Shared computer systems
and groupware
development

Escaping the personal computer paradigm
Abstract

For the majority of the computers existence, we humans have interacted with them in a similar way, usually with a strict one-to-one relationship between user and machine. This is reflected by the design of most computers, operating systems and user applications on the market today, which are typically intended to only be operated by a single user. When computers are used for teamwork and cooperation, this design philosophy can be restricting and problematic. This paper investigates the development of shared software intended for multiple users and the impact of the single user bias in this context. A prototype software system was developed in order to evaluate different development methods for shared applications and discover potential challenges and limitations with this kind of software. It was found that the development of applications for multiple users can be severely limited by the target operating system and hardware platform. The authors conclude that new platforms are required to develop shared software more efficiently. These platforms should be tailored to provide robust support for multiple concurrent users. This work was carried out together with SAAB Air Traffic Management in Växjö, Sweden and is a bachelor’s thesis in computer engineering at Linnaeus University.
Contents

Abstract ........................................................................................................................................... 2

1 Introduction .................................................................................................................................. 1
  1.1 Background ............................................................................................................................... 1
  1.2 Project goals ............................................................................................................................. 2
  1.3 Prototype description and requirements .................................................................................. 3
  1.4 Motivation .................................................................................................................................. 4
  1.5 Scope .......................................................................................................................................... 4
  1.6 Method ....................................................................................................................................... 5

2 Concepts, Theory and Technology ................................................................................................. 6
  2.1 Remote Tower current implementation ..................................................................................... 6
     2.1.1 Synergy ............................................................................................................................... 6
     2.1.2 Remote Tower Synergy integration ..................................................................................... 6
  2.2 Relevant concepts and technologies ......................................................................................... 7
     2.2.1 Personal computers ............................................................................................................. 7
     2.2.2 Groupware and collaborative software .............................................................................. 7
     2.2.3 Graphical User Interface ................................................................................................... 8
     2.2.4 Operating systems, desktop environments and window managers .................................. 8
     2.2.5 Mouse & Keyboard input in UNIX and UNIX-like systems ................................................. 9
     2.2.6 Distributed computer applications ................................................................................... 9
     2.2.7 C++ .................................................................................................................................... 10
     2.2.8 Qt ....................................................................................................................................... 10
     2.2.9 Raspberry Pi/Raspbian ....................................................................................................... 11
  2.3 Shared and personal computer systems ..................................................................................... 11

3 Analysis ......................................................................................................................................... 12
  3.1 Methods for shared application interface development ............................................................ 12
     3.1.1 Tailoring - Application specific custom implementation .................................................... 12
     3.1.2 Adaptation - Integration with external frameworks ............................................................ 13
     3.1.3 Innovation - Custom input handling and UI management scheme ................................... 14
  3.2 Method evaluation and implementation .................................................................................... 14
6.2.5 Qt and general development considerations .................................. 34
6.3 Conclusion ......................................................................................... 36
6.4 Future Research ............................................................................... 37
References ............................................................................................... 38
1 Introduction

Computers are among the most common tools at any workplace today, but even though work is often carried out collaboratively in teams computer systems are generally designed to be controlled by a single user. This design pattern introduces many limitations when real-time cooperation between multiple users in a single shared environment is desired.

Collaborative software, also known as groupware, exists to promote group collaboration and communication in the digital realm. However, these tools are usually built in and for a single-user computer environment and thus tend to support only one input source. Virtual solutions exist for allowing multiple users to control the same input in a many-to-one relationship, but most operating systems don’t support multiple-input to multiple-output interactions involving several users. This makes true real-time cooperation where multiple users can interact locally with a system independently of each other impossible at the operating system level.

For this reason, multi-user interactions typically have to be implemented higher up in the program stack. This thesis project seeks to investigate how several users can interact with the same instance of a system, and is carried out together with SAAB Air Traffic Management in Växjö, Sweden. A prototype program was created to explore this kind of development. Said prototype demonstrates solution strategies that let multiple users connect and perform mouse and keyboard operations simultaneously without interfering with each other.

1.1 Background

Since the introduction of the first desktop computers in the 1970s, they are probably what most people think of when they imagine a computer. Designed to be more portable and compact than their earlier counterparts, the desktop computers were small enough to fit on a desk or in an office. This technological leap sparked the microcomputer revolution and ultimately led to the dominance of the personal computer in the consumer market as well as many businesses. These personal computers were intended to be operated by a single person, typically with a keyboard and monitor.

Computers and their applications have changed much since the time of their introduction, but are still typically operated in a similar fashion. Today computers are a standard tool in most workplaces and are often used to communicate, plan and carry out various other tasks. Multiple users frequently operate on the same set of data at
the same time, but despite this, the one user per computer paradigm has remained dominant ever since the inception of the personal computers.

At the time of writing, SAAB Air Traffic Management, a subdivision of SAAB, is in the process of developing the Remote Tower system for remotely managing air traffic control towers. This technology lets operators manage airport traffic at any site where it is deployed through a connected computer system. One responsibility of the system is to provide operators with control over airport subsystems like lighting and optics. Currently, several operators can be active in the system simultaneously with the aid of external 3rd party software running on top of the operating system of the host machines. However, this software is not tailored to the requirements of SAAB:s specific system, and integration is less than optimal.

In this environment, it is desirable to let multiple operators work together simultaneously. This lets them manage different aspects of the system, hand over operations or simply communicate and cooperate more effectively in general. It is also necessary to provide redundancy backup in the case of a catastrophic failure of either operator or system. Furthermore, it is critical that the user interface is responsive, reliable and capable because of the high stakes and security requirements involved in air traffic control. The purpose of this paper is to examine how such an interface for multiple users could be developed.

1.2 Project goals
- Discover or develop methods and techniques for writing shared applications for multiple simultaneous users
  - Evaluate these methods with respect to development effort, functionality provided, code portability, user experience and overall quality of the product
- Discover potential problems and limitations with this kind of shared system
  - Attempt to provide strategies and techniques to solve, bypass or mitigate these problems and limitations
- Deliver a prototype program to SAAB that demonstrates certain properties specified in the following section of this paper
1.3 Prototype description and requirements

The prototype system delivered to SAAB should have the following properties and features:

- Distributed and controlled by several users on separate machines simultaneously
- Support real-time interaction and visual feedback
- Contain a shared working environment with interactive components
- Local network communication
- Deployable in a hardware environment similar to that of the current configuration of the Remote Tower system

The system should consist of three logical components with distinct responsibilities, each with its respective requirement criteria.

- User input
  - Support for two or more input terminals with mouse and keyboard. User input is forwarded to a separate server over the connected network.

