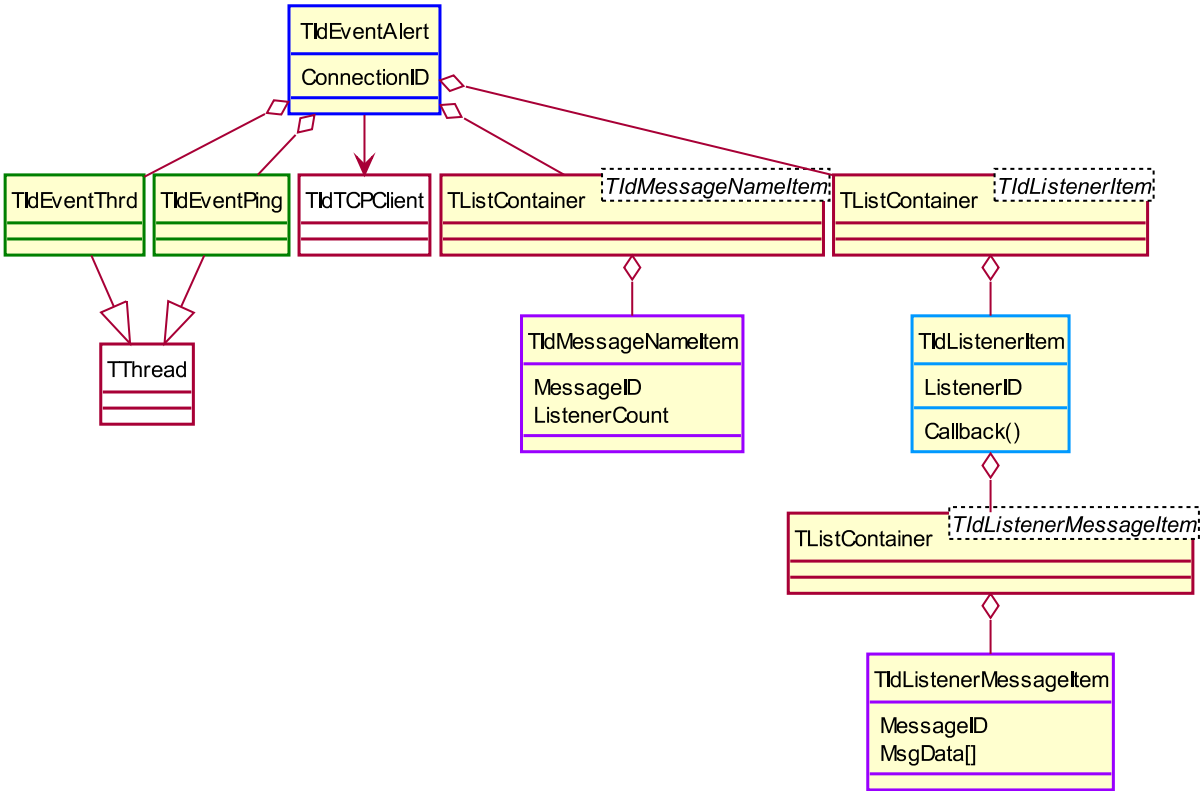


Figure 3.10: Client classes overview.

3.6.1 Broadcast messages client support classes

Figure 3.10 shows the classes used to interact with the Message Broadcast Framework on the client-side. These classes are:

- **TIdEventAlert** is the interface between client and the MBF. *TIdEventAlert* implements and exposes all the necessary properties and methods to connect to the EAS, register messages, register listeners, post messages and receive messages. The property *ConnectionID* uniquely identifies each client connected to the MBF. The EAS uses the *ConnectionID* to forward the received messages to the proper clients.

- **TIdEventPing** implements the logic to ensure the connection to the EAS is always active. It mainly implements a timer to post a recurring "ping" message directed only to the EAS.

- **TIdEventThrd** is responsible for receiving messages sent by the EAS, decoding their content and notifying the target listeners. Internally uses the *TIdTCPClient*

component to handle the TCP/IP communications.

- **TIdTCPClient** is responsible for the TCP/IP communications. This component and its ancestors implement the input/output abstraction mechanism used for connecting, disconnecting, sending, and receiving data through socket connections, (Indy Pit Crew, 2020). It is available in the Embarcadero RAD Studio development tool (Embarcadero, 2019) and is the counterpart of the *TIdTCPServer* component.

- **TIdListenerItem** represents the listener component. This class is the bridge between the application UI and the MBF. Each listener contains a set of registered message types and a callback function. The callback function is invoked by the *TIdEventThrd* class when a message is received from the MBF. The callback function implements the business logic operations associated with the message type within the context of the listener instance, representing the message endpoint.

- **TListContainer<TIdListenerItem>** contains the list of all active listeners registered in the EACAPI. When a message is received from the EAS, the list is scanned to find the listeners that should get the message.

- **TIdMessageNameItem** stores one message type definition (identified by a GUID) and the number of instances of the listeners that registered it. When a listener registers a new message (not yet registered by any other listener), a new *TIdMessageNameItem* record is created with the *ListenerCount* variable set to 1. On the other hand, if the message has already been registered by other listeners, the *ListenerCount* will be incremented by 1. In the same way, when a listener unregisters a message, the *ListenerCount* is decremented by 1. When the *ListenerCount* reaches a value of 0, it implies that no more listeners are referring to the message. In this case, the message definition record is deleted.

- **TListContainer<TIdMessageNameItem>** contains the list of unique message types registered by the local listeners. There is only one global list containing the registered messages in the client. A message type in the list implies that at least one listener has registered it.

- **TIdListenerMessageItem** holds one message type registered by one listener. When a message is received from the EAS, the *MsgData* variable is populated with the message data. This message data is then made available to the application UI through the *Callback* function.

- **TListContainer<TIdListenerMessageItem>** contains the list of all message types registered by a listener. Each listener maintains an individual list of its registered message types. When a message is received from the EAS, the list is scanned to find if the listener that owns the list should get the message.

3.6.2 Application UI workflow

The client application integrates the Message Broadcast Framework as shown in Figure 3.11. The activity diagram shows the application workflow. At application startup, the main thread *Main Thread UI* is created. The user interface thread is used to create visual controls, handle user input and respond to user events independently of threads executing other portions of the application, see (Microsoft, 2019b). The *TIdEventAlert* component, from the EACAPI, is also initiated at startup time. At this point, only the main thread exists. When the connection to the EAS is made through *TIdTCPClient::Connect()*, two threads are created. The first thread, *TIdEventThrd* is responsible for listening and forwarding received messages to the appropriated objects. The second thread, *TIdEventPing* ensures that the TCP/IP connection to the EAS remains active.

The main thread not only handle the application visual controls and user operations but also post messages ("Post Message") to the EAS in response to the "User Actions", to notify other applications that they will need to update their local data. The other two threads are managed by *TIdEventAlert* class and do not depend on user interaction to work.

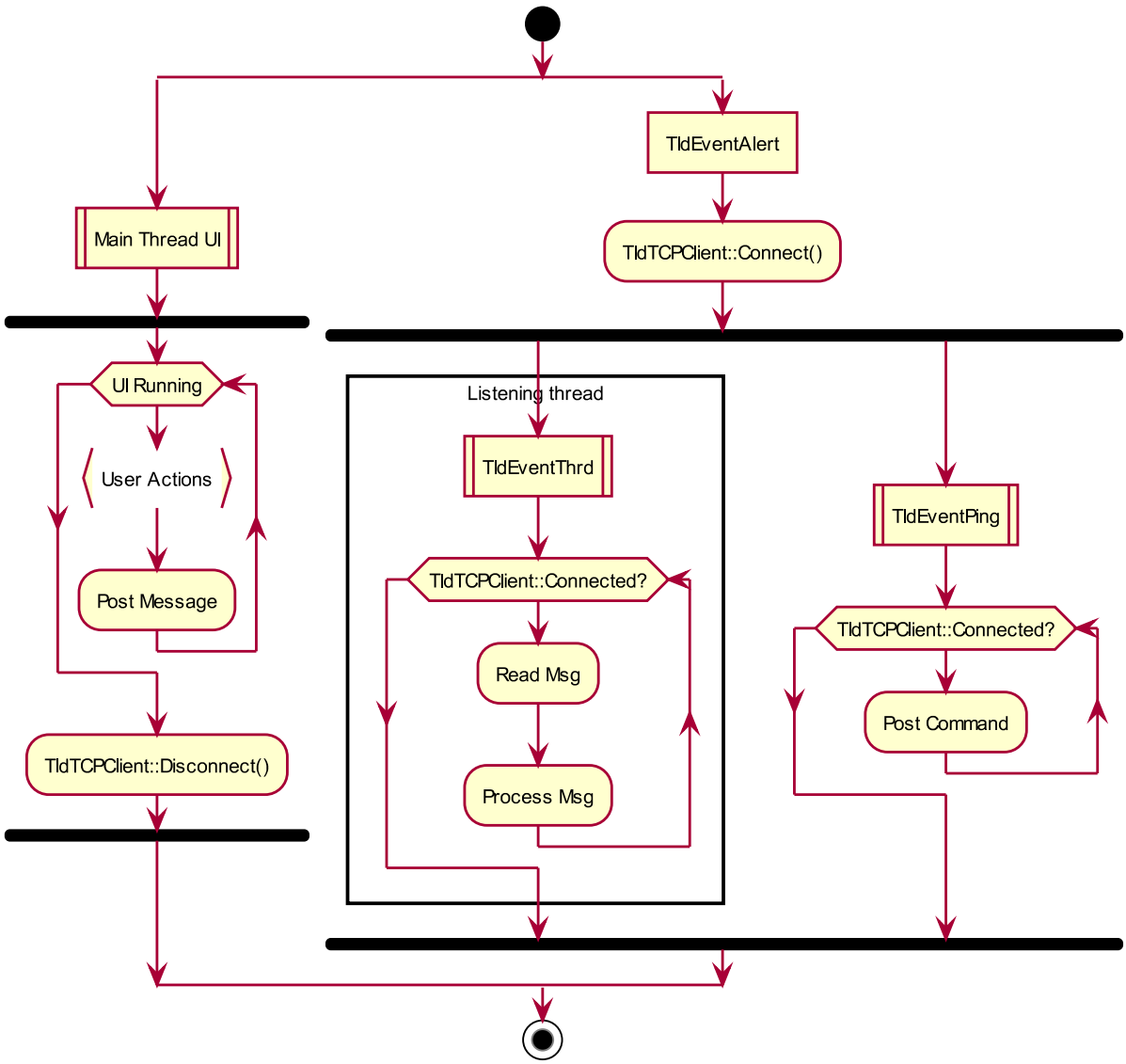


Figure 3.11: Client application messages workflow.

3.6.3 Client Identification

The client application opens a single TCP connection to the EAS. The connection is uniquely identified by a *ConnectionID* string using a unique identifier (GUID). The EAS uses the *ConnectionID* to track the messages registered by each client. In addition to the public *ConnectionID*, the application manages a private collection of listeners also identified using a GUID. All the messages registered by the local listeners are registered on the server using the *ConnectionID*.

Figure 3.12 shows the link between client applications and the Event Alerter Server application. The server identifies the connected applications using the *ConnectionID*,

that is, “Connection 1” and “Connection 2” instead of the listeners, “Listener 1”, . . . , “Listener 4”, which are internal to clients.

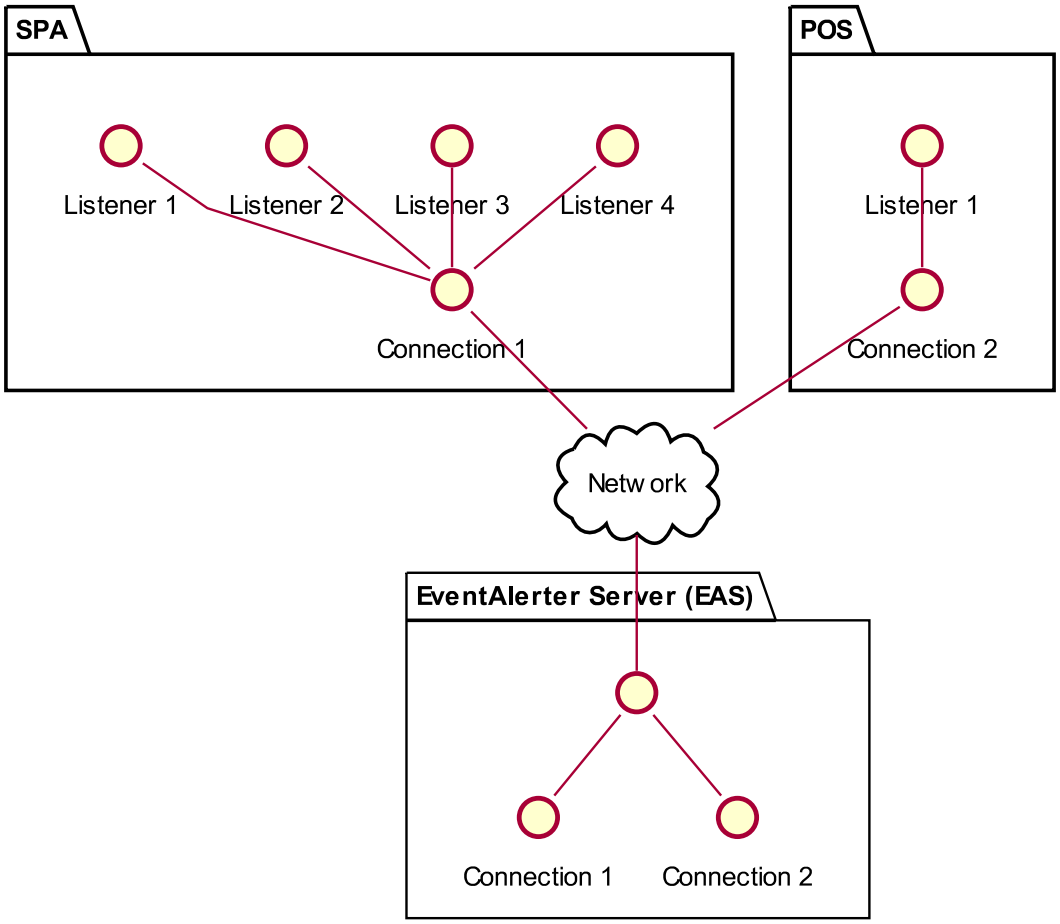


Figure 3.12: Application identification.

