
D3.3 – Phase II Component Interfaces

Deliverable Type *	: PU
Nature of Deliverable **	: S
Version***	: Released
Created	: 21 September 2009
Contributing Workpackages	: WP3
Editor	: Pieter Simoens
Contributors/Author(s)	: IBBT, Prologue, NTUK, IMEC, IT, DT-Labs
File Name	: [D3.3 - Phase II Component Interface.doc]

- ***Deliverable type:**
 - *PU = Public,*
 - *RE = Restricted to a group of the specified Consortium,*
 - *PP = Restricted to other program participants (including Commission Services),*
 - *CO= Confidential, only for members of the MobiThin Consortium (including the Commission Services)*
- **** Nature of Deliverable:**
 - *P= Prototype,*
 - *R= Report,*
 - *S= Specification,*
 - *T= Tool,*
 - *O = Other.*
- *****Version:**
 - *Preliminary,*
 - *Draft 1, Draft 2,...,*
 - *Released*

Abstract:

This deliverable identifies and describes the interfaces between the system components of the MobiThin architecture that will be developed within WP3. Besides the exact interface definition, this deliverable provides an overview of the deployment of the WP3 components over the MobiThin hosting infrastructure and a detailed functionality description of all components.

The current deliverable takes into account the remarks of D2.4 – System Review and contains the updated WP3 architecture for phase II. It replaces completely D3.1 – Phase I Components and comes in parallel to D4.4, describing the WP4 system components and interfaces.

“The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 216946”

The **MOBITHIN Project Consortium** groups the following Organizations:

Interdisciplinary Institute for BroadBand Technology vzw	IBBT vzw	BE
Deutsche Telekom Laboratories	DT-Labs	DE
Prologue Software	Prologue	FR
Interuniversitair Micro-Electronica Centrum vzw	IMEC vzw	BE
NEC Technologies (UK) Ltd	NTUK	UK
Groupe des Ecoles des Télécommunications	GET	FR
JCP-Consult SAS	JCP	FR

EDITORIAL Change Management

Version	Date	Editor/Author	Comments
Draft	July 9, 2009	IBBT P. Simoens	Creation of document.
V1	September 14, 2009	IBBT P. Simoens	Merging of input Preparing document for review
V2	September 21, 2009	IBBT P. Simoens	Updated with comments from reviewers
R	September 29th	IBBT P. Simoens	Final editing

Contents

1.	<i>Executive Summary</i>	4
2.	<i>Introduction</i>	5
2.1	Project Context	5
2.2	Relation with deliverable 4.4	5
2.3	Deliverable scope	5
3.	<i>System Architecture</i>	6
3.1	Overall architecture	6
3.2	WP3 and WP4 components	7
3.3	Architectural Subsystems	8
3.3.1	Thin Client protocol architecture	8
3.3.2	Self Logistics Management	10
4.	<i>WP3 component functionality</i>	13
4.1	WP3 Components on Thin Client Server	13
4.1.1	WP3 components at User Session level	14
4.1.2	WP3 Components at Channel Level	16
4.2	WP3 components on MobiThin Client	19
4.2.1	General Components	21
4.2.2	Channel Components	22
5.	<i>Sequence Diagrams</i>	24
5.1	Image Transmission	24
5.2	Protocol Adaptivity	24
5.3	Starting up ContentManager components	24
5.4	A new application is started by the user	25
5.5	Forwarding of user events to the application	25
6.	<i>Interface definition</i>	31
6.1	Generic SLM interfaces	31
6.1.1	High-level view of Self Logistics Management	31
6.2	Thin Client Server interfaces	36
6.2.1	Components at user session level	36
6.2.2	Components at channel level	40
6.3	MobiThin Client interfaces	43
6.3.1	General Components	43
6.4	Components per channel	44
7.	<i>References</i>	47
8.	<i>Appendix A: Helper classes for SLM_MON and SLM_SLM interfaces</i>	48
8.1	Helper classes	48
8.1.1	Parameter	48
8.1.2	Value	49
8.1.3	Report, MonitoringReport, ActionReport	50
8.1.4	ReportMode	50
9.	<i>Appendix B: Definition of keycodes and buttoncodes</i>	51