- Program state and interaction logic
  - A server which receives and interprets forwarded mouse inputs from multiple connected users.
  - Keeps track of separate users and interactions with objects in the application environment.
  - Three different types of objects are recognized and implemented:
    - Statically positioned objects
    - Objects movable by users
    - Moving objects, automatically tracked by active user cursors
  - Objects should be clickable and movable by dragging the mouse
  - Deployable on Windows 7/10

- Output and rendering
  - Program state should be viewable from machines other than the program server. States should be synchronized and consistent across all machines.
  - It should be possible to divide the entire workspace into smaller parts and display only a select portion on any given output machine.
  - Deployable on Windows 7/10
1.4 Motivation

Having multiple users concurrently operating a system brings many potential benefits. Operators can communicate and cooperate more easily when they can share the same working environment. They can also assume each other’s responsibilities as needed, adding another layer of redundancy for reliability and safety reasons. However, implementing support for multiple concurrent users in applications for Windows and other mainstream OS:s is non-trivial because of limitations in how user input is handled in the operating system. Therefore a custom solution to this problem is required.

Distributing an application across several separate machines also has many benefits when done properly. Performance can be enhanced since the total workload on any given machine is lowered, which reduces the risk of bottlenecking in the system. The total computational power can also be increased by adding more hardware to the system. Reliability can be improved by providing redundant backups and reducing the number of critical failure points.

1.5 Scope

This project is intended as an exploratory investigation into writing shared multi-user applications. The aim is not to put forth a complete and comprehensive method for software development, nor any specific techniques or technologies.

The prototype to be delivered is meant as a proof of concept rather than an early version of software to be implemented or a definitive solution to the problems that have been introduced in this paper. Thus, the focus is on the lower layers of the application stack rather than complex high-level interactions between users and objects. Furthermore, all user interactivity will be replaced completely upon potential integration with SAAB:s existing system. For this reason, the prototype application will not define any user interactions other than clicking and moving objects and tracking them with the mouse cursor. As SAAB:s current system does not rely on any keyboard commands, the prototype does not need to fully implement keyboard functionality. However, the system should support the transmission of keyboard events although they are not implemented on the receiving end. Keyboard events are simply discarded upon arrival.

While the prototype application should be distributed, the majority of the development effort will be directed towards user interactivity and not the distribution
of the program. Due to the exploratory nature of the project, the quality, portability, stability, robustness and performance of code is not a primary concern.

1.6 Method

First, the problem area was analyzed. A list of requirements was produced that when fulfilled would constitute a prototype application that demonstrates a viable partial solution to the problems identified. Two prototypes were developed and tested against said requirements. An attempt was then made to generalize the prototypes and make the code more portable and reusable. Finally, the process of developing the prototypes was analyzed in order to further understanding of the problem area.

The development of the prototypes followed the Kanban model of project management, with a pool of different tasks that were planned out and performed on a day-to-day basis. Kanban is an agile project model with an emphasis on flexible planning and low documentation overhead.
2 Concepts, Theory and Technology

Here some necessary background information and explanations of key ideas and technologies involved in this project will be provided.

2.1 Remote Tower current implementation

The control software for SAAB:s Remote Tower system (henceforth referred to simply as RT) is distributed over several physical machines. These computers (henceforth referred to as control machines) each control a subsection of the RT, the airport and its subsystems. The RT application currently relies on the host operating system for getting mouse and keyboard inputs. As previously mentioned, this effectively limits the program to one set of user inputs at a time per machine. Furthermore, it requires one set of input peripherals per control machine. To bypass these limitations, RT uses the Synergy application to forward mouse and keyboard input from a single terminal operated by the air traffic controller (henceforth referred to as input machines) to all the connected control machines.

2.1.1 Synergy

Synergy is a mouse and keyboard sharing software published by Symless [1]. It lets users control multiple computers from one machine by sharing its input data over local networks. It follows a client-server pattern, where the computer with the active input devices is the server and the computers under remote control are the clients. Client machines connect to the server, which shares mouse and keyboard input seamlessly to its clients as if the server was one computer with multiple monitors. Synergy is used by SAAB to coordinate mouse input in the Remote Tower system.

2.1.2 Remote Tower Synergy integration

As previously stated, the Remote Tower system relies on Synergy to provide remote access to the control software. This setup makes it so that only one set of input peripherals is required to operate multiple control machines, but does not solve the problem of allowing several users control of the program simultaneously. Therefore two or more client instances of Synergy are run on each control machine, with each instance receiving input from the server on its respective input machine. This allows each control machine to be operated by several users, but it does not allow true simultaneous multi-user control, as the operating system (Windows 7 at the time of writing) only accepts input from one device at a time. This restriction can lead to erratic or undesired behavior when more than one user attempts to operate the same control machine at a given moment.
2.2 Relevant concepts and technologies

The following sections introduce some of the concepts and terminology used in this paper and their technological contexts.

2.2.1 Personal computers

Personal computer (PC) is a generic term for any computer whose size, price and features make it viable for personal use. The first personal computers were made available for consumers in the 1970s, and stood in contrast to the larger and much more expensive mainframe computers typically only used by corporations in the past. The personal computer quickly rose in popularity and was named the “Machine of the Year” by Time Magazine in 1982 [2]. Because of their convenient size and price point, the PC quickly became the most common type of computer, or at least what is often referred to when the term ‘computer’ is used. The personal computer has remained highly popular ever since its introduction, and today you can find one in most homes.

As the name suggests, the personal computer is typically operated by a single end user. They are designed to be easy to use and require no special technical skills, and usually have an attached monitor for visual feedback. Users interact with the computer by using a GUI (Graphical User Interface) or a text console [3]. Commands are generally issued to the computer via connected physical input devices such as a mouse and keyboard. The vast majority of all interactive computer systems have adopted this control scheme, since this is how most people are used to interacting with them.

2.2.2 Groupware and collaborative software

Groupware, also known as collaborative software, is a category of software meant to aid the work effort of a group working towards a common goal [4]. Examples could be communication platforms, shared calendars and other software that aids group processes and teamwork. It could also be software more directly related to achieving the actual goal, such as document editing software and other programs that allow a group to work together on a single task, file or dataset.

Groupware can be divided into synchronous or asynchronous programs. Synchronous groupware are programs where the users interact in a shared instance in real-time. The program states of all members are synchronized, and any action taken by one user is immediately visible to the others. Examples of this could be multiplayer games and online real-time document editing software.
Asynchronous groupware, on the other hand, is updated in discrete steps. Two examples of such programs are email clients and version control software. Asynchronous groupware often uses parallel instances; each user maintains a local copy of the work object, and changes are made only to the local instance. When appropriate this local copy can be synchronized and merged with the shared, master version. One potential drawback of this kind of software is that it can introduce divergence or inconsistencies between different users and their local states.

2.2.3 Graphical User Interface
A Graphical User Interface (GUI) is the visual representation of an application or system which facilitates interactions between user and computer [5]. The GUI is typically made up of graphical windows, buttons, icons, menus and dialog boxes, and makes the working environment more intuitive and gives it a more natural feel compared to text-based interfaces. Complex operations can be done by manipulating a small number of graphical objects, which greatly helps the user gain control and navigate through large, complicated software systems.

2.2.4 Operating systems, desktop environments and window managers
The operating system (OS) is the software foundation of any computing system. It is among the first programs to run on startup and encapsulates and protects core system resources. The OS provides access to critical hardware and software components of the computer, such as CPU cores, memory and networking, to applications as well as users. All programs launched by the users run on top of the operating system, and request access to different subsystems from the OS as needed [6].