When a client sends a message to the MBS, the EAS searches in its internal *Messages* and *Subscribers* structures to find which clients to forward the received message. Similarly, when clients receive the message, they search in their *TListContainer<TIdListenerItem>* and *TListContainer<TIdListenerMessageItem>* internal structures to identify the target listeners who should handle the message.

3.6.4 Client Listeners

A listener is an object of class *TIdListenerItem* and allows the client to send and receive broadcast messages from the MBN. In Section 4.2 we materialize the concept of the listeners in terms of the classes that will implement the local cache.

Each listener maintains an individual list of the registered messages, but they share common messages. When a listener adds a message type to its list, it checks if that message has already been registered by another listener. If no client has yet registered the message, then the message is registered on the server. On the other hand, if the message was already registered by another listener, no request is made to the server. Hence no duplicated messages are registered using the same *ConnectionID*.

Figure 3.13 illustrates connections and listeners along with the registered messages. The example shows 4 forms: *Configuration*, *Booking Creation*, *Booking Confirm* & *Check-In* and *Agenda*. Each form is associated with a listener to allow send and receive messages, according to the application business logic. "Send Messages" refers to messages that can be sent by listeners. "Registered Messages" refers to messages registered by listeners. As can be seen, some messages types are specific to individual listeners while others are shared by several listeners. For example, the message "ComplexChanged" is registered in the *Booking Creation* and *Agenda* forms, while the message "BookingCreated" is only registered by the *Agenda* form.

Figure 3.13 also shows that the messages registered in a form are not necessarily related to the messages sent. For example, the *Booking Creation* form sends the messages "BookingCreated" and "BookingUpdated" but does not register them. The reason is that the form does not rely on these messages to know when bookings are created or updated, instead, it uses the data from the database directly. On the other hand, the *Agenda* form does not send any message but registers the messages "BookingCreated" and "BookingUpdated".

3.6.5 Listeners in Action

We outline below, based on Figure 3.13, how the messages flow within SPA and POS applications and MBN. We focus on the messages exchanged between applications and the MBN and not on the data transferred between applications and the database server. The use case describes (1) activity booking creation; (2) activity check-in; and (3) activity check-out.

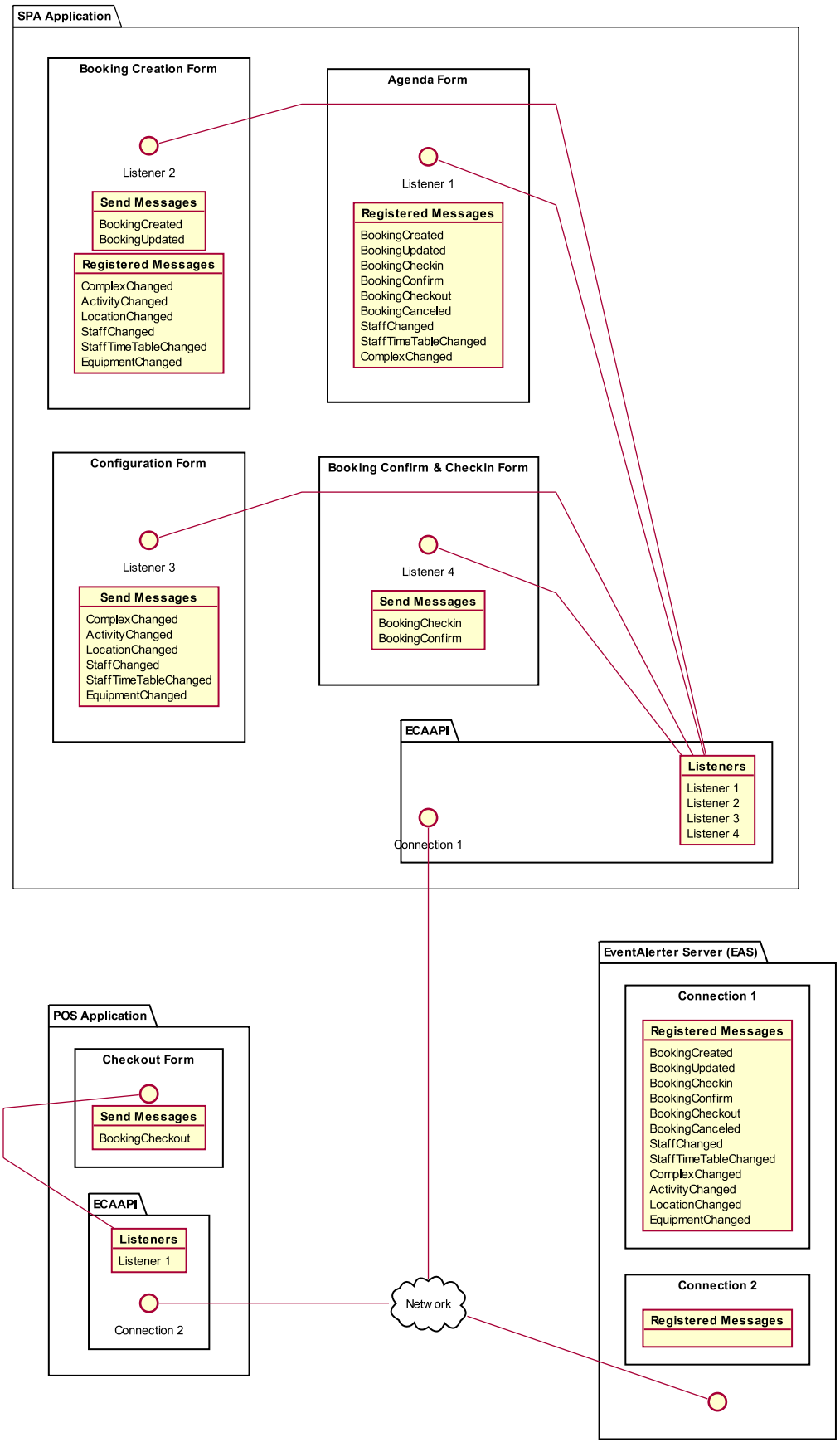


Figure 3.13: Client listeners within SPA and POS applications.

1. The operator starts the SPA and POS applications. Both applications initialize their EACAPI instance, which connects to the MBN using "Connection 1" and "Connection 2" respectively. Next, the operator opens the *Agenda* form. The *Agenda* form creates the local "Listener 1" which registers the messages "BookingCreated" ... "ComplexChanged" into the EAS.

   Then, the operator opens the *Booking Creation* form. The *Booking Creation* form creates the local "Listener 2" and registers the messages "ComplexChanged" ... "EquipmentChanged". The EAS receives the messages registration request and adds the messages to the "Connection 1" messages list.

   From there, the operator sets the customer's preferences and creates the reservation. At the end of the creation process, the application prepares the message "BookingCreated" and sends it through the "Listener 2", which calls the EACAPI to send the message to the MBS using "Connection 1" as the message's sender.

   In the meantime, the EAS receives the message "BookingCreated" and locates it in the *Messages* structure. Next, it searches in the *Subscribers* structure to find which clients have registered the message. In this case, it finds that the message "BookingCreated" was registered by the "Connection 1". Therefore, the server forwards the message to the client owning the "Connection 1", i.e., the SPA application.

   The SPA EACAPI receives the forwarded message from the EAS. Next, it walks on the list of listeners (see *TListContainer<TIdListenerItem>* above) and for each listener record, it searches in its list of registered messages (see *TListContainer<TIdListenerMessageItem>* above) for the message type "BookingCreated" and finds that the "Listener 1" has registered that message type. Therefore, it forwards the message to the "Listener 1" by calling its callback function.

   The "Listener 1" callback function, extracts the message content and tries to locate in the internal (cache) records, for a record matching the message content. In this

case, it does not found a record because the message refers to a new booking. Consequently, it adds a new record, flagged as out-of-date, into the cached records list. The out-of-date flag indicates that the next time this record is required, the database must be contacted to obtain the most recent data matching the cached record.

The *Agenda* form regains focus after the operator closed the *Booking Creation* form. Now the agenda must update the list of bookings.

Before requesting data from the database, the internal list containing the cached booking records is searched and it finds that for the current agenda visible filters, there exists a cached record flagged as out-of-date. Therefore the application submits a query to the database server to obtain the most recent record. After, it clears the out-of-date flag from the local record. Finally, the newly created booking is added to the visual grid control.

2. On the booking day, the operator opens the *Booking Confirm & Check-In* form. The *Booking Confirm & Check-In* form creates the local "Listener 4" which does not register any message. From there, the operator changes the booking status to check-in. After the change, the "Listener 4" sends the message "BookingCheckin" with the relevant booking information to the EAS. The EAS receives the message and finds that the message shall be forward to the "Connection 1".

   The SPA EACAPI receives the forwarded message from the EAS. Next, it walks on the list of listeners and finds that "Listener 1" has registered that message type. Therefore, it forwards the message to the "Listener 1" by calling its callback function.

   The "Listener 1" callback function, extracts the message content and try to locate in the internal records (cache), for a record matching the message content. In this case, it founds a record. Consequently, it sets the record as out-of-date. (The out-of-date flag indicates that the next time this record is required, the database must be contacted to obtain the most recent data matching the cached record.)

The *Agenda* form regains focus after the operator closed the *Booking Confirm & Check-In* form. Now the agenda must update the list of bookings.

Before requesting data from the database, the internal list containing the cached booking records is searched and it finds that for the current agenda visible filters, there exists a cached record flagged as out-of-date. Therefore the application submits a query to the database server to obtain the most recent record. After it clears the out-of-date flag from the local record. Finally, the checked-in booking is updated on the visual grid control.

3. Once the activity completes, the operator initiates the check-out. The SPA application invokes the POS by sending a message using the WinAPI *SendMessage()* containing the necessary information to the POS complete check-out process. When POS receives the message sent by SPA, it opens the *Checkout* form. The *Checkout* form creates the local "Listener 1" which does not register any message. From there, the operator completes the checkout operation. After the completion, the "Listener 1" sends the message "BookingCheckout" with the relevant booking information to the EAS. The EAS receives the message and finds that the message shall be forward to the "Connection 1".

   The SPA EACAPI receives the forwarded message "BookingCheckout" from the EAS. Next, it walks on the list of listeners and finds that "Listener 1" has registered the message "BookingCheckout". Therefore, it forwards the message to the "Listener 1" by calling its callback function.

   The "Listener 1" callback function, extracts the message content, locates the matching internal record and sets it as out-of-date.

   The *Agenda* form regains control after the operator closes the POS *Checkout* form. Now the agenda must update the list of bookings.

   Once again, before requesting data from the database, its searches in local memory and finds the matching record flagged as out-of-date. Therefore, it submits a query to the database server to obtain the most recent data and clears the out-of-date flag

from the local record. Finally, the checked-out booking is updated on the visual grid control.

The key point to retain from the above steps is that, because of the use of the MBF, SPA requested only once the full list of records matching the *Agenda* form visible filters. The following requests to the database server, requested only the record matching the changed booking, resulting in a faster update of data in the *Agenda* form visual grid control.

3.7 Messages

3.7.1 Messages Encoding

Messages are transmitted between client and server in System Data Format (SDF) using the newline character `'\n'` as message terminator. The SDF encoding is similar to Comma Separated Values (CSV) text file format and is used extensively by the VCL classes (for example, the *TStringList* class) to represent a list of strings as by a unique string using commas to separate the individual strings. Any string in the list that includes spaces, commas or quotes will be contained in double quotes, and any double quotes in a string will be repeated. For example, if the list contains the following strings[3]:

```
Stri,ng␣1
Stri"ng␣2
String␣3
String4
```

The encoded raw string will be:

```
"Stri,ng␣1","Stri""ng␣2","String␣3",String4
```

Spaces and commas that are not contained within double quote marks are delimiters. Two commas next to each other will indicate an empty string, but spaces that appear next to another delimiter are ignored (I. C. Embarcadero, 2009).