1. EXECUTIVE SUMMARY

This deliverable defines the interfaces between all the MobiThin system architecture components that will be developed by WP3 during the MobiThin project. Deliberately, it has been written as an incremental update to deliverable 3.1, in which the phase I components are detailed. All remarks formulated by the System Review (WP2 – D2.4) are included, as well as the new WP3 components that were already introduced in D3.2. Consequently, this deliverable must be regarded as the single reference point for the WP3 architecture, providing a complete overview of the definition of all interfaces between WP3. The deliverable should be read in parallel with D4.4, defining the interfaces of the other components of the MobiThin architecture.

The new issues to be dealt with in phase II refer to:

- cross-layer communication between application layer and wireless layer
- implementation of the Self Logistics Management framework
- QoS mechanisms at the wireless layer
- a generic framework to capture application output and transcode it to an appropriate format

The first part of this deliverable provides a detailed insight on the parts of the MobiThin system that are developed in WP3. In D3.2, describing the components developed during phase I, a reworked version of the architecture presented in D3.1 was already presented. Several new components were added, related to capturing and encoding the application content, as well as to delivering user events to the application. The current deliverable studies in more detail the interfaces and architecture of these components. The chapters that are new or have been updated, are marked with an asterisk (*). This should allow the reader to clearly see the increment done for phase II.

The involved components are the ApplicationListener, capturing the application output, and the ContentConvertor, translating the application output to the most appropriate encoding for transmission. The delivery of user events is handled by the EventConvertor. The most prominent component is the SceneStateManager, coordinating the data flow in up- and downstream direction between the ApplicationListener and the data channels (TCSCCX components).

The second part of this deliverable contains an in-depth functionality description of every WP3 component. For every component, its task is detailed, as well as the other components it interfaces to. In addition to the two sequence diagrams already included in D3.1, three additional sequence diagrams were included. These illustrate how the new components behave during start-up of the User Session, what happens when a new application is started by the user and how user events are forwarded to the application.

The last part of the deliverable contains the actual interface definitions. Lastly, a new appendix was added, containing the protocol encoding for the user events.

2. INTRODUCTION

2.1 PROJECT CONTEXT

The objective of WP3 in MobiThin is the design of the adaptive cross-layer thin client protocol that makes use of supporting network services and comprises a wireless medium transmission protocol and an image transmission protocol. In this work package, the main building blocks of the cross-layer thin client protocol are investigated, designed and realized.

MobiThin is split in two phases. Phase I ran for 18 months, while phase II is shorter, taking only 12 months. Deliverable 3.1 defined the interfaces of the MobiThin components that were developed during phase I of the MobiThin project. The current deliverable D3.3 provides an update of D3.1, defining several new components and their interfaces that will be developed during phase II of the components. Furthermore, it takes into account the remarks formulated by the System Review, performed in WP2 and reported in D2.4. The present deliverable contains the final MobiThin WP3 architecture and replaces D3.1.

2.2 RELATION WITH DELIVERABLE 4.4

Together, the components developed in WP3 and WP4 constitute the complete MobiThin architecture. Therefore, this deliverable should be regarded as a whole with D4.4, describing the phase II components developed in WP4. In order to underline the complementarity, both deliverables were structured in the same way. For clarity reasons, some identical sessions appear in both deliverables.

2.3 DELIVERABLE SCOPE

The deliverable is intended to define the interfaces between the components that will be designed in the scope of WP3 during phase II of the project. This includes the deployment of system components over the MobiThin hosting infrastructure and a detailed functionality description with identification of the interfaces between the components. It was the goal to set this deliverable as the single reference point for the WP3 architecture. Therefore, it was decided to conceive D3.3 as an updated version of D3.1, also taking into account the remarks of the System Review of D2.4.

The remainder of this document is structured as follows. Section 3 extends the description of the system architecture of D2.2 and includes several new components. Furthermore, this section outlines the scope of WP3 and WP4 and gives an overview of the MobiThin subsystems that are in scope of WP3: the self logistics management and the adaptive thin client protocol. Section 4 contains a brief description of the functionality of all components, together with a high level overview of the communication between different MobiThin components. This section is split in two parts, one elaborating on the MobiThin Server components (section 4.1) and one on the MobiThin client components (section 4.2). Sequence diagrams, illustrating the use of WP3 components, are provided in section 5. In section 6, a detailed description of the interfaces of all components is given.