Operating systems generally provide a desktop environment (DE) as a means for users to interact with the computer in an intuitive fashion. The term comes from the desktop metaphor, a common way of arranging computer systems to imitate an office environment with desks, files, folders, recycling bins etcetera [7]. This is the default mode of operation for the computer when no other applications are being run, and the first view that greets the user after startup. The DE is a kind of GUI that allows users to launch, view and control programs as well as monitor the computer system and perform various other operations. The main view is the desktop, a collection of icons used for providing quick access to system resources such as files, folders and programs. Desktop environments are generally two-dimensional, and often include or require a window manager to form a complete interface for the user.

Interactive user applications typically run in one or several windows, portions of the screen dedicated to providing a visual representation of the program. A window
manager (WM) is a program responsible for arranging these windows and presenting them to the user. It performs actions such as opening, closing, positioning and resizing of windows, among other things [8]. The WM also tracks window focus, which application the user is currently interacting with. Apart from window management logic the WM also provides tools for simplifying graphical application development, which results in a coherent visual style across the operating system for programs that rely on it.

2.2.5 Mouse & Keyboard input in UNIX and UNIX-like systems

One of the most common ways for humans to interact with computers today is by keyboard and mouse. The keyboard is a typewriter-like device which sends commands to the computer when the user presses a key. The commands are interpreted differently depending on which application the user is currently interacting with. The mouse typically consists of a set of buttons and a sensor for detecting motion and is used to control the operating system cursor. This is done by physically moving the device in the plane, which results in a corresponding movement by the cursor on the screen, or by pressing the buttons which triggers other context-dependent responses.

Upon connecting the devices, the computer registers them as HID:s (Human Interface Device). When the user presses the keys on the keyboard or moves the mouse, a string of raw data is sent containing the string of keys pressed or, in the case of the mouse, how far the mouse has moved in relative coordinates. The HID drivers format this data to make it more accessible and streams it to predetermined input event files [9]. Figure 1 illustrates a dump of the raw hexadecimal data from one such file in Linux, generated by mouse actions.

![Hexadecimal input event data](image)

Figure 1. Hexadecimal input event data

High-level applications continually read these files and interpret the input according to their functionality, or get their instructions from another application that does so.

2.2.6 Distributed computer applications

A distributed application actually consists of several subsystems running on separate machines in a connected network, all working together towards a certain goal [10]. This can be done for many reasons, for example to distribute the workload, make certain services or resources available from multiple locations or to improve runtime
reliability of the program by allowing services to move in the event of failure of one machine. Distributed applications on different machines pass information between each other to avoid keeping all data locally or having to perform all computations themselves.

2.2.7 C++
C++ is a programming language created by Bjarne Stroustrup as a superset of the C language, with added capabilities for objects and classes [11]. It is a compiled language with many features and strong library support, and it is also standardized by the ISO. A plethora of C++ compilers exist for many different systems and platforms due to its popularity and widespread use [12]. This allows programs written in this language to be run on most machines which makes it highly portable. While often considered a high-level language, it can at the same time be considered relatively low-level compared to other languages such as C# or Java, due to its roots in C and the low-level operations it provides. Because of these features and the control it grants the programmer, it is possible to create fast and well-performing programs in C++ given that they are otherwise properly optimized.

2.2.8 Qt
Qt is a software development framework for open source as well as commercial development, first released in 1995 and currently published by Nokia [13]. It is one of the leading cross-platform frameworks for creating graphical applications and user interfaces and is used by many prominent companies in a wide range of industries [14]. As well as providing tools for graphics and UI creation, Qt also natively extends the C++ language by offering generic programming libraries and APIs for networking, I/O operations, data storage and more [15]. Because of the wide range of functionality it provides, Qt is a suitable choice for an all-in-one solution to building interactive user applications.

Two of the key features of Qt are its event and slot/signals systems. These are different strategies for connecting and passing information and function calls between objects and classes. Slots behave similarly to functions that are called when their respective signals are emitted at runtime and can take a set of parameters. Events are packets of information about some system or program event that can be passed to different objects in the program to cause a response. Specialized event types inherit this class, such as QMouseEvent for representing a mouse click or movement. Widgets and other objects that respond to these events implement their own event handlers for
events of all relevant types. Qt:s event loop continually sorts through incoming events and dispatches them to their intended recipients during runtime.

### 2.2.9 Raspberry Pi/Raspbian

Raspberry Pi is a small and popular single-chip computer with most commonly used I/O interfaces such as USB and Ethernet. The Raspberry Pi comes in several versions with different hardware specifications and features. It is capable of running a full modern operating system, including many popular Linux distributions [16]. Raspbian is a Debian-based Linux distribution made specifically for the Raspberry Pi.

### 2.3 Shared and personal computer systems

This section will introduce some terminology and its relevance to the core ideas presented in this paper. Note that some of the classifications and terminology is proposed by the authors for this particular context, and is not necessarily widely adopted in the software industry or congruent with other definitions or interpretations.

The dominance of the personal computer paradigm is reflected in software design; most systems are built to be operated only by a single user at any given moment. More than one user attempting to control the system simultaneously will typically result in unpredictable or erroneous behavior. These kinds of applications and systems will be referred to as personal for the remainder of this paper.

In contrast to these personal systems, the authors of this paper recognize an alternative category, namely systems capable of being operated by multiple users concurrently. These could be local users controlling the same instance of an application directly on the same machine. It could also be geographically separated users operating in a distributed version of the system on separate computers, each controlling a shared instance via networking. These kinds of applications and systems will henceforth be referred to as shared, and software of this kind can be considered synchronous groupware.
3 Analysis

This section will cover how the problem area was analyzed and methods and strategies that were considered for development.

3.1 Methods for shared application interface development

As previously mentioned, not many shared computer systems exist that are suitable candidates for deployment of large-scale enterprise software solutions. Thus, any shared application must likely target a personal computer system. One of the biggest challenges with writing shared applications for such systems is binding user input to system events and providing an interface that lets users interact with the program. In some cases, the underlying operating system scheme for input management may be sufficient, but in others it might be necessary to bypass this entirely and implement a completely custom solution.

Many possible patterns exist for implementing such functionality, and because shared applications are relatively rare there is no obvious best-practice to follow known to the authors. For this reason, several development methods must be considered. Three such methods that have been deemed viable alternatives by the authors will be proposed and later evaluated.

3.1.1 Tailoring - Application specific custom implementation

The most straightforward and least sophisticated approach is coming up with a custom solution tailored to the specific requirements and properties of the system under consideration. This means specifying the bare-minimum interface of user actions that make up the necessary functionality of the program and implementing each of those actions directly in said program. In a typical GUI program, this could be actions corresponding to left mouse click, drag-and-drop etcetera.

One of the key properties of this approach is that it scales with the program scope. For simple applications with a limited set of required operations, this interface could be relatively simple to implement. This is a potential advantage for programs that don’t require a broad set of sophisticated user interactions or that are unlikely to change a lot over the course of its life cycle. This approach is also highly flexible because all code would be tailored to the specific needs and constraints of the program in question.