[3]One string per item in the linked list.

Like a list of strings which be represented by a single comma separated string, any serializable[4] object, can also be represented in the same fashion. Consider another example using strings with *key=value*:

```
Key1=Value1
Key2=Value␣2
Key3=Value,3
Key4=Value"4
```

The result encoded raw string[5]:

```
Key1=Value1,"Key2=Value␣2","Key3=Value,3","Key4=Value""4"
```

The last example introduces the foundations on how *TListContainer<typename>* objects serialize their content. We could have chosen to use another format like *XML* or *JSON*, but we believe that the SDF is more appropriated due to its simplicity and because it is the serialization format used by the *TStringList* class.

Typographic conventions:

The symbols <>{|}[]↩␣ are not part of the message. They are used to show how the syntax is used to build the messages. Angle brackets <> indicates that the text inside them is inserted at application execution time with the proper text; curly brackets {|} can read as "take one value from set of values", where each value of the set is separated by the vertical bar |; square brackets [] denotes optional text to be included in the message depending on its type; ↩ indicates a long text split across multiple lines because it does not fit in the page text width; and ␣ represents the space character.

3.7.2 Messages Format

The messages used in the MBF use the following syntax:

[4]Binary object expressed as a chain of characters.

[5]Note that the pair `Key1=Value1` is not enclosed in double quotes because it does not contain spaces, commas or double quotes characters.

```
ID=<Id>,C={R|U|P}[,E=<MessageID1>[,<MessageID2>[,...,<MessageIDN>]]]↩
[,D=<MsgData>]
```

where:

- ID specifies the client connection unique identifier, which is by default filled with a Globally Unique Identifier (GUID) string.
- C refers to the message command. It can have one of the following values: R for register messages; P for post messages; and U for unregister messages.
- E specifies the name of the message to be sent. Depending on the message command type, it can have multiple values separated by commas.
- D contains the optional message data to be included on the message. The <MsgData> part is the information specific to each message type (<MessageID>) and it is encoded in SDF format before insertion into the message frame.

3.7.3 RegisterMessages

The general message syntax to register messages is:

```
ID=<Id>,C=R,E=<MessageID1>[,<MessageID2>[,...,<MessageIDN>]]
```

For example, to register one single message, the following message is used:

```
ID={812DF399-A252-4293-8A12-4EDE2F58F660},C=R,E=StaffTimeCreated
```

To register multiple messages, the message changes to:

```
ID={812DF399-A252-4293-8A12-4EDE2F58F660},C=R,↩
"E=StaffTimeCreated,StaffTimeUpdated,StaffTimeDeleted"
```

3.7.4 UnregisterMessages

The general message syntax to unregister messages is:

```
ID=<Id>,C=U[,E=<MessageID1>[,<MessageID2>[,...,<MessageIDN>]]]
```

For example, to unregister all messages, the following message is used:

```
ID={FABEF49D-C565-42E5-9984-6C0EAEDAAE52},C=U
```

To unregister one single message, the message changes to:

```
ID={FABEF49D-C565-42E5-9984-6C0EAEDAAE52},C=U,E=StaffTimeCreated
```

To unregister multiple messages, the message changes to:

```
ID={FABEF49D-C565-42E5-9984-6C0EAEDAAE52},C=U,↩
"E=StaffTimeCreated,StaffTimeUpdated,StaffTimeDeleted"
```

3.7.5 PostMessage

The general message syntax to send notification (invalidation) messages is:

```
ID=<Id>,C=P,E=<MessageID1>[,<MessageID2>[,...,<MessageIDN>]]]↩
[,D=<MsgData>]
```

For example to send a message with no data, the following message is used:

```
ID={426DF813-25AA-4270-A3EA-51844E3FE416},C=P,E=ClearCache
```

To send a message with data, the following message is used:

```
ID={EA86D788-95D1-49F7-939C-4CC8316E297C},C=P,E=FacilUpdate,↩
"D=RecID=456,Code=AC30,""Name=Acupuncture 30 min"""
```

Another example:

```
ID={67B11443-6B20-41A7-8C41-17EDE5EF5B5E},C=P,E=StaffTimeCreated,↩
"D=""RecID=27239,StaffID=21,PeriodNo=13707,StartDate=43539,↩
EndDate=44196,ComplexID=3"""
```

3.8 Security

The encoded messages have security information to certify the source, destination, and the integrity of the data. This information is never exposed because it is confidential. Besides, the focus of the current work is to show how to create and use a caching platform to optimize one particular application.

Everything is vague to a degree you do not realize till you have tried to make it precise.

Bertrand Russell

4

Application Integration

In this chapter, we describe the integration of the Message Broadcast Framework with the SPA application. We start by defining the architecture using the search engine as a use case. We enumerate the class templates used to connect to the Event Alert Client API, linking them to the listener components introduced in the Chapter 3. After that, we move on to the implementation model, showing how cache classes communicate with the Message Broadcast Framework. Finally, we present the integration of the class templates using a concrete example of how the staff timetable information is cached in the local memory and how it is accessed on both the old and the new system with cache.

4.1 Overview

The Message Broadcast Framework integration was designed to not require significant changes to the application codebase or directly interfere with existing functionality. In fact, our proposed solution was built to run as a passive decoupled component to allow easily tuning without changing the existing application's behaviour.

The messaging system was integrated into the reservation creation process, specifically in the search engine component. We opted to start with the search engine operation, shown in Section 2.4, because (a) the search operation involves a significant number of queries submitted to the database in a short period, (b) with some of them returning the same data in successive calls (data with low change frequency in database), and (c) the storage of data in local memory from the distinct queries can share the storage mechanism, reducing the initial integration development time.

With a considerable reduction of database queries at an early stage of integration, we would have a better perspective of the proposed solution viability.

4.2 Architecture

The integration architecture model relies on the creation of generic and specific classes. Generic classes provide the functionalities to communicate with the EACAPI whereas the specific classes implement the details of the business logic to manage the cached data in memory. The specific classes are self-governing and independent of each other, and we refer to them as *Cache Classes*. *Cache classes* are the materialization of the concept of listeners, introduced in Chapter 3, Section 3.6.

Figure 4.1 shows the classes created to integrate the MBF in the application's search engine. Each class holds a specific type of information to be cached in the local memory. The EACAPI is responsible to maintain the cache object instances up to date by notifying them when data becomes out-of-date.

The use of class templates simplifies codification of the necessary methods to support the implementation of the final classes dedicated to each set of data to be

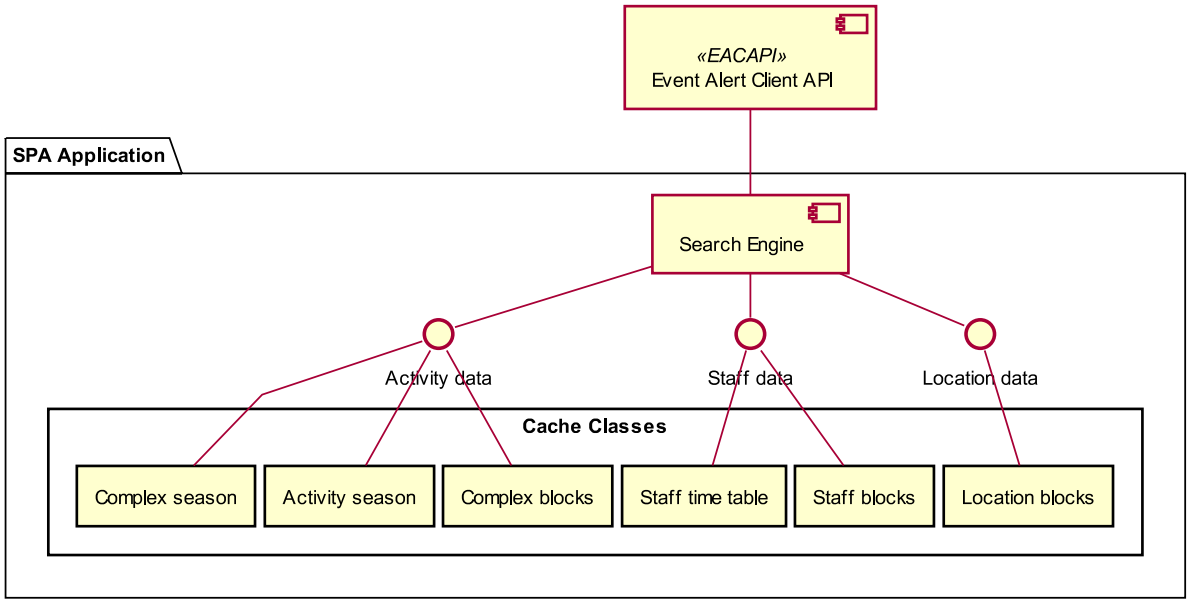


Figure 4.1: The cached data integration model used in the search engine, with cache classes showing the type of information they keep in local memory.

cached. Figure 4.2 exhibits the skeleton classes *TListContainer<typename>*, *TLockListContainer<typename>* and *TEventLockListContainer<typename>* used to create "cacheable" classes.

- **TListContainer<typename>** provides functionalities to maintain a list of indexed records with data in local memory. Each record is a dictionary entry, composed by the pair <`key;value`>, where the <`key`> is a string that uniquely identifies the record and the <`value`> represents a specialized structure to hold the information for each cache object.

- **TLockListContainer<typename>** introduces support for critical sections through the member functions *LockList()* and *UnlockList()*. All manipulation of data between *LockList()* and *UnlockList()* calls are thread safe. This is crucial not only because there are multiple threads running in application UI, but also because EACAPI is asynchronous. Messages are received and forwarded (via callbacks) to the application UI for processing in a dedicated thread.

- **TEventLockListContainer<typename>** makes the bridge between the MBF and the specialized classes responsible for manage data in memory. The variable

member *ListenerID*, identifies the virtual connection to the EAS as disposed in Subsection 3.6.4. All descendant classes of *TEventLockList* fall in the category of the specific classes, i.e., they are subject to the business rules established in the application for the data they hold.

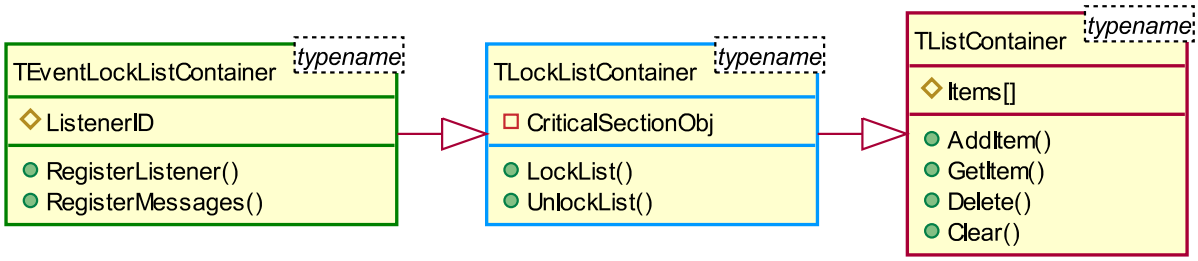


Figure 4.2: Integration class templates hierarchy.

The creation of a class with cache requires: discovering the minimum useful information to be kept in memory; finding the right keys to identify each record; and last, the definition of the messages to control the behaviour of the data (when data becomes out-of-date).

Figure 4.3 demonstrates the layout used for the classes. The *TMyList* holds a list of *TMyItem* records, whose content is stored in *Variable1, . . . , VariableN*. In other words, *TMyList* defines the behaviour of *TMyItem*. The *TMyItem* is a structure which can be used also by other functions or classes in the application. This separation is very important because any existing data structure in the application can be used in the messaging-based data management classes, by simply defining an offspring class of *TEventLockListContainer<typename>* (where *<typename>* is the referred structure). The ability to cache any existing structure of data, gives us the ability to progressively and selectively cache data in local memory with minimal modifications in the application. It is important to note that data structures to be cached can contain variables with dynamically allocated memory. For this reason, the memory occupied by each record in the cache classes may vary.