Compared to D3.1 – Phase I component interfaces, the following sections are reworked or new:

- 3.1 – overall system architecture
- 3.2 – WP3 and WP4 components
- 3.3 – architectural subsystems
- 4.1 – WP3 components on the thin client server
- 5.3 – 5.4 – 5.5 – new sequence diagrams
- 6.2.1.3 ApplicationListener@US
- 6.2.1.4 EventConvertor@US
- 6.2.1.5 ContentConvertor@US
- Appendix B – Definition of keycodes and buttoncodes

In the deliverable, these sections are indicated with an asterisk.

3. SYSTEM ARCHITECTURE

The MobiThin system architecture was described in D2.2 – System Architecture. In this architecture, the building blocks of the MobiThin Client and the MobiThin Server were introduced. This architecture was reviewed and complemented within previous deliverables.

The first part of this chapter shows a brief overview of the entire MobiThin architecture. The second part recalls the distribution of the components between WP3 and WP4 workpackages.

3.1 OVERALL ARCHITECTURE(*)

The architecture elaborated in previous deliverables was based on the concept of the MobiThin Client (MTC) and the MobiThin Server (MTS). MTC includes all MobiThin software to be installed on the terminal device. MTS includes all MobiThin software to be installed on the servers of the MobiThin System hosting infrastructure. For details on this architecture please refer to previous MobiThin project deliverables (D2.2, D3.2).