However, a major downside is that coupling between the defined user actions and application-specific objects and control logic is very high, making the code difficult or impossible to reuse. Another major disadvantage is that code maintainability is lower,
and that adding functionality to the program might require expanding the interface and adding completely new implementations or changing existing code. For these reasons, this approach is considered by the authors to be suitable mostly for small, simple projects that are unlikely to evolve a lot after initial development.

3.1.2 Adaptation - Integration with external frameworks
Several frameworks exist for building GUI-applications, each with their own scheme for handling system events like user input and window management. Two examples are the Qt and GTK+ frameworks for C++. It might be possible to adapt such a framework to provide support for application sharing while at the same time relying on its underlying structure for input handling and other runtime events.

Such frameworks have their own design philosophies and rely on different strategies for handling system events, so the specifics of such an adaptive solution would depend heavily on the chosen framework. The viability and overall quality of the solution are difficult to assess beforehand since it depends so heavily on the existing foundation. The development effort required is similarly difficult to assess without knowledge of the specific implementation in the framework in question. Adapting the framework to support multiple concurrent users may be trivial, or perhaps even require more effort than building a new framework from the ground up. As such, the outcome of this approach is hard to predict before the start of development and is heavily dependent on the chosen framework as well as the developers understanding of it.

However, if done successfully adapting an existing framework could prove to be highly beneficial. Many frameworks already provide extensive functionality when it comes to creating graphical applications, and full integration with such a framework would simplify development significantly. As previously noted it is hard to assess the effort required, but in the event of successful integration it’s likely to yield good returns in terms of functionality provided in relation to development effort. If such a solution is to be attempted, the chosen framework should be carefully studied before development starts to avoid wasted effort.

It is the opinion of the authors that this is a suitable option for extending the functionality of an existing program that already relies on a suitable framework, or building a new program when it’s unlikely that other similar projects will be undertaken in the near future. In such cases adapting an underlying framework may prove to require less effort and be more cost-efficient than devising a new scheme entirely.
3.1.3  **Innovation - Custom input handling and UI management scheme**

The design of most operating systems is heavily dependent on the old personal computer paradigm, and this philosophy is reflected in many GUI frameworks as well. Most computing environments are designed primarily with personal applications in mind. For this reason, it might be necessary or desirable to create a new scheme for capturing and handling user input as well as binding this to actions in a graphical application. This offers the possibility of providing more robust, flexible and innovative interfaces for interacting with a program without the need for clumsy, retrofitted solutions.

A customized scheme for handling user input can vary widely in scope. For example, it could be generating system events based on user input, dispatching these events to the appropriate part of the program and implementing a context-specific handler for these events. This could then be integrated with an existing framework for building graphical applications, in order to only switch out input management and user-program interactions. A more complex approach might be to create a custom framework for input and system event handling, window management, and GUI creation. This could be done by switching out parts in an existing framework or by creating one completely from the ground up. Another approach might be to create a desktop manager or even a whole operating system with more robust support for multiple concurrent users and targeting this system with new applications.

The scope of such a custom solution should match the requirements and scope of the intended program or system. More effort spent on the foundation of such an implementation could result in more maintainable and development-friendly code as well as provide more powerful features. However, it will probably also be more costly up-front in terms of development resources. While it likely requires the largest initial investment, this solution offers the greatest possibility of valuable returns. Being more portable and general than the previously outlined approaches means that this scheme could be used in many future programs and reduce the development cost of further projects. It might even be licensed and sold as a stand-alone product. Therefore, it is the opinion of the authors that this method is most suitable for large projects or organizations and programs with long expected life-cycles.

3.2  **Method evaluation and implementation**

Three different approaches for creating shared applications have been proposed. To evaluate these approaches the first two (Tailoring, 3.1.1 and Adaptation, 3.1.2) were
utilized in the implementation of the prototype program as requested by SAAB for the Remote Tower project. The third method (*Innovation, 3.1.3*) was deemed too complex and time-consuming to attempt given the time and resource constraints of the project.

The *Tailoring* approach was used to create a minimum viable product to demonstrate required program features and to improve understanding of the problem area in general. Knowledge gained from this experiment was then used for attempting a more general solution by altering the underlying framework used according to the *Adaptation* method.
4 Prototype Design

The following sections will introduce the central system design and architecture. First, an overview and a brief explanation of the system will be provided. Each part of the system will then be described in greater detail, and the reasoning behind design choices elaborated upon.

4.1 General system design

This section will describe the overall system design as well as the tools and frameworks upon which it is built. Two versions of the software were created by following the methods tailoring and adaptation as outlined in the previous sections of this paper. However, the overall system design of both versions is very similar. Therefore they are presented as one version, and their differences will be described in the Server subsections where they appear.

4.1.1 Design overview

Based on the requirements outlined in a previous section of this paper it was decided that the system should consist of three separate parts, corresponding to the previously identified logical parts in section 1.3 of this paper. Figure 2 illustrates the basic system architecture.
The first part is for capturing and forwarding user input from connected input devices, represented by input clients and devices in the diagram. The second part consists of a server responsible for receiving input data, interpreting it and managing program state and interactions between different objects. The server is also responsible for publishing the program state to connected output clients. The output client is the third and final part and is responsible for receiving program state data from the server and making this visible to a human user by displaying it on a connected monitor. As seen in Figure 2, information only flows one way, upwards in the conceptual program stack or clockwise in the diagram.

An instance of the server should allow multiple input and output clients to be connected simultaneously. A connected input client is represented by a mouse cursor object that can interact with other objects in the program. The cursor is controlled by the user’s mouse device. Objects managed by the server are visible in the connected output clients. These clients are assigned a portion of the program’s workspace to display to the user, that can be a part of or the entire workspace. When multiple output clients are connected and displaying separate parts, the entire system should behave similarly to an extended desktop for the user. The key differences are that multiple users can be connected and active in the same region of the workspace simultaneously and that the system is distributed on separate machines.

4.1.2 Distribution and deployment
The entire system consists of several instances of three separate programs deployed across several Linux and Windows machines. One or more Raspberry Pis running Linux are responsible for capturing user input with a connected mouse. User operations are coordinated in a server program that runs on a single Windows 7 machine. A partial and static copy of the server state can be displayed by running an output client on a Windows 7 machine and connecting to the server. This client mirrors the current server state but can not make any changes to it. Thus all program interaction logic is contained in a single point in the system, while input and output are distributed over as many machines as needed.

4.1.3 Tools and frameworks
The entire system is written in C++ due to it already being heavily used in the RT software, as well as being a powerful and efficient programming language. The UI components utilize the Qt framework due to it having native C++ support and being heavily adopted in the industry as well as at SAAB. Since C++ does not provide a native networking API, the networking components of the program rely on the Asio
library. This makes the programs more portable than C-style programming of the network sockets directly through the API:s provided by the operating system. Asio was chosen due to being one of the more well-regarded and frequently used third-party networking libraries available.

In the following sections, the design of each part of the system will be explained in greater detail.

4.2 Input
Here the part of the system responsible for capturing user input, formatting it and sending it to the server will be described.