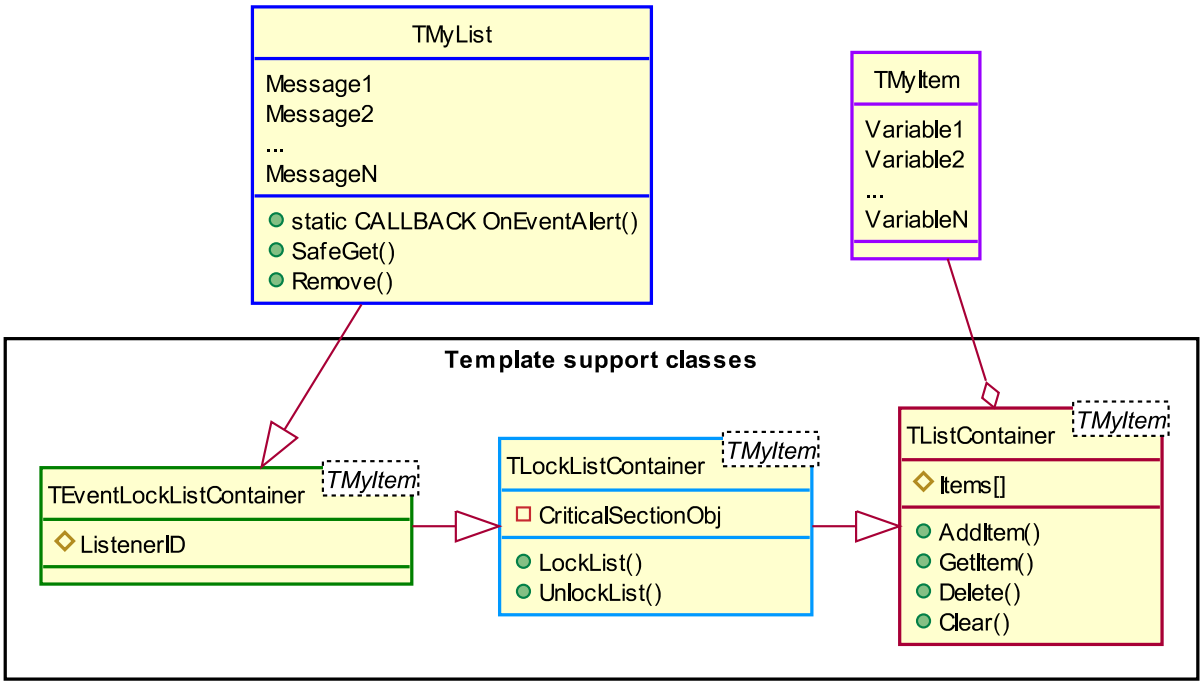


Figure 4.3: Example of class for storing data in memory.
All classes built with cache support, follow this schema to take advantage of the facilities provided by the EACAPI.

4.3 Implementation

Figure 4.4 shows how cache classes manage the data they hold in memory following and extending the pattern presented in Figure 2.10, Chapter 2. At that time, the concepts *Set out-of-date*, *Reset out-of-date* and *Is out-of-date*, were briefly introduced to give the idea behind detection and usage of cached data. Here we will concretize these concepts for use in cache classes:

Set out-of-date denotes that the data stored local in local memory is no more synchronized with database, hence needs to be refreshed before next use. Concretely, the term *Set out-of-date* is the removal of the record (associated with the message) from the in-memory *items[]* list.

Reset out-of-date is a way to identify that the local data is synchronized with the database. In short, the data is up to date if it exists in the in-memory *items[]* list, upon the call of function *SafeGet()*. Therefore, the designation *Reset out-of-date*, is the action of database data gathering and list insertion.

Is out-of-date is the method to determinate if a specific record stored in *items[]* list is up to date. Formally, it is the search in the *items[]* list for the record where *key* matches the search value. In conclusion, if the required data exists in the *items[]* list (out-of-date is false), it is ready to be used; otherwise (out-of-date is true), the data must be obtained from database and inserted into the *items[]* list in order to be ready for use.

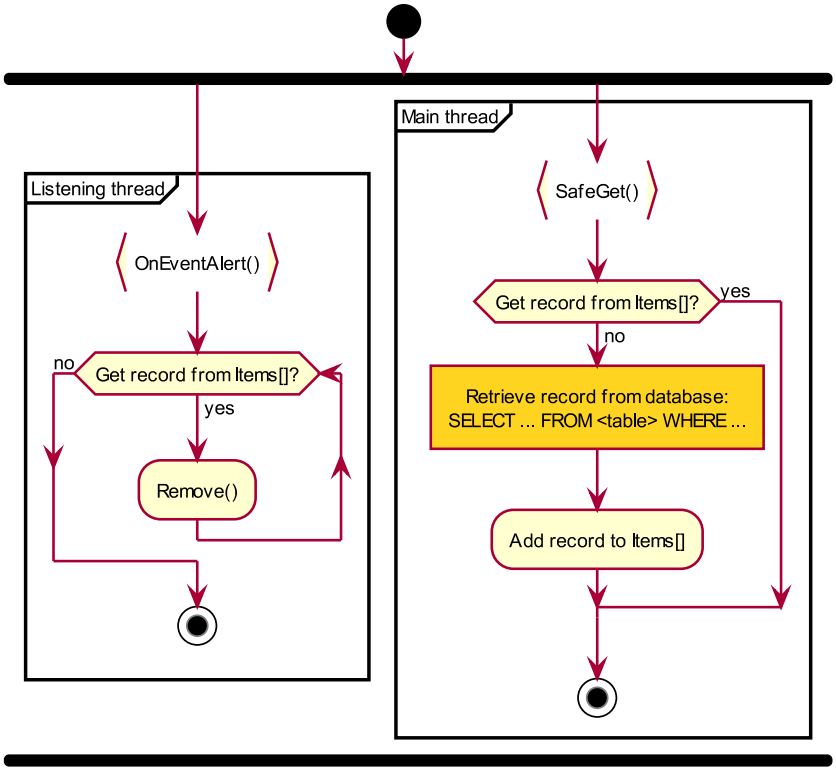


Figure 4.4: Data management within a cache class.

In Figure 4.4, we can see that there are two independent flows. The left flow runs in the thread context of the EACAPI in a listening state (passive) and the right flow runs on the main thread UI (active). On the *Listening thread*, the data associated to the received message is removed from the *items[]* list (corresponds to *Set out-of-date*); on the *Main thread* the data is fetched from the database only if it is not in the in-memory *items[]* list (corresponds to *Is out-of-date* and *Reset out-of-date*).

- **OnEventalert()** function callback is called by the EACAPI in a response to a received message registered by the class. Here the message is used to search for

matching record in *items[]* list, using *key* as input parameter. The record found is deleted from the list. No other resources are accessed from the callback function, for example database and interfaces, because all callback functions connected to the EACAPI, are executed on the same thread context. The actions performed in this function corresponds to the *Set out-of-date*.

- **SafeGet()** is called by external functions in the context of the main thread UI, in particular from the search engine, requiring access to the data stored in the *items[]* list. As the name suggests, the function is thread safe. The *LockList()* and *UnlockList()* locks and unlocks respectively, the access to the *items[]* list shared by *SafeGet()* and *OnEventAlert()* functions. The reason is because these functions are executed in distinct threads.

  The first action performed inside the function, excluding the lock mechanism, is to Look into the *items[]* list for a record that match the input parameters passed to the function. If a record is found, it is returned to the caller function. On the contrary, if no record found, then a query is submitted to the database inquiring for a record matching the input parameters. In case of success, the data retrieved from database is placed into the *items[]* list. In case of no data returned, i.e., no matching record found on database, a record with default values (each class has its own set of default values according the data structure they implement) is added into the *items[]* list. The addition of a record into the *items[]* list corresponds to the *Reset out-of-date*.

4.4 Integration

The integration stage consists in the utilization of the facilities provided by the MBF, to maintain data stored locally through the cache classes built for the effect. From the list of validations performed in the booking creation process, we chose a subset of validations to integrate within the MBF:

- Related with **activity** we decide to cache information about *complex blocks*, *complex*

season and *activity season*. This information usually changes when setting up the application and almost always outside of business hours.

- From **staff**, we chose *staff timetable* and *staff blocks*. At the present time, *staff timetable* and *staff blocks* have a considerable impact on performance due to the number of requests performed. As a rule, the staff schedule is done once per month. Afterwards, it may be adjusted due to several factors, like staff illness, staff operations and scheduled exchange between staff members. As a result, a peak activity may occur at the end of each month during the preparation of next month's schedule. However, even though performance degradation may occur (due to the number of messages circulating on the MBN in response to time change messages), it will have a limited impact over the month.

- Finally, from **location**, our choice fell on the *location blocks*, because changes are unlike to happen daily or even monthly.

The criteria for selecting these rules to apply the data cache were: the low rate of data change in the database, and the number of queries that requested the same data. The former refers to the frequency of *insert*, *update*, and *delete* operations in the database. The latter refers to obtaining the same data in a short period given the same input parameters. For example, the staff timetable is usually set once a month or once a week; and locations are generally set up during the initial installation and are rarely changed. In both examples, the number of modifications is low, while the number of queries is high. Although the size of the data transmitted over the network is relatively small, it is not necessary to consult the database every time the availability needs to be known. (Not to mention that these queries return, in most cases, the same results.)

We will demonstrate the cache mechanism, by taking the *staff timetable* as an example, and show how data is cached and accessed from the search engine in comparison with the old system without cache.

4.4.1 Definition of cache class TStaffTimeList

As we state in Section 4.2, every cache implementation requires two classes. In this case the *TStaffTimeList* implements the logic to manage the *TStaffTimeItem* data. The class *TStaffTimeItem* was already in use by the search engine, while *TStaffTimeList* was created to make use of the functionalities provided by the MBF.

Figure 4.5 illustrates the structure of the integration classes. The *TStaffTimeItem* class contains variables to identify a record in the staff timetable. As can be seen, the information is related to staff working time, identifying when staff in on schedule. The *TStaffTimeList* aggregates the features provided by the class templates *TEventLockListContainer<typename>*, *TLockListContainer<typename>*, and *TListContainer<typename>*. The *ListenerID* variable provides the link to connect the class *TStaffTimeList* object instance to the EACAPI, which uses its *ConnectionID* to connect with the MBF. The messages "StaffTimeCreated", "StaffTimeUpdated", "StaffTimeDeleted", and "ClearCache" are registered using the *ListenerID* to identify the callback function *OnEventAlert()* to be called when one of these messages is received by the EACAPI.

The messages registered have the following business logic:

- **StaffTimeCreated**: Sent by the application after user has created a new staff timetable definition in the database.
- **StaffTimeUpdated**: Sent by the application after user has modified an entry of the staff timetable.
- **StaffTimeDeleted**: Sent by application after user has deleted a staff timetable record from database.
- **ClearCache**: As opposite to the above messages, which had in mind a subset of subscribers, this message was designed to be registered by all cache classes, to purge data from memory.

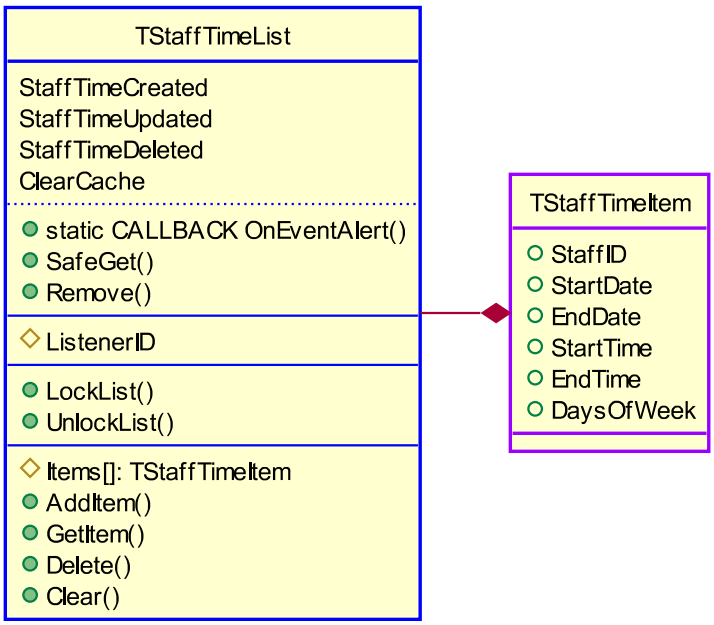


Figure 4.5: Example of the staff timetable classes used on integration.
Each record of *Items[]* contains a *TStaffTimeItem* object identified by the search key composed by *StaffID;StartDate;EndDate;StartTime;EndTime*.

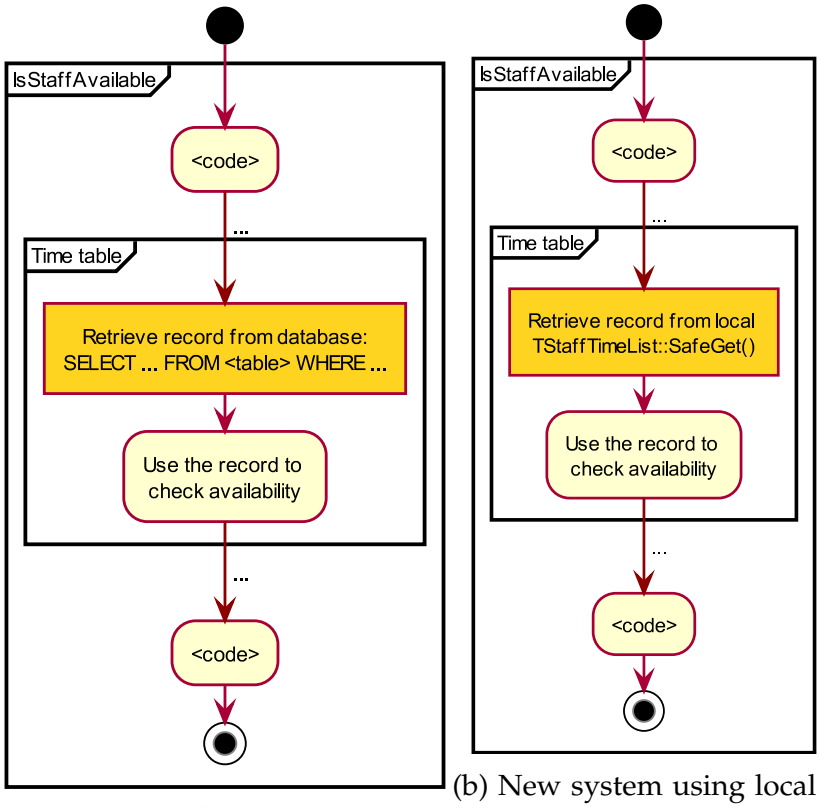
4.4.2 Utilization of cache class TStaffTimeList

To create a reservation, the application must check predetermined rules to complete the process. To make the rules work, the pertinent information must be used. In order to be accurate it needs to be obtained from database. This is the current method used by the application whenever requires data in an operation.