4.2.1 Deployment
This section describes the hardware and software environments to which the input clients should be deployed and the reasoning behind these choices.

It was decided that the input clients should be set up on Raspberry Pi:s running the Linux distribution Raspbian. Since the software was not expected to be particularly performance demanding a small and cheap microcontroller was deemed a sufficient and appropriate platform. Running the input client on a desktop computer is wasteful in terms of hardware price, physical space and power requirements. Deploying the input clients on separate computers was deemed desirable for several reasons. Primarily, the system becomes more reliable due to having multiple points of failure instead of one. In the event of a failure in one input client, others can continue to function correctly. It is also convenient to deal with only one set of input peripherals per computer for programming purposes.

The Raspberry Pi was chosen due to being relatively cost-efficient and easy to develop for, as well as being the most convenient option. It provides all the necessary I/O ports and is a convenient microcontroller for software development. It can run a full operating system environment and is easily accessible either by connecting input peripherals and a monitor or by running in ‘headless mode’ and connecting over SSH from another machine. The Raspberry was also a suitable personal choice due to the authors already having experience with developing for this platform.

Linux was decided on as the operating system due to being performance efficient and developer friendly as well as allowing easy direct access to connected input devices.
Raspbian was chosen due to being a fairly lightweight Linux distribution suitable for running on a Raspberry Pi while providing all necessary tools and utilities.

4.2.2 Software design
This section describes the general design of the input client software. The simple nature of the program is mirrored by a similarly simple design; the program consists of four distinct parts. The first part is responsible for reading input device data from its respective HID event file provided by the Linux kernel. The second part is for formatting this data, appending some necessary information and serializing it for transmission to another machine. The third part is responsible for network communication and data transfer to the server. The final part is the logic for coordinating these operations and managing program state and execution.

Input data packets should be small and be transmitted continuously and immediately to reduce noticeable input delay for the user. Mouse movements should be transmitted as relative movements on the X or Y axis rather than the absolute position of the cursor. Button presses/releases are encoded and sent one at a time for the sake of simplicity and clarity. Relative movement was chosen for two reasons; packets arriving out of order will still result in the same total movement on the server and not introduce noticeable inconsistencies or rubber-banding, and it makes the server implementation more decoupled from the input client since it is independent of any coordinate system.

Since the data packets should be transmitted as soon as they are generated UDP was deemed a more suitable choice than TCP. TCP buffers data and sends it in intervals as larger packets, rather than sending it immediately. Packets are also acknowledged by the recipient upon arrival. This means that while TCP throughput may be high, the delay for any given packet is also higher than with the UDP protocol. Because input delay should be minimized UDP was chosen as the transport protocol.

4.3 Server
This section will elaborate on the design behind the server application. At the time of writing the Remote Tower software runs on Windows 7 desktop computers, and therefore this was chosen as the deployment environment for the server application as well.

The server software is divided into four components with corresponding areas of responsibility, as illustrated in Figure 3.
These four components are responsible for networking and passing information between machines, parsing received input data and translating it into user actions, managing and updating a shared workspace with cursors and other objects, as well as distributing the program state to output clients. The server is the only part of the system where the design differs between the two software versions developed. The user management and workspace components vary somewhat in how system events are handled and how information is passed between them.

The *tailoring* version handles user input by generating a sequence of signals that correspond to different user actions. Interfaces define the different operations that can be performed on an object, and each object must implement the actions it recognizes. A manager object listens to mouse signals and calls their corresponding functions on the target objects as required.

The *adaptation* version of the server attempts to build a foundation for a multi-user application by adapting the Qt framework to this purpose. This is done by parsing input data, manufacturing event objects according to Qt:s event system and dispatching them through Qt:s event handlers. This is conceptually similar to how system events are handled when relying on the operating system for getting user input in Qt applications. The design of the server components will be described in greater detail in the following sections.

### 4.3.1 Networking

The networking component is divided into two parts. One part listens for incoming connection attempts from clients, performs a handshake and starts a new session. The other part manages and maintains this session with a single client and passes data back
and forth. The session also passes received data forward to the user management component.

### 4.3.2 User/input management

The user management component is responsible for managing active users in the program. Each user receives input data from the associated network session, and this component then parses this data and forwards it as user actions to the workspace. Movement data results in moving the user’s cursor, while button presses ultimately perform different context-sensitive actions on nearby objects in the workspace. The interface for binding user input to a program action differs between the two versions of the software, and thus the dependencies between the user management and workspace components of the program also deviate from each other. The design of these components was not decided beforehand; rather they evolved over the course of development. Specific implementations of these two different strategies are covered in the implementation section of this paper.

### 4.3.3 Workspace/program state

The server manages a canvas representing the shared working environment with cursors and other objects. These objects exist as geometrical shapes with location coordinates in a 2D-plane. The workspace component is responsible for managing this canvas and the objects contained within. It contains all the logic for object behaviour and interactions between them. This means that all user actions are defined in this component. It also updates the graphical objects according to their predefined behaviour and performs the actions supplied by the user/input management in real time.

### 4.3.4 State distribution

The final program component is responsible for making the current program state known to connected output clients. Each client is assigned a portion of the whole workspace, and objects in this space and their states are encoded in a specific structure, serialized and transmitted to the client via the networking component. Object coordinates are transposed before transmissions so that each client’s coordinate system begins at 0,0. Thus clients do not require any information about what part of the workspace they are displaying, and the server is responsible for assigning each client a section of the workspace.
4.4 Output

The output client consists of three parts; a networking part for receiving object data from the server, an object manager for parsing object data, creating and updating objects accordingly and managing their states, and a graphical component for displaying the program state to the user. The client is designed to be run on Windows 7 or higher in order to be consistent with the server.

The networking part performs a handshake with the server and then begins to receive object data according to the predetermined encoding format. This data is then passed along for parsing. The object manager maintains a set of object types corresponding to the objects in the server workspace. Upon receiving object data, the manager either creates or updates the corresponding object. The graphical component - which relies on the Qt graphical framework - then places this object in a coordinate system and renders it for output to the user on a monitor.
5 Prototype Implementation

The following sections describe the two prototype versions created and the different programs of which the system consists. The input and output clients are identical in both versions of the software, while the server implementation differs somewhat. Each part of the system implementation will now be described in greater detail.

5.1 Input

The program for capturing user input gets events continually from mouse and keyboard by reading a struct - input_event, as defined in the Linux header linux/input.h - from their respective event files. Relevant data is extracted and packaged with an extra byte for encoding what type of device generated the event. The resulting object has the following members:

```c
unsigned char device;
unsigned __int16 type;
unsigned __int16 code;
__int32 value;
```

*Type* specifies the kind of action taken, and could be mouse movement or a keypress for example. *Code* and *value* have different meanings depending on the event type. For a keypress *code* corresponds to the key or button pressed and value the button state, i.e. ‘1’ or ‘0’ for the button being either pressed or released. For movements, *code* corresponds to the axis (X, Y, Z on the mouse) and *value* the distance of the movement, positive or negative. The encoding of these members is defined in a header file that corresponds to the Linux header linux/input-event-codes.h. The newly created object is then serialized and sent to the connected server.