The validation of the staff schedule is done in two phases: 1st data collection, 2nd data use. In the first, data is obtained by requesting it from the database. In the second phase, the data obtained is processed to identify if the staff has a defined time for the period in question. This process is repeated whenever staff time needs to be evaluated. This scheme is shown in Figure 4.6a.

Looking at Figure 4.6b, we observe the new system with messages integrated. The workflow has not changed, it continues with the two phases of obtaining and using data. Moreover, in spite of not being observable in the picture, the availability remains untouched. However, the fundamental change resides in the first phase. Data is now requested from the cache class (which is responsible for its management) through call to *SafeGet()* (Figure 4.4), instead of database. It is the responsibility of the cache class

to read from database only if the requested data is not stored on the local memory. From the caller's perspective, there is no difference in the source of the data, whether taken from local memory or from the database, because it is encapsulated in a local data structure which is then used on either the old system without cache or the new system including cache.



(a) Old system.

(b) New system using local cache.

Figure 4.6: Example of staff timetable availability algorithm.

It is a capital mistake to theorize before one has data.

Sherlock Holmes,
Scandal In Bohemia

5

Performance tests and results analysis

This chapter presents the performance tests and their results, taken from the old system and the new system with the cache. We start by showing the assumptions and components used to perform the tests. Next, we describe the actions taken to execute each test, explain their purpose and what data is analysed. After that, we show the test results accompanied by several graphs. The chapter ends with an analysis of the results obtained.

5.1 Overview

To test the impact of the Message Broadcast Framework integration within the SPA & Leisure application (client), we design a batch of tests to inquire about the response of the system under different system load conditions. These tests were prepared to take

into consideration the immutability of the input conditions and system state at runtime, to provide a reliable comparison between the old system and the new system with local cache.

5.2 Assumptions

The following assumptions were endorsed for the execution of the tests:

- The client-server database system is dedicated to the applications that perform the tests. Therefore, the results obtained depend only on the tests and not of other external factors.
- The used database contains real data, i.e., it is not a purpose-built database with dummy data for testing. For this reason, the results reflect a client-server production environment very accurately.
- Clients are connected to the database server and the EAS through a LAN.
- To follow the conditions of a production environment, connections to the database server use an encrypted channel over Transmission Control Protocol over Internet Protocol (TCP/IP), consequently increasing the network latency. Latency can play a big part in performance (Chapman, 2016, p. 12) and cannot be ignored in our tests.
- The profiling system uses the client components and the facilities of the MBF to capture profiling data and save it into persistent storage for latter analysis.

5.3 System Model

Figure 5.1 shows the architecture assembled to execute the tests and to save the results into a file. There are four parts involved in the profiling system:

Profiling Workstation: is the workstation from which the tests are launched. A set of applications were built to assist the user in the execution of the tests. The

Script-parser application is a small Windows command line application created with the purpose to parse a script file with test case commands and send them to the client connected to the MBF. The Event Inspector application is another GUI Windows application created to diagnose the MBF. We use its functionalities to receive and save to a file, the profiling data captured by the client and forwarded by the EAS.

User Workstation: contains one client application to execute the tests issued by the user through the MBF. Tests requiring more than one application are executed in distinct workstations. With this configuration, the results taken from each client, are not influenced by the other applications also running the tests. Tests were executed using one, two and four application instances running simultaneously.

Database Server: comprise the Relational Database Management System (RDBMS) used by client to store the business information.

Message Server: includes MBF server application. The MBF provides the functionalities necessary to implement the new system with cache by forwarding cache invalidation messages. The MBF is also used to coordinate the simultaneous execution of the tests by clients.

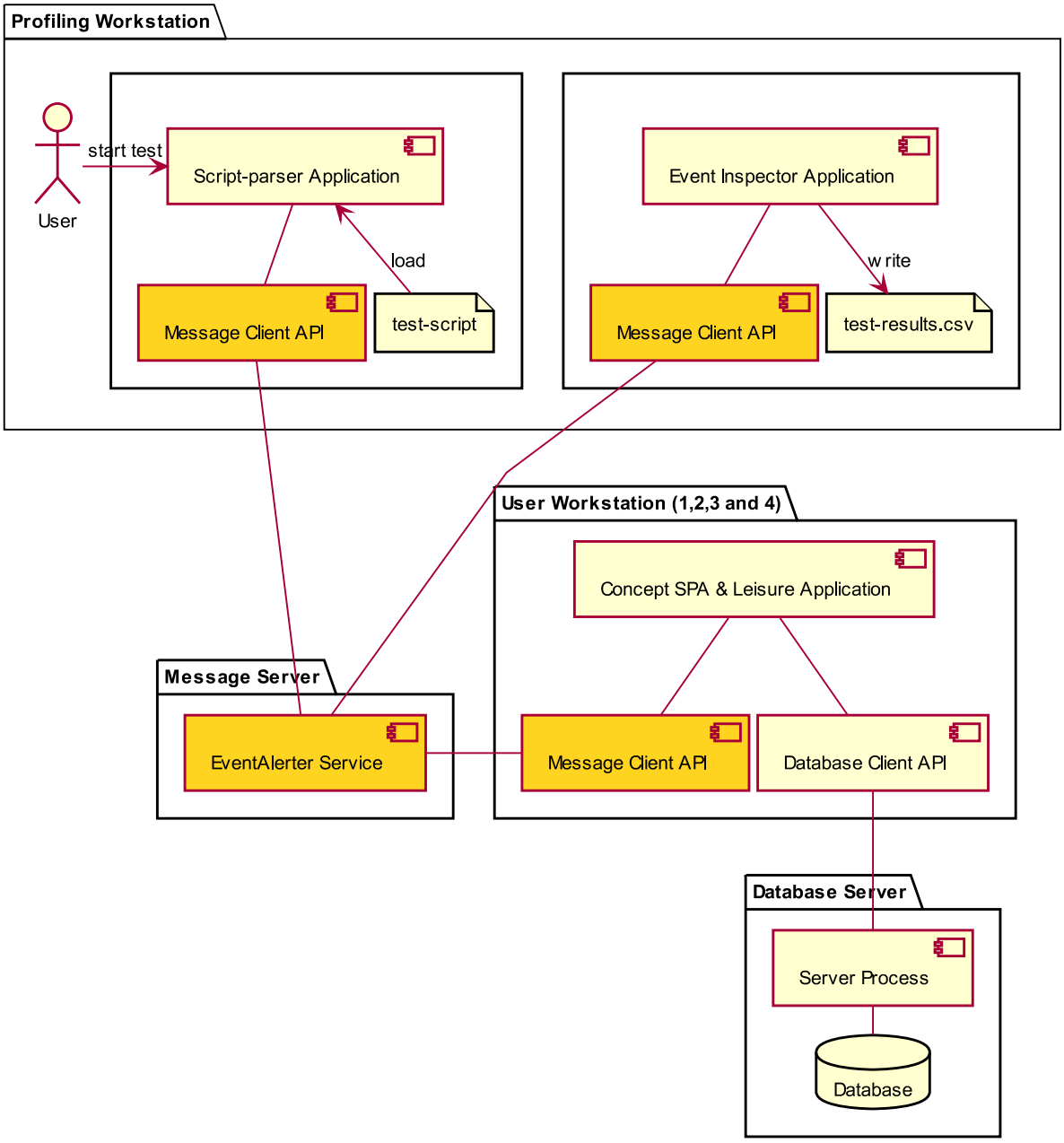


Figure 5.1: Tests architecture model.

5.4 Workload Model

Using the system model shown in Figure 5.1, the execution of the tests proceeds as described below:

1. Choose the test to run and determine how many simultaneous applications are necessary to execute the test.

2. Launch the number of applications required to run the test.

3. Launch Event Inspector application, connect it to the EAS and go to listen mode.

4. Start the Script-parser application and load the test script file to initiate the tests. From here the Script-parser application takes full control of the process. Applications respond to commands sent by the Script-parser application (SP) without any user interaction.

5. When test completes, applications send a profiling message to the Event Inspector application through the EAS with the data captured. The data is then saved into a CSV file for latter analysis.

The tests were directed to the reservation creation process (search engine) because it was the component that received the caching facilities provided by the integration of the MBF (see Chapter 4). The creation of multi reservations in the application let us stress the search engine, as the number of the booking activities can be as high as tens or hundreds per reservation process. The method consists in (a) start the search on the first activity in the list and finding the first available time (according to the established business rules); (b) block the time found to prevent other applications from using that time; (c) select the next activity and restart the search and lock process until it reaches the last activity; and (d) create records in the database, if applicable in the test.

We divided the tests into two categories: volatile (Batch 1) and persistent (Batch 2 and Batch 3). In the first category, the application only searches for availability times and temporarily locks the first found time of each activity in the list (no bookings are created in the database). In the second category, the application executes the same procedures, but it creates bookings on database. Therefore the next reservation session will be affected by the previous sessions (as it would be in a production environment).

The *volatile* tests allow us to see how close are the results when running the same tests on different days, while the *persistent* tests show how the system responds in the presence of busy days (increase of data).

5.5 Profiling data

Table 5.1 describes the profiling data captured in the tests. The parameter *SqlCount* contains the number of requests submitted to the database server using "select" statements; *SqlRecords* contains the number of records returned the "select" operations; *SqlTime* contains the time spent by applications since the submission of the "select" requests to the database server until they receive the results; The *TotalTime* includes the *SqlTime* plus the time spent by applications in the business processes. The last two, *Cached* and *MemSize* refers to the number of records kept in the local memory and to the occupied memory.

Table 5.1: Captured parameters

Name	Description
SqlCount	No. of "select" statements
SqlRecords	No. of records retrieved with "select" statements
SqlTime	Time spent on "select" statements
TotalTime	Total time spent on operation (including SQL statements)
Cached	No. of records cached in local memory
MemSize	Memory occupied by cached records

Percentage Change (Gain)

To calculate the percentage change between values of new system and old system, we used the following expression:

$$Percentage\ Difference^{1} = \frac{NewValue - OldValue}{OldValue} \times 100\,\% \tag{5.1}$$

where *OldValue* is the reference (initial) value and *NewValue* is the target (final) value to compare. The result is the relative change of *NewValue* compared with *OldValue*.

[1]When the result is negative it is a percentage decrease. Because we are comparing the new system with the old system, we expect negative results. To avoid presenting negative values, we reversed the sign of the result to indicate that the new system is behaving better than the old system ('increase of performance').

5.6 Test Results

To measure the client-server database system payload on each test execution, we created a table in the database with one column and one row. Each time a new test starts, applications perform 100 iterations by reading the current column value the table, perform some computations at client-side and update the column with the calculated result. The tests produced a total of 455 records with profiling data gathered from the applications involved in the tests. We obtained an average of 2.990 s with a standard deviation of 0.904 s within the interval of values [1.534 s, 10.146 s] from the payload algorithm.

Figure 5.2 presents the distribution of the time taken to execute the payload algorithm. We can see that 4 values are completely out of the other values, so they can be seen as outliers. All tests were performed under the same network infrastructure conditions and the measurements were recorded using the same procedure. We do not have a concrete explanation of the causes, but we may advance possible reasons for the abnormal results like temporary high process activity in client or server computers, network instability or database server scheduled tasks running at test start time.

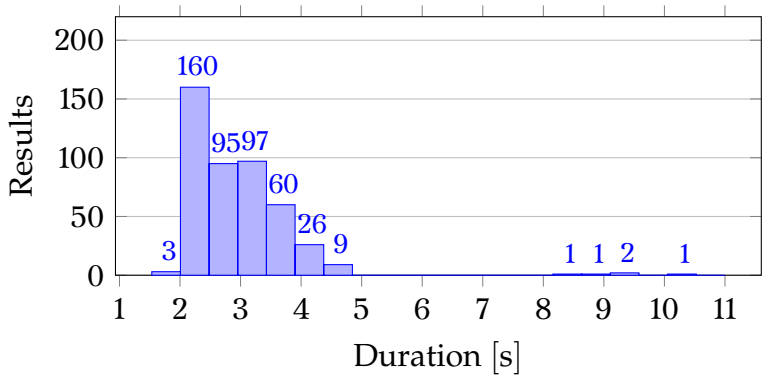


Figure 5.2: Average of the payload algorithm.