5.2 Server

The two different implementations of the server will be described here. As noted in the Design section, much of the implementation is shared between both versions. Components for networking and state distribution are identical and are also not particularly relevant to the domain of user interaction. Therefore these components will not be described in detail. The common aspects of the server will now be described before further explaining the implementation specific details in their respective subsections.

A listener thread runs in the background and accepts new connections from clients. The maximum number of connected clients is set to five by default, but this can be changed according to need and hardware. Each connected user exists as an object and
has a networking thread for receiving input events. These events are then forwarded to a user manager object for parsing and notifying the program that some user action has taken place. See sections 5.2.1 and 5.2.2 for implementation specifics on how user actions are propagated and handled. The program state is shared with connected output clients at regular intervals by sending metadata about the objects maintained in the workspace. This data contains object type, geometry, positions and a unique identifier for each item, and is sent to each connected client. Each client is assigned a portion of the whole scene, and item coordinates are transposed by vector subtraction to the client’s local coordinate system before transmission.

5.2.1 Tailoring - user interactions

The Tailoring implementation of the server relies heavily on the Qt framework for graphical scenes and items. User cursors, as well as all other interactive components of the program, exist as graphical items managed in a single QGraphicsScene. This approach deviates somewhat from the typical way of building GUI applications in Qt, where QWidget structures are generally employed to provide interactivity to the user. Custom input management is implemented by relying on slots and signals for calling functions directly on items in the scene. User actions are implemented by each item in functions specified in different interaction interfaces.

When an input event reaches the server, this results in a series of signals that are emitted and ultimately ends with a function call on the targeted item. When the user thread receives an event, it calls a function for decoding it and passes along the event data and its unique user ID to the object responsible for user management. The event is then unpacked in a series of conditional statements, and a signal that corresponds to the nature of the event is generated from the user object. This signal is then routed to the cursor item that represents the graphical cursor as well as its position in the program workspace.

The cursor updates its position if necessary, and otherwise emits its own signal corresponding to the action taken. Signals corresponding to key presses and releases are routed to a program module object that coordinates interactions between items in the program. For instance, when the left mouse button is pressed the cursor grabs all items under it and calls a click()-function if the object implements it. Otherwise the action is ignored. The following figure shows the specific calls and signals in such a case.
Each cursor object can emit five different signals. One signal for button pressed and one for button released, for the left and right mouse buttons respectively. The final signal is for mouse move events. The program module is responsible for managing the workspace state and item interactions. It also implements the reactions to signals involving mouse buttons by calling an associated function on the items at the location of the event. The target object implements the resulting action in a specific function, if that action is recognized by the object. The cursor itself implements the response to mouse move events from the user. It also generates a signal that other objects can connect to in order to follow the movements of the cursor.

The set of possible user actions is limited. Objects can respond to the five signals emitted, as long as the response to such a signal is implemented. Objects can be clicked to activate them, or dragged to move them. Mouse cursors automatically track moving objects by temporarily migrating to the object in question’s local coordinate system. As stated previously, keyboard events are supported but not implemented in this scheme and thus are simply discarded upon decoding by the user manager. If keyboard support is desired the corresponding keyboard signals should be added as well.

5.2.2 Adaptation - user interactions
This implementation of the server relies on Qt's widgets and event handling. When event data is received the user manager creates a new QMouseEvent object and passes it to the program module. Depending on the event type and location the program
module then dispatches the event to the appropriate widget. This process is illustrated in the following figure, and described in greater detail below.

![Diagram showing event flow and function calls in Adaptation prototype.](image)

Figure 5. Event information flow and function calls in Adaptation prototype.

When the program module receives a mouse event, it extracts the coordinates and identifies the window in which it occurred. It then retrieves the widget at the position of the event from said window and dispatches the event to the widget. This is done with a call to the static function `QCoreApplication::sendEvent()` which in turn calls the protected `event()`-function on the widget with said `QMouseEvent` as an argument. If the widget implements a handler for the event type in question, it will respond depending on the widget and event types.

5.3 Output

The output client initiates a connection to the server and then waits to receive item data. Visual output and rendering rely on the Qt framework. A set is maintained of all items currently added to the scene. Upon receiving item data, an attempt is made to add the item to the set. If it already exists the item state is updated, and if not it is added to the program. Items then draw themselves using Qt's QPainter tools for rendering. As such, the output client has no awareness of object interactions and is merely a partial mirroring of the server state with respect to object shape and positioning.

5.4 Evaluation

In this section, the two prototype programs will be evaluated and compared.
5.4.1 Tailoring prototype

The prototype performs according to requirements and specifications. The required user actions, clicking and dragging objects, as well as all other requirements are implemented. The cursor automatically tracks moving objects that the cursor enters. However, the set of possible user actions is restricted. Actions are defined in interfaces, which means that making hybrid objects or minor alterations is potentially cumbersome and requires code duplication or reimplementation of basic functions. Due to the syntax for Qt:s slots and signals it can be hard to follow the execution path and function calls of the program, which makes it harder to get an overview of the system.

As previously mentioned, all objects and interactions must be implemented from the ground up. Therefore many common UI features that are typically provided by Qt:s standard widgets are currently unsupported. For example, visual feedback when interacting with UI elements - such as button click animations - must be implemented directly.

In summary, the prototype performs well within its intended domain. However, the scalability and maintainability of the program are relatively poor, and further development is projected to be costly or troublesome. Therefore, it is the opinion of the authors that this prototype is not a suitable basis for further development.

5.4.2 Adaptation prototype

This version of the prototype relies more on Qt:s libraries for GUI development. This means that a vast collection of widgets and other features is available for use in the program, which can potentially save a lot of development resources. However, the integration of Qt:s event and widget systems in the prototype is less than optimal. Certain actions are supported while others are not, and some widget behavior is inconsistent, unreliable or completely broken.

In theory, any widget should respond to any event in a similar manner as though it were generated by the operating system’s mouse and keyboard, since Qt:s own widgets and events are used. However, when using the local OS:s input devices several other events are generated as well, many of them relating to window focus, mouse grabbing and other phenomenon that rely on having only one set of input peripherals. Certain widgets and features depend heavily on these auxiliary events, and therefore some widgets do not respond as expected to user actions. It is possible to either
generate these auxiliary events or override or reimplement event handlers to bypass them. However, this is a daunting task that partially or completely negates the advantages of relying on this framework in the first place.

Delegating input events to the intended recipient widget can be challenging. Events are generated by the cursor, and the cursor itself is a graphical object existing in a confined geometric space. Newly generated events should be dispatched to widgets at the cursor’s position. This means that the program requires a consistent and global set of coordinates for all objects in order to forward generated events to the widget at the location of their origin. This introduces some difficulties and constraints when it comes to the layout of the application.

In its current inception, the adaptation prototype provides limited functionality for a fairly high development cost. Despite this, it is the opinion of the authors that it could provide a suitable foundation for development given some refactoring and alterations to the core design. Achieving complete integration with Qt’s widget and event frameworks without reimplementing some parts is thought to be unlikely, but it is estimated to require less effort than developing equivalent components from the ground up.
6 Discussion and Conclusions

This section contains our thoughts on the subject of shared applications for multiple users and our experiences with developing such applications.

6.1 The personal computer problem

In today’s corporate environment work is often carried out in teams. Cooperating towards a shared goal brings many benefits like diversity of opinions and perspectives, complementary skill sets and ideas, and makes it easier to identify and mitigate potential problems on the way. In such an environment where work is often shared, it makes sense to also have shareable digital tools that facilitate cooperation and teamwork. For example, this could be software to let multiple people collaborate in real time on design, product development and planning tasks etcetera. We call this shared applications or shared computer systems.

Such tools exist, but they are often created as an afterthought or retrofitted on top of programs that were originally intended to be operated by a single user. This can make them cumbersome and unfriendly to the user, to the point where their adoption sometimes gets in the way of what people should really be doing: working together on what they are truly passionate about.

There is, of course, a reason for this state of affairs. It is the conventional approach to software development, as most users are accustomed to the traditional design philosophy of one user per computer or application. We call these personal computer systems. Many operating systems are firmly based in the personal computer paradigm and rely heavily on the assumption that only one set of input is provided from one user at any given time. This strict one-to-one relationship between computer and user can almost be considered a fundamental maxim in most computing and software design contexts.

Both the Mac OS and Windows families of operating systems are based on this design philosophy, as well as most prominent Linux distributions and many others. Since most applications rely on the operating system or its desktop environment for input, the vast majority of all programs, API:s and development tools also adheres to this scheme, either due to convenience, conformance or tradition. For this reason, many of the existing tools for building GUI:s and other interactive applications can prove to be cumbersome or unsuitable when developing programs for multiple users.
We believe that to efficiently develop shared applications a shared computer system is required as the target platform. By this we are referring both to the actual hardware platform as well as the operating system and desktop environment it runs. Developing truly shared applications for personal computer systems is difficult and time-consuming, and integration with the host system is often poor. The following section contains our experiences from attempting this.

6.2 Shared application development

Developing shared applications is potentially much more complex than their personal counterparts. The task of adding another set of input peripherals and connecting them to an additional user may seem trivial, but in practice quickly proves to be more challenging. While the practical applications for our prototype software are somewhat limited, the development process provided valuable insights and knowledge into writing shared applications. The following sections contain our reflections on some of the challenges faced and other lessons from the development of our software.

6.2.1 Locally and virtually shared systems

Most groupware programs, even synchronous ones, are still personal applications in the sense that each instance is meant to be operated by a single user on a personal computer. When creating a shared application some kind of networking and synchronization structure is typically used to connect multiple separate instances of personal applications together. This creates the illusion of a single unified instance to the users and makes up a kind of virtual cloud application. We call these virtually shared systems, since they are stitched together from several personal systems. This design pattern introduces significant overhead in terms of complexity and development effort, as well as dependencies on resources such as hardware and networking.

An alternative to such virtually shared programs could be locally shared systems. This means multiple users interacting directly with a single instance of an application contained within a single computer system. To clarify, we do not mean local in a geographical or networking sense. Rather, this term is referring to the fact that the application and all of its users are contained within a single program instance with little to no external dependencies or connections. Concretely this might consist of multiple sets of input devices connected to a single physical machine, operated by several people at once. Some video game consoles are examples of this kind of locally shared computer system. The differences between virtually and locally shared systems are illustrated in the following figure.
Figure 6. Virtually- and locally shared system instances.

Shared applications offer some potential benefits over personal applications, however, as previously stated they are more complex and possibly require more effort to develop. Developing locally shared applications would eliminate or significantly reduce the overhead compared to their virtually shared counterparts, given the existence of robust development tools and platforms suited to the task. However, this kind of software is very difficult to develop for personal computer systems because of their previously outlined limitations.

Many of the difficulties with developing shared applications arise because of fundamental differences in the design philosophies of shared and personal computer systems. They are often fundamentally incompatible with each other from the operating system level and up. Developing a locally shared application ideally requires a locally shared computer system as the target platform, but at the time of writing no such systems exist that are widely adopted and suitable targets for deployment.
Our prototype software is a kind of hybridization of these two different types of shared systems. While the system as a whole is virtually shared, the control structures in the server are completely agnostic as to the source of the input. Thus it is theoretically possible to connect multiple input devices to the same instance of the application on a single computer and run it as a true locally shared application. However, as the operating system does not easily allow input capturing from multiple mouse sources we simply route the input from different machines for convenience and simplicity.

The following sections contain our thoughts on some specific problem areas of shared applications and our prototype development.

6.2.2 Input forwarding
The chosen strategy for capturing and transmitting user input proved to be sound and resulted in a well-performing software module as well as a solid foundation for the entire system. Formatting of input events is relatively intuitive and easy to read from a developer perspective, and familiar to anyone who has worked with raw Linux input before. No noticeable input delay is present in the prototype implementations, nor were there ever any problems with lost or unordered commands during testing. The user experience is pleasant and responsive, and more or less indistinguishable from using a local, directly connected mouse. Actual program functionality linked to the input is realized entirely on the receiving end; therefore the user experience is largely dependent on the server implementation. However, the formatting of input data is flexible enough to achieve completely agnostic input handling on the server end.

It should be noted that this module has only been tested over local networks, and that performance might be degraded over longer physical distances. While our particular system is only intended to operate locally, it is possible to imagine scenarios where transmitting input over longer distances is desirable. In these cases, the protocol may need to be expanded to compensate for longer propagation times or unordered packets. This should not pose any major challenge, however, as the foundation of the protocol is light-weight and performance friendly, consisting of only the most essential parts.

6.2.3 Input handling
On the receiving end of the input event pipeline, some type of input handling layer needs to be implemented. By this we are referring to some construct that provides the
interface between a physical action by a user, such as moving a mouse or pressing a button, and a specific operation in a specific computer program. Ideally, this interface should form a universal middle layer between the human user and whatever application with which said user wants to interact. This layer should avoid strong coupling with application-specific operations in order to be portable and reusable as well as easily reconfigured. What this might mean in practice is that rather than linking the left mouse button directly to Action X in Application Y, the button is bound to some system or API event that in turn is linked to Action X in Application Y. Under these circumstances, programs can connect to this middle layer rather than needing to handle the raw input below it, which makes development easier and more uniform across all platforms.

Typically, such an interface is provided by the operating system or some other subsystem on which user applications rely. The component that is responsible for input management is typically also responsible for window and GUI management since these two areas are heavily dependent on each other. For example, this is the case for the X Window System as well as many higher level GUI development frameworks such as Qt. Swapping out either the input or window management part could then be difficult, depending on how closely they are coupled. In certain cases many of the more frequently used such middle layers are not suitable for the needs of the intended program, such as in the case of shared applications. As we have discovered during development, constructing this layer can be quite challenging in these cases. Due to the strong coupling between window/GUI management and input handling these components might require developing in tandem to produce an intuitive and well-performing solution.