We prepared three batches of tests to be executed by the old system (without cache) and by the new system (with daily cache and with weekly cache) as follows:

Batch 1: applications only search for available times and temporarily lock the first time found for each activity in the list. Each test is executed 5 times to obtain more accurate results.

Batch 2: applications search for available times and create bookings in the database using the times locked. The tests of this batch are composed of 5 daily tests over 5 consecutive days. The daily tests are cumulative because each new test execution takes into consideration the reservations created by the previous tests (daily cache). At the start of a new day of tests, the applications are restarted to make the results independent of the previous days.

Batch 3: the same as Batch 2, but with the following exceptions:

1. Applications continue in execution until all tests of the week have been completed. In this way, the cached data is kept in local memory and is available for the following days (weekly cache). Consequently, the daily results are affected by the results of the previous days.
2. Tests executed only by the new system using weekly cache because the old system does not take advantage of the use of the local cache.

We randomly selected a set of 4 activities with different session duration: Chiropractic 15 mins, Acupuncture 30 mins, Swedish Massage 60 mins, Pedicure Intensive 90 mins. The tests evolved a maximum of 20 staff and 11 locations. Each application instance received a unique set of 20 activities using a random combination of these 4 activities. A total of 4 sets were created (one for each application instance) which remained unchanged during the execution of all tests. Tests results are detailed in Appendix A.

5.6.1 Comparison between new and old systems

The comparison between the new system versus the old system shows the impact of storing data on local memory.

Figure 5.3 compares the variation of new and old systems of Batch 2 when the number of applications change and uses data from Appendix A, tables A.3 and A.4. From a general point of view, the number of applications had little effect on results

and the analysed parameters did not follow the same pattern. The number of queries submitted to the database (SqlCount), had a reduction of around 45 %, but the number of records retrieved from the database (SqlRecords) did not follow the same reduction. The reason is that some queries submitted to the database did not return records in either the old system or the new system. The total time spent in operations (TotalTime) had also a considerable cut around (45 %). This reduction is directly related to the less time spent by applications in network and database operations (SQL open time). We can conclude that the time reduction was a direct effect of records kept in local memory.

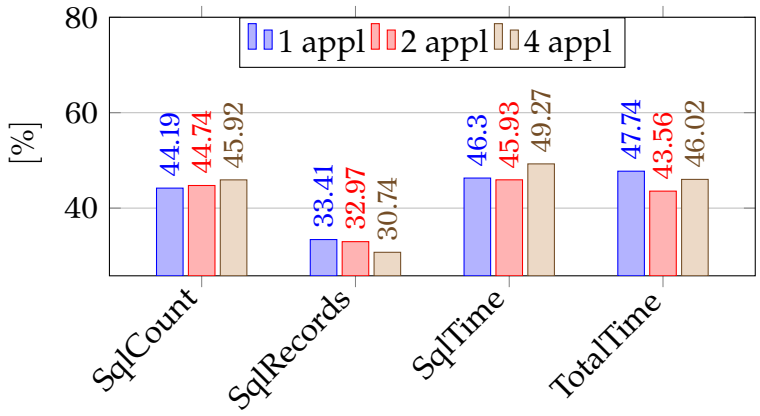
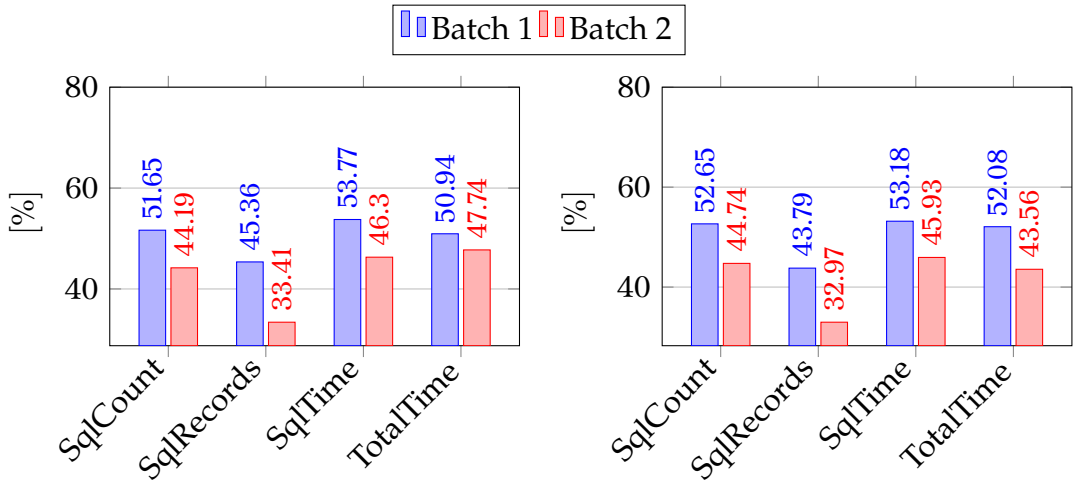


Figure 5.3: Average gain of new system with daily cache over the old system Batch 2.

Figure 5.4, based on data of Appendix A, tables A.1, A.2 and A.3, A.4, compares batches 1 and 2 performance between new system with daily cache and old system. All parameters indicate that the new system always outperforms the old system and the gain obtained varies between 32.97 % (see Figure 5.4b, Batch 2, SqlRecords) and 53.77 % (see Figure 5.4a, Batch 1, SqlTime). The parameter which shows the total time of operations (TotalTime) had 50.94 % improvement in activity search (Batch 1) and 47.74 % improvement in activity search with booking creation (Batch 2), regarding the tests with one application (see Figure 5.4a). Tests with two applications showed the same performance improvement trend across all parameters as with one application, though slightly lower (see Figure 5.4b).

The interpretation we make is that the application significantly reduced the time spent searching for availability times when using local cache as a result of the reduction

of (a) the number of queries submitted to the database and (b) the reduction of records fetched from the database.



(a) 1 application

(b) 2 applications

Figure 5.4: Average gain of the new system with daily cache over the old system.

5.6.2 Comparison between weekly cache and daily cache

The comparison of the new system using different cache configurations shows us what might be the best solution, particularly whether weekly or daily caching.

Figure 5.5, based on Appendix A, tables A.25 and A.4, shows the average gain of the application using weekly cache versus daily cache. The obtained results with one application indicate a small improvement when keeping data on local memory during the week. However, the results with two applications reveal that there was almost no improvement by using weekly cache.

This divergence of results, requires a detailed view break down by day, which is presented on Figure 5.6. Each sub-figure represents the results of a day, based on tables of Appendix A identified in each sub-figure. On one hand, the data presented confirm for one application with weekly cache, although with some variations during the week, a gain between 4.06 % (see Figure 5.6a, SqlCount) and 22.68 % (see Figure 5.6b, SqlTime). On the other hand, it shows that for two applications, there was a performance gain on the first three days, but a loss on the last two days of the week, in particular on

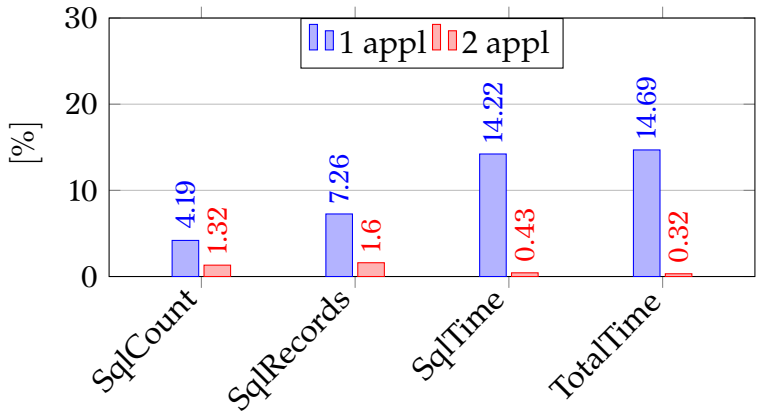
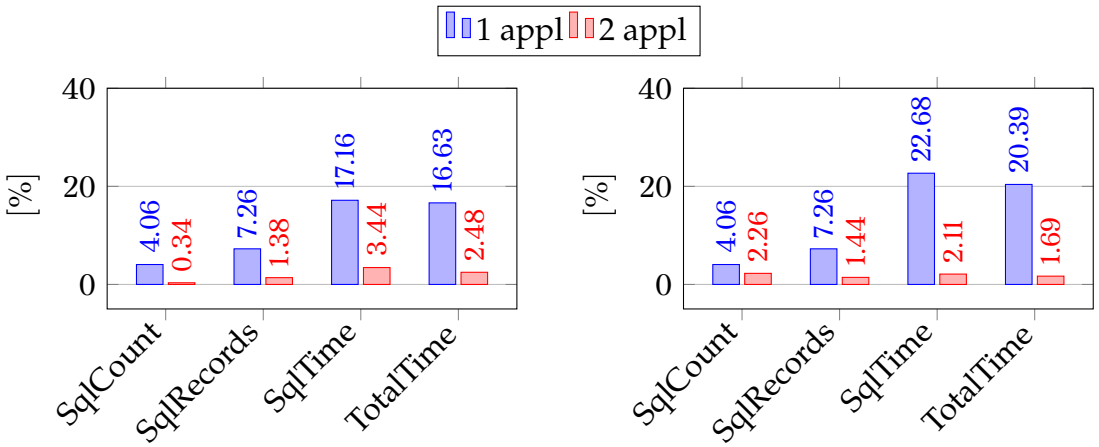


Figure 5.5: Average gain of weekly cache over daily cache.

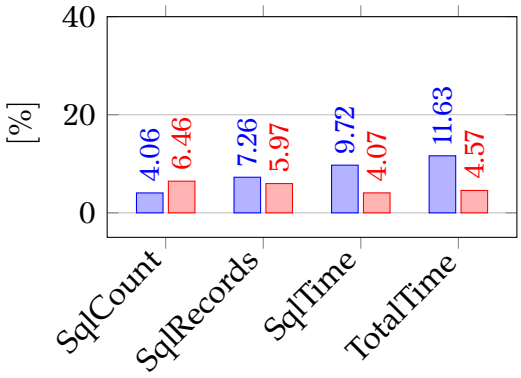
Thursday (see Figure 5.6d), which had a noticeable drop of performance. All parameters are negative, which removes the possibility of network congestion or a busy database server because, under these circumstances, only the time-dependent parameters, *SqlTime* and *TotalTime* would be affected. We investigated possible causes and found nothing weird on configuration, although we find some minor differences in the staff timetable, but nothing that justifies the worst performance.

It is difficult to explain such results in this context. However, we can speculate that the cause of the "less good" results could be due to the algorithm used by the application to fairly distribute the activities by the available staff.

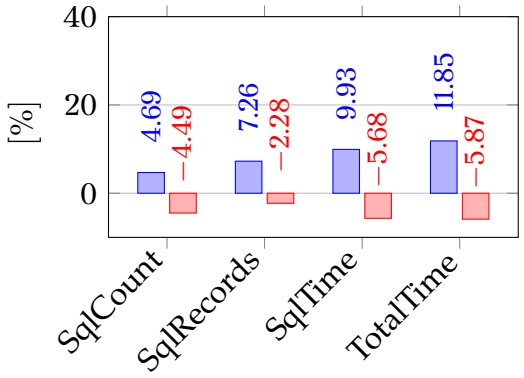


(a) Monday (tables A.26, A.16)

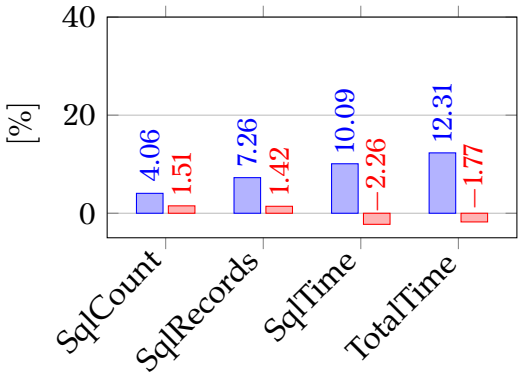
(b) Tuesday (tables A.27, A.18)



(c) Wednesday (tables A.28, A.20)



(d) Thursday (tables A.29, A.22)



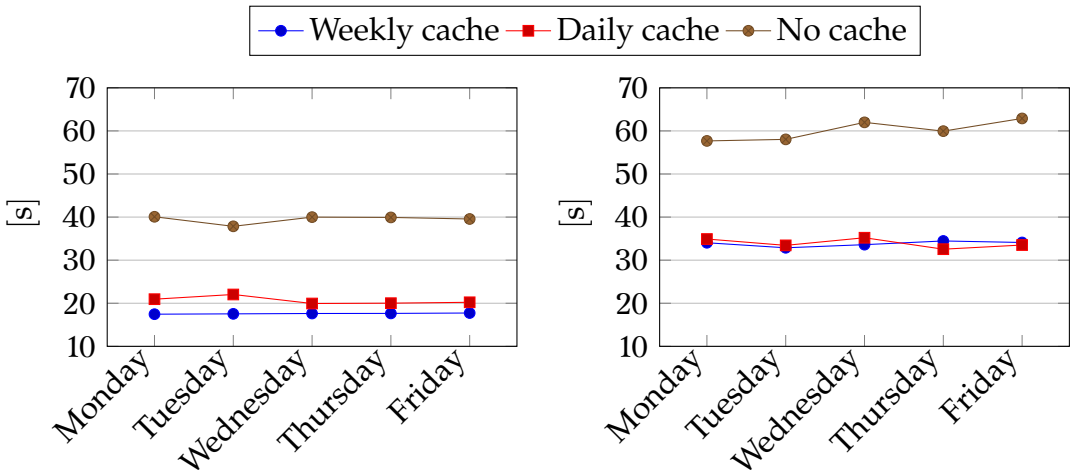
(e) Friday (tables A.30, A.24)

Figure 5.6: Average gain of weekly cache over daily cache break down by day.