6.2.4 Distribution

This section will discuss some of the necessary considerations when developing interactive distributed applications as well as our chosen distribution strategy for the development of the prototype programs.

There are many different ways of distributing an application and no clear-cut rules for how it should be done. Ideally, the strategy used should fit the restraints and demands of the specific software while also being as agnostic as possible when it comes to the actual deployment of resources in the system. One of the biggest questions when it comes to distributed applications that allow multiple users to interact with them is how to assign responsibility for maintaining and changing the program state. The program state needs to be synchronized across all machines and updated frequently.
enough to appear coherent and consistent to all users. The application also needs to deal with potentially conflicting commands from different users without introducing any errors or inconsistencies. It should also be error-proof in the sense that runtime failures in one location should not cause critical errors in other parts of the program. In the event of failure in a system component or resource it should be capable of recovering or migrating elsewhere in the system hardware. At the same time, the distribution scheme should not introduce unnecessary redundancy or computational overhead.

While planning the system distribution, we opted to spread input and output across multiple machines while keeping the responsibility for program state and interactions in a single point in the system on a single physical machine. This greatly reduces the complexity in synchronizing the visible program state across the system due to only ever keeping a single master instance of it. Minor inconsistencies may manifest at any given moment where the output clients fall behind the server due to propagation delay, but there is no risk of long-term divergence or conflicting program states under normal circumstances.

However, the choice to maintain the program state in one master instance negates some of the reliability benefits of distributed software since it introduces a single point of failure to the system. In the event of a server crash both the input and output clients will be completely disabled, and the program state, unless backed up, will be lost. This strategy can also be suboptimal from a performance perspective as it places a disproportionate portion of the workload on a single point in the system, rather than spreading the work out evenly across all machines. This creates a potential bottleneck and increases the risk of network or CPU congestion. Since the scope of our prototype software is small and its primary purpose is to investigate user interactivity rather than performance, we decided that our distribution scheme was an acceptable compromise given the limited development resources.

6.2.5 Qt and general development considerations
As previously mentioned, Qt was the framework chosen for the attempt to implement the Adaptation method. This section will evaluate Qt as a potential fundament for shared applications based on experiences from our prototype development. We will also discuss some necessary considerations when establishing a foundation for continued development.
The one-to-one personal computer paradigm is clearly reflected in the Qt framework and its core design principles for developing GUI applications. All mouse input is expected to come from the same source, and only one conceptual mouse exists in the software structure; thus there is no built-in way of conveniently handling input from multiple sources. One of the problems with this approach becomes evident in the example of drag-and-drop actions - when the user clicks an object, moves the mouse and then releases the mouse button, typically to move the object on the screen. In the default Qt implementation of this use case the clicked object temporarily becomes the mouse grabber, meaning that all future mouse actions are directed solely to this object for handling.

If multiple users with multiple mouse devices are active in the program at the same time, this leads to conflicts because the source of the input is not considered. If the first user clicks an object with the intention to move it, and the second user moves his mouse during this time, the second user’s mouse movements would be directed towards the clicked object and result in undesired and incorrect movements. The object should of course only respond to mouse movements from the same user that clicked it. However, all mouse events are indistinguishable from each other in Qt regardless of their origin and are directed towards whichever object currently is acting as the mouse grabber.

This dependency on the assumption of only having one input source can be circumvented, but it requires some significant reworkings of core Qt components. The structure of events and their surrounding infrastructures, such as event propagation and handling, has to be changed to make multiple concurrent foci possible. Additional information about the source and the user that generated it needs to be included with the event object, and this information needs to be considered when handling the event. A new scheme for spontaneously generating and delegating input events needs to be created since the current solution does not scale to multiple users. Changing event types, propagation or both may also necessitate reimplementation of some UI widget event handlers, which can be an extensive task due to the large variety of widgets Qt provides.

A permission/privilege hierarchy can or should be introduced to resolve cases of conflicting operations, and to introduce some degree of order to what may otherwise devolve into somewhat chaotic interactions between multiple users. Cases of conflicting commands that might otherwise result in ambiguous or undefined behavior - such as two users attempting to move the same object in two different
directions - can then easily be avoided, for example by giving one user elevated privileges and automatically overriding all other conflicting commands.

Ultimately Qt is a powerful framework with many useful tools for developing interactive applications. However, it is clearly rooted in the personal computer paradigm, and as such may require some major alterations to be used for applications with multiple users. This is a feature it likely shares with many similar development frameworks, but it is also possible that alternatives exist that are more suitable when developing applications for multiple users. It should be noted, however, that we have not studied the Qt source code extensively, and it is possible that solving or mitigating some of the problems we encountered is easier than we have estimated. How Qt measures up in comparison to other frameworks in this respect is a subject for another discussion.

6.3 Conclusion

In this paper we have attempted to outline some of the potential benefits of shared applications, and some common obstacles on the way to implementing them. We believe that programs with better integrated and more intuitive support for multiple concurrent users could increase utility and productivity in many situations, both for professional and personal use.

For practical and traditional reasons most computer systems are built around the assumption that only one user will control them. Many software development tools mirror this design choice, which makes developing applications for multiple users more difficult than for a single user. Even with better software development tools, multiple input sources are often unsupported by the operating system. This makes the realization of shared applications impractical on many levels under most circumstances. To be able to develop better applications for multiple users we believe that the following three things are required:

- A universal method of capturing input from several sets of devices
  - This could take place at the operating system level or through some service that is available for user applications to connect to.
- A window or program manager to coordinate user input and bind it to specific actions in the currently running program or programs
  - Should not rely on the assumption of needing only one set of inputs.
  - Could be integrated directly into an operating system or run as a container application inside of the operating system.
○ Should preferably be agnostic to the source and nature of the input device used, so as not to introduce limitations if other control schemes than mouse and keyboard are desired

● Software development tools to provide a flexible, intuitive and developer-friendly way of constructing applications for end users to run on top of the hereto suggested layers
  ○ These tools should have shared application development in mind and constitute a practical and preferably unified alternative to the personal computer paradigm as we have described it.

Together these three components make up a significant chunk of the user interface and experience of an operating system. For this reason it may be reasonable to package them as a complete desktop environment and include it either in a new operating system or as a modification to an existing system, together with accompanying development APIs. If this is done we believe it could greatly simplify and improve the entire realm of shared applications, both for developers and end users.

6.4 Future Research

There is much work to be done in the field of shared applications. Currently there are no widely adopted tools or practices for developing such applications, and it can therefore be quite resource-demanding. Close examinations of available development frameworks might reveal suitable starting points for building software of this kind. As previously mentioned, the possibility of developing a solid foundation for multi-user environments in the form of new operating systems or desktop environments could also be investigated.

We believe that shared systems are an often neglected area of computing with much promise and untapped potential. Further research towards this kind of technology could potentially change how we interact both with computers and with each other. As society grows more connected with people sharing more aspects of their lives, the development of technology should reflect this evolution. Escaping the personal computer paradigm and heading towards more shared digital environments could make computer assisted work more effective and efficient, and bring people and technology together in innovative and exciting ways.
References