5.6.3 Weekly cache vs daily cache evolution over the week days

To better visualize the difference of performance when using no-cache, daily cache we show in this section, the average results evolution of the individual parameters over the week. These parameters are *TotalTime*, *Cached* and *MemSize*. All figures are based on tables of Appendix A.

Based on tables A.26 – A.30 (weekly cache); tables A.16, A.18, A.20, A.22, A.24 (daily cache) and tables A.15, A.17, A.19, A.21, A.23 (no cache), Figure 5.7 shows the total time taken by the application to complete the 5 daily tests. The old system took more time to execute the tests compared to the new system. The new system with weekly cache and daily cache took practically the same time.



(a) 1 application (b) 2 applications

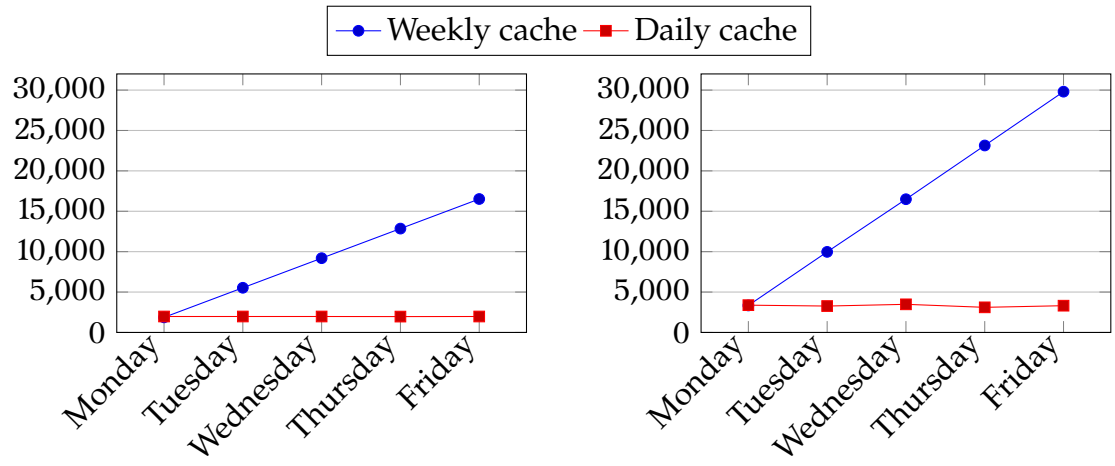
Figure 5.7: Evolution of *TotalTime* parameter over the week.

Figure 5.8 shows the average of the number of cache records of the five daily tests based on tables A.26 – A.30 (weekly cache) and tables A.16, A.18, A.20, A.22, A.24 (daily cache). In both figures, Figure 5.8a and Figure 5.8b, the number of records cached in the local memory remains practically constant throughout the week for the new system with daily cache and increases linearly for the new system with weekly cache.

A very important parameter, *MemSize*, which indicates the impact in the memory required to store the cached records on local memory, based on tables A.26 – A.30 (weekly cache) and tables A.16, A.18, A.20, A.22, A.24 (daily cache), is introduced in the Figure 5.9. The behaviour observed, is identical to the parameter *Cached*, i.e., they both keep constant with daily cache and increase with weekly cache. This relation is expected because the records cached have the "same" individual size[2]

It is important to highlight the fact that in the worst case, in total, there were

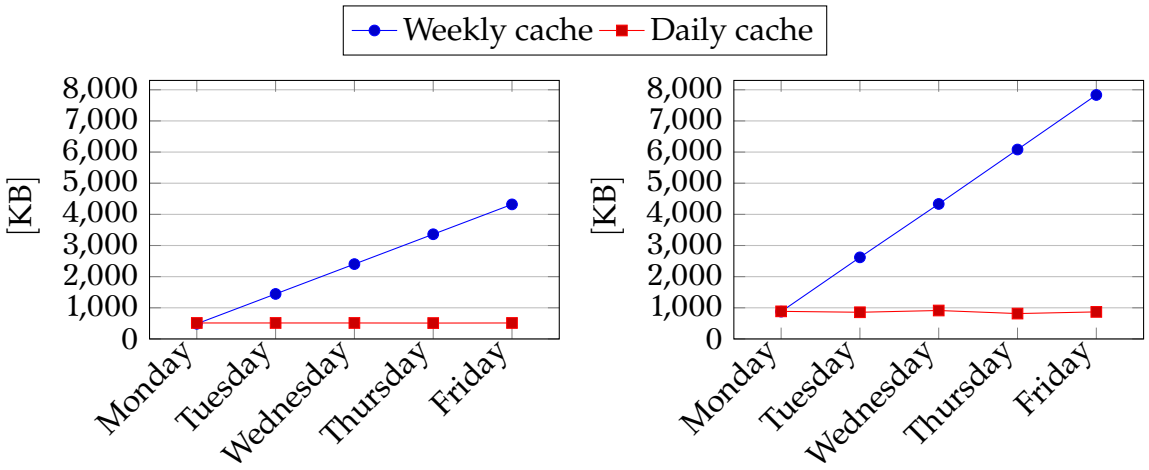
[2]This statement is not 100 % correct because each record may have a different size depending on the data, but in general, they do not differ much. For example, the activity name "Acupuncture 30 mins", requires less space in memory than "Pedicure Intensive 90 mins".



(a) 1 application (b) 2 applications

Figure 5.8: Evolution of *Cached* parameter over the week.

nearly 30 000 records cached occupying around 8 MB in local memory. These values demonstrate the low impact in memory, according to the memory capacity of current equipment. According to these results, we can infer that it would still be possible to cache records of a month before applying a record retention policy.



(a) 1 application (b) 2 applications

Figure 5.9: Evolution of *MemSize* results over a week.

5.6.4 Results Analysis

From the previous results, we see that the new system with daily cache performed quite significantly better compared to the old system without cache. The total operation

time had an improvement of about 40 %. However, when comparing the new system including weekly cache with the new system including daily cache, there were much smaller improvements, mainly in the 2 applications case, despite the higher number of records stored on local memory.

Both tests of Batch 2 (daily cache) and Batch 3 (weekly cache), were designed to create daily reservations using the availability search and booking process in one-pass. In total, 5 daily tests were performed for 5 days, starting the availability search always from 08:00, which mean that on each new daily test, the application had to search for the times already taken from the previous tests before finding available times to book. Each test stored the relevant data on the local memory. Therefore, it was possible to use part of the records stored in the memory obtained in the previous tests. So far, this method was common to both cache models.

The difference was when moving to the next day, the system with daily cache purged all cached records from memory, and the system with weekly cache maintained the records. The system with weekly storage should have been much more efficient, but it was not, because the records stored in memory mainly concerned the current day and previous days. Therefore, when starting a new day, regardless of the cache type, most of the needed records were not in memory; consequently, they needed to be retrieved from the database to be available for subsequent queries. This was the reason why the weekly cached results did not perform better, even while keeping more records in memory.

However in a production environment, the search and booking creation process does not happen always in a sequential fashion. For example, clients may ask for availability on Monday, then on Friday and again on Monday. Therefore the application may search for the same availability multiple times before creating a booking.

In the system with daily cache whenever the search day changes, records are removed from memory. Therefore any records required for the availability check must be retrieved from the database and cached again. On the other hand, in the system with weekly cache, records are removed from memory only when the search week changes. The change to a day in the same week is supposed to happen more often than a change

to another week. Therefore the records in the weekly cached system are supposed to be reused longer, reducing the number of requests submitted to the database.

For the reasons appointed above, we consider that the new system with the weekly cache has significant advantages in a production utilization context.

6

Conclusions

The SPA & Leisure application is being developed for about twenty years, containing hundreds of thousands of source code lines. The search engine component used to find available times to book contains some classes and functions shared by other areas of the application, with complex business logic. The rules and restrictions are very challenging and arduous to rewrite.

To overcome the decrease in performance due to the booking process in response to the continuous increase of data in the database, we proposed and implemented a broadcast message framework to allow store data on local memory. This framework acts as a parallel system keeping watching data changes on the database and as soon as data change, a message is sent to the applications to inform them that current data held is out-of-date and must be refreshed before the next use.

The proposed framework was based on the following premises:

- Load of data, data is retrieved from the database and kept in memory using local variables.
- Usage of data, when it is required, data is accessed directly from local memory. After use, memory is not destroyed, it is available for use next time.
- Maintenance of data, the message broadcast system notifies connected applications to refresh or flag local data as out-of-date to ensure it is up to date when it is needed.

The Message Broadcast Framework integration was designed to not require significant changes to the application code base or directly interfere with existing functionality and to be easily integrated and tuned in the application. For example, in the presence of a large number of updates on the database, it may be necessary to turn off or adjust the local cache of data affected.

The results of the tests realized, clearly show that the new system using local data outperforms the old system. The most important achievement was the reduction of the total time of the booking process in around 50 %. It is important to clarify that these results were achieved only by storing the data in local memory for later use, keeping the availability search algorithm unchanged[1].

Another point is the amount of local memory required to hold the cached data. Our results showed that for two applications, the total memory used was below 8 MB when using weekly cache and bellow 5 MB when using the daily cache.

In summary, based on the test results obtained, we conclude that using the new system with cache yields a relevant performance gain over the old non-cached system (around 50 %) and that the memory needed to store data is negligible for today's storage capacity. Lastly, the use of weekly cache is justified for normal daily operations, because usually, the same availability times are sought more than once before booking creation.

[1]While this is true, there were modifications in the application to support the integration of the MBF, but they were not related to the application's business rules.

6.1 Publications

One publication resulted from the present work entitled "Message broadcast framework for local storage in distributed applications" (A. Miguel[2,4] R. Lam[3[000-0003-4297-2441]] G. Schütz[3,4[0000-0001-5081-3913]]), available at https://doi.org/10.1007/978-3-030-30938-1_25, and was presented with oral communication in the INCREaSE 2019, Proceedings of the 2nd International Congress on Engineering and Sustainability in the XXI Century, at Instituto Superior de Engenharia from Universidade do Algarve, Faro, Portugal.

6.2 Future Work

We intend to extend the current work into three areas. The first area is the integration of the framework into other parts of the application. Figure 3.13 (Chapter 3) showed few areas that would have a significant benefit using the facilities offered by the MBF, in particular, the *Agenda Form*. Operators spend a substantial part of the day using the main agenda, whether to view, create, check-in or check-out reservations. Whenever the operator changes the calendar view or makes changes in reservations, the agenda is always filled with information retrieved from the database. Much of this information has a low change rate, such as staff time, so keeping this data in local memory will reduce the number of times the database returns the same information significantly.

The second area is to make better use of the local memory by refactoring some parts of the application related to the management of the cached records. Instead of submitting SQL statements with specific searches to the database server, the application submits less restrictive queries, therefore transferring part of the logic contained in the SQL statement into the client application. The specific filtering is to be carried by the *SafeGet()* method (of each cache class) when the application requests a record from the local cache. With this new approach, the number of searches in the database, as well as the number of downloaded records, should decrease substantially.

[2]Institute of Engineering of the University of Algarve (MEEE), Portugal
[3]Institute of Engineering of the University of the Algarve, Portugal
[4]Center for Electronic, Optoelectronic and Telecommunications (CEOT)

The last area of interest is to investigate how to send notification messages directly from the database server, in the context of a transaction, as referred in Chapter 1, Subsection 1.3.3, but using the EACAPI. The objective is to transfer part of the client application business logic into the database. In current notification implementation, it is the responsibility of the client to send invalidation messages after performing changes on the database. With this new design, the notification process will be initiated by the database server after it commits the transaction. We expect client applications to be notified faster with the most recent snapshot of the changed data.

Bibliography

Abraham Pizam (Ed.). (2005). *International encyclopedia of hospitality management*. Taylor & Francis.

Babuškov, M. (2005). The Power of Firebird Events. Retrieved September 19, 2019, from https://firebirdsql.org/file/documentation/papers_presentations/Power_Firebird_events.pdf

Bardi, J. A. (2002). *Hotel front office management*. Wiley.

Beasley, J. S., & Nilkaew, P. (2015). *Networking essentials: A comptia network+ n10-006 textbook* (4th edition). Pearson IT Certification.

Bukhari, F. (2012). *Maintaining Consistency in Client-Server Database Systems with Client-Side Caching* (Doctoral dissertation). Newcastle University.

Burleson Consulting. (2019). Oracle TNS Ping tips. Retrieved November 23, 2019, from http://www.dba-oracle.com/tips_oracle_tnsping_command_example.htm

Chapman, C. (2016, May 1). *Network performance & security*. Elsevier.

Conceptek. (2019). Concept Spa & Leisure. Retrieved January 15, 2019, from https://concept.shijigroup.com/en/products/concept-spa-leisure-software/concept-spa-leisure/

Council, P. S. S. (2020). Securing the future of payments together. Retrieved February 1, 2020, from https://www.pcisecuritystandards.org/

Eckel, B. (2000). *Thinking in C++* (2nd ed). Prentice Hall.

Embarcadero. (2019). RAD Studio. Retrieved December 11, 2019, from https://www.embarcadero.com/products/rad-studio

Embarcadero, I. C., An Idera. (2009). System.Classes.TStrings.CommaText. Retrieved September 10, 2019, from http://docwiki.embarcadero.com/Libraries/Rio/en/System.Classes.TStrings.CommaText

FirebirdSQL. (2019). The true universal open source database. Retrieved September 19, 2019, from https://firebirdsql.org/

GDPR. (2020). Rules for business and organisations. Retrieved February 1, 2020, from https://ec.europa.eu/info/law/law-topic/data-protection/reform/rules-business-and-organisations_en

Glass, G. (1993). *UNIX for programmers and users: A complete guide*. Prentice Hall.

González, B., & Thiruvathukal, G. (2006). A Distributed StorageFramework. Retrieved April 14, 2019, from http://www.linuxclustersinstitute.org/conferences/archive/2006/PDF/08-Gonzales_B_final.pdf

Hower, C. Z. (2006). Internet Direct (Indy). Retrieved January 8, 2019, from http://ww2.indyproject.org/docsite/html/frames.html?frmname=topic&frmfile=index.html

Indy Pit Crew, H., Chad Z. (2020). Indy 10 documentation. Retrieved May 6, 2020, from https://www.indyproject.org/documentation/

Lucid Software Inc. (2019). Data Flow Diagram. Retrieved January 27, 2019, from https://www.lucidchart.com/pages/data-flow-diagram

Microsoft. (2009). Pushing the limits of windows: Processes and threads. Retrieved April 28, 2020, from https://docs.microsoft.com/pt-pt/archive/blogs/markrussinovich/pushing-the-limits-of-windows-processes-and-threads

Microsoft. (2019a). Introduction to Windows Service Applications. Retrieved March 24, 2019, from https://docs.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications

Microsoft. (2019b). Multihreading. Retrieved July 4, 2019, from https://docs.microsoft.com/en-us/cpp/parallel/multithreading-creating-user-interface-threads?view=vs-2019

Microsoft. (2019c). Synchronization Functions. Retrieved January 15, 2019, from https://docs.microsoft.com/pt-pt/windows/desktop/Sync/synchronization-functions

Microsoft. (2019d). User mode and kernel mode. Retrieved June 20, 2019, from https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

Microsoft. (2019e). WM_copydata message. Retrieved March 27, 2019, from https://docs.microsoft.com/en-us/windows/desktop/dataxchg/wm-copydata

Microsoft. (2020). WM_user message range numbers. Retrieved June 14, 2020, from https://docs.microsoft.com/en-us/windows/win32/winmsg/wm-user

Naderializadeh, N., Maddah-Ali, M. A., & Avestimehr, A. S. (2017). On the optimality of separation between caching and delivery in general cache networks. *2017 IEEE International Symposium on Information Theory (ISIT)*, 1232–1236. Retrieved August 21, 2019, from http://ieeexplore.ieee.org/document/8006725/

Pressman, R. (2009). *Software Engineering: A Practitioner's Approach* (7th edition). McGraw-Hill Education.

Shiji Group. (2020). Concept golf & spa. Retrieved June 7, 2020, from https://www.shijigroup.com/brands/concept-spa

Shipley, T. G., & Bowker, A. (2014). Internet Criminals. *Investigating Internet Crimes* (pp. 21–39). Elsevier. Retrieved August 21, 2019, from https://linkinghub.elsevier.com/retrieve/pii/B9780124078178000023

Silberschatz, A., Korth, H. F., & Sudarsham, S. (2006). *Database System Concepts* (International Edition 2006). Mc Graw Hill.

Spa Business Handbook. (2015). Spa Business Handbook. *Spa Business Handbook*. www.spahandbook.com

Stallings, W. (2018). *Operating systems: Internals and design principles* (9th edition, global edition). Pearson.

Vakali, A. (1999). A Web-based evolutionary model for Internet data caching. *Proceedings. Tenth International Workshop on Database and Expert Systems Applications. DEXA 99*, 650–654. Retrieved August 21, 2019, from http://ieeexplore.ieee.org/document/795261/

Walsh, I. (2020). Business rules vs. business requirements. Retrieved June 21, 2020, from https://www.brcommunity.com/articles.php?id=b631

Wikipedia. (2018). Passive Data Structure. Retrieved March 27, 2019, from https://en.wikipedia.org/wiki/Passive_data_structure

Wikipedia. (2019). Instant Messaging. Retrieved November 24, 2019, from https://en.wikipedia.org/wiki/Instant_messaging

A

Results

Results presented in this section are the average of the values captured when executing performance tests with the application.

A.1 Batch 1

Average results from 5 tests performed using the same initial conditions and with no bookings creation.

A.1.1 Daily results

Table A.1: Daily average results (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	3,854.0	2,867.0	9.206	15.557
2	5,011.3	3,681.5	11.705	18.994

Table A.2: Daily average results (new system with daily cache)

Appl	SqlCount	SqlRecords	SQL Time (s)	Total Time (s)	Cached	MemSize (KB)
1	1,863.4	1,566.4	4.256	7.632	549.0	141.947
2	2,373.1	2,069.3	5.481	9.102	744.6	192.168

A.2 Batch 2

Average results from 5 daily tests in the period of 5 days with bookings creation. Data stored on local memory was discard on each new day.

A.2.1 Weekly results

Table A.3: Weekly average results (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	9,486.6	6,891.8	23.890	39.480
2	17,159.0	13,469.4	39.560	60.108
4	31,611.0	28,550.1	83.051	124.528

Table A.4: Weekly average results (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,294.2	4,589.2	12.828	20.633	1,972.0	512.582
2	9,482.1	9,028.5	21.389	33.923	3,309.6	867.917
4	17,096.3	19,772.9	42.128	67.220	5,902.8	1,553.251

A.2.2 Weekly results partition by test of the day

Table A.5: Weekly average results of test No. 1 of the day (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	2,918.0	2,037.0	8.809	16.897
2	3,960.7	2,827.5	9.519	15.525
4	6,953.1	4,677.5	18.425	29.041

Table A.6: Weekly average results of test No. 1 of the day (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	1,581.0	1,162.0	4.417	7.565	402.0	102.456
2	2,529.7	1,883.0	6.136	10.024	622.7	160.400
4	4,208.7	3,017.6	10.872	17.110	998.3	259.817

Table A.7: Weekly average results of test No. 2 of the day (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	5,913.0	4,294.0	15.303	26.614
2	10,056.6	6,928.8	23.464	35.918
4	19,637.9	14,594.4	51.009	76.373

Table A.8: Weekly average results of test No. 2 of the day (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	3,373.0	2,489.0	8.601	13.882	1,009.0	260.784
2	5,763.3	4,403.1	13.144	20.820	1,585.8	413.560
4	10,980.3	9,353.3	28.155	43.741	2,844.2	746.691

Table A.9: Weekly average results of test No. 3 of the day (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	9,369.0	6,530.0	23.387	38.701
2	16,854.8	12,450.7	38.855	58.875
4	33,297.3	28,070.8	86.958	129.667

Table A.10: Weekly average results of test No. 3 of the day (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,201.0	4,175.0	12.700	20.380	1,726.0	447.995
2	9,363.1	8,236.8	20.922	32.994	2,916.9	764.081
4	18,107.2	18,694.1	44.511	69.389	5,391.7	1,419.050

Table A.11: Weekly average results of test No. 4 of the day (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	13,023.0	9,369.0	32.290	51.876
2	24,133.8	19,241.3	55.390	83.660
4	46,461.3	43,220.9	120.804	181.450

Table A.12: Weekly average results of test No. 4 of the day (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	7,304.2	6,353.4	17.336	27.614	2,758.6	717.998
2	13,135.2	12,890.6	29.217	46.122	4,691.3	1,231.827
4	25,040.3	29,896.1	61.848	99.059	8,615.2	2,269.280

Table A.13: Weekly average results of test No. 5 of the day (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	16,210.0	12,229.0	39.662	63.313
2	30,788.9	25,898.6	70.573	106.563
4	51,705.6	52,186.8	138.058	206.109

Table A.14: Weekly average results of test No. 5 of the day (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	9,012.0	8,766.6	21.087	33.726	3,964.2	1,033.677
2	16,619.1	17,729.0	37.528	59.652	6,731.5	1,769.717
4	27,145.0	37,903.6	65.252	106.804	11,664.4	3,071.421

A.2.3 Daily results

Table A.15: Average results of Monday (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	9,485.2	6,891.4	24.842	40.071
2	16,566.1	13,130.4	37.842	57.678
4	30,777.4	27,663.6	77.325	117.298

Table A.16: Average results of Monday (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,286.0	4,588.6	13.155	20.945	1,974.2	513.170
2	9,621.7	9,015.5	22.086	34.907	3,377.8	885.915
4	16,904.1	18,991.4	40.527	65.350	5,888.4	1,549.424

Table A.17: Average results of Tuesday (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	9,485.2	6,891.4	24.587	37.857
2	16,844.9	13,418.1	38.162	58.037
4	31,761.6	28,952.7	85.346	126.322

Table A.18: Average results of Tuesday (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,286.0	4,588.6	14.171	22.036	1,974.2	513.170
2	9,328.2	9,063.5	21.094	33.437	3,267.6	856.942
4	16,851.0	19,586.9	38.262	62.380	5,823.9	1,532.322

Table A.19: Average results of Wednesday (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	9,485.2	6,891.4	23.324	39.993
2	17,606.9	13,698.2	40.804	61.985
4	30,827.8	28,445.7	78.533	119.606

Table A.20: Average results of Wednesday (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,286.0	4,588.6	12.228	19.942	1,974.2	513.170
2	9,950.2	9,052.2	22.164	35.207	3,481.5	913.292
4	17,122.3	19,912.3	44.197	69.461	5,902.8	1,553.136

Table A.21: Average results of Thursday (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	9,492.2	6,893.4	23.494	39.921
2	16,610.6	12,996.3	39.628	59.948
4	33,061.1	29,345.0	89.269	132.852

Table A.22: Average results of Thursday (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,327.2	4,591.6	12.248	20.018	1,963.0	510.230
2	9,041.0	8,924.2	20.569	32.549	3,109.2	814.900
4	17,591.1	20,381.2	44.496	70.298	6,018.5	1,584.280

Table A.23: Average results of Friday (old system)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)
1	9,485.2	6,891.4	23.204	39.559
2	18,166.3	14,103.9	41.365	62.893
4	31,627.2	28,343.3	84.779	126.562

Table A.24: Average results of Friday (new system with daily cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,286.0	4,588.6	12.340	20.225	1,974.2	513.170
2	9,469.3	9,087.1	21.034	33.513	3,312.1	868.535
4	17,013.0	19,992.9	43.156	68.613	5,880.3	1,547.095

A.3 Batch 3

Average results from 5 daily tests in 5 days with the creation of reservations. As the data stored on the local memory was not discarded during the execution of the tests, *Cached* and *MemSize* will increase in all the days because they accumulate the results from the previous days, as opposite to *SqlCount*, *SqlRecords*, *SqlTime* and *TotalTime* which store the results of each current day individually.

A.3.1 Weekly Results

Table A.25: Weekly average results (new system using weekly cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,072.6	4,256.1	11.004	17.602	9,192.3	2,403.376
2	9,357.3	8,884.4	21.297	33.816	16,550.9	4,348.962

A.3.2 Daily results

Table A.26: Average results of Monday (new system using weekly cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,071.4	4,255.6	10.898	17.463	1,867.8	486.682
2	9,588.9	8,891.4	21.327	34.041	3,342.5	876.755

Table A.27: Average results of Tuesday (new system using weekly cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,071.4	4,255.6	10.956	17.542	5,529.8	1,444.964
2	9,117.4	8,932.8	20.648	32.874	9,972.7	2,619.778

Table A.28: Average results of Wednesday (new system using weekly cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,071.4	4,255.6	11.039	17.623	9,191.8	2,403.247
2	9,307.8	8,511.9	21.262	33.599	16,492.1	4,333.583

Table A.29: Average results of Thursday (new system using weekly cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,077.6	4,258.2	11.032	17.645	12,854.2	3,361.636
2	9,446.6	9,128.0	21.738	34.459	23,142.7	6,081.742

Table A.30: Average results of Friday (new system using weekly cache)

Appl	SqlCount	SqlRecords	SqlTime (s)	TotalTime (s)	Cached	MemSize (KB)
1	5,071.4	4,255.6	11.095	17.736	16,517.8	4,320.346
2	9,325.9	8,957.7	21.509	34.106	29,804.5	7,832.953