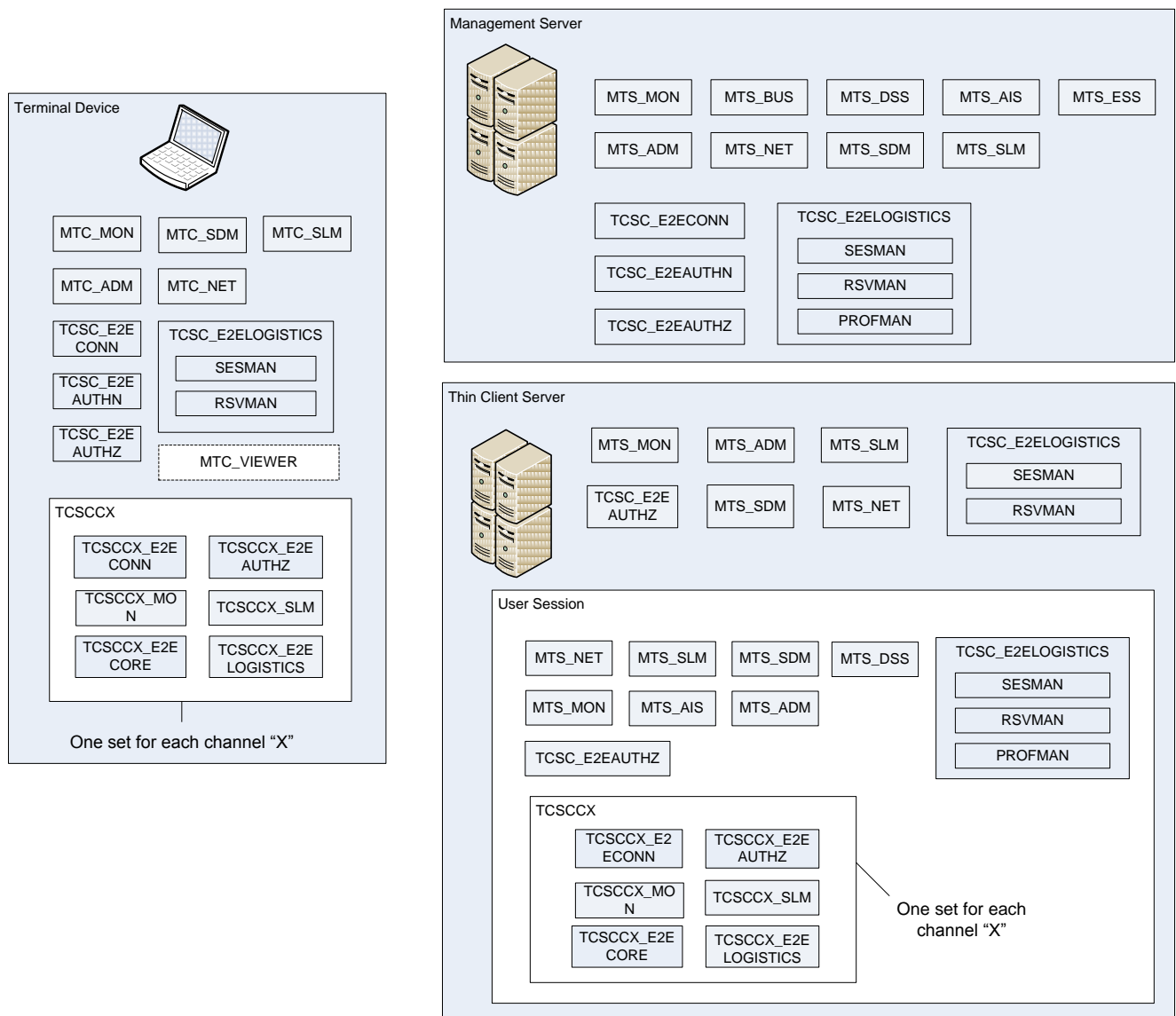


Figure 1 - Exemplary deployment of MobiThin suite

As can be seen in the figure, many components with the same name appear on different levels. It was proposed to extend their naming by a suffix to indicate on which level the component is situated. Components at Management Server will be indicated by the suffix @MS, at User Session level by @US, etc. An overview of the suffixes is given in Table 1. These suffixes will only be added to avoid ambiguity. For example, some component names at the MobiThin Client start with

“MTC_”. Since it is already clear from the component name on where these components are situated, no suffix will be added.

Suffix	Component is at level of
@MS	Management Server
@TCS	Thin Client Server
@US	User Session
@TCSCCX	Channel “X”

Table 1: Suffixes for component naming

Also, for commodity reasons, monitoring components will be abbreviated to “MON” components, and Virtual Machine will be abbreviated to “VM”.

3.2 WP3 AND WP4 COMPONENTS (*)

WP3 is responsible for the development of the thin client transmission protocol, while the focus of WP4 is on system management. For each component appearing in Figure 1, the table below indicates in which WP it will be designed and developed.

Table 2: WP3 and WP4 components

	Component	Developed in		Component	Developed in
Management Server	MTS_MON@MS	WP4	User Session	MTS_NET@US	WP3
	MTS_BUS@MS	WP4		MTS_SLM@US	WP3
	MTS_DSS@MS	WP4		MTS_SDM@US	WP4
	MTS_AIS@MS	WP4		MTS_MON@US	WP4
	MTS_ESS@MS	WP4		MTS_AIS@US	WP4
	MTS_ADM@MS	WP4		MTS_ADM@US	WP4
	MTS_NET@MS	WP4		MTS_DSS@US	WP4
	MTS_SDM@MS	WP4		TCSC_E2EAUTHZ@US	WP4
	MTS_SLM@MS	WP4		TCSC_E2ELOGISTICS@US	WP4
	TCSC_E2ECONN@MS	WP4		TCSCCX@US (all components)	WP3
	TCSC_E2EAUTHN@MS	WP4			
	TCSC_E2EAUTHZ@MS	WP4			
	TCSC_E2ELOGISTICS@MS	WP4			
Thin Client Server	MTS_MON@TCS	WP4	MobiThin Client	MTC_MON	WP4
	MTS_ADM@TCS	WP4		MTC_SDM	WP4
	MTS_SDM@TCS	WP4		MTC_SLM	WP3
	MTS_SLM@TCS	WP4		MTC_ADM	WP4
	MTS_NET@TCS	WP4		MTC_NET	WP3
	TCSC_E2ELOGISTICS@TCS	WP4		TCSC_E2ECONN@MTC	WP4
	TCSC_E2EAUTHZ@TCS	WP4		TCSC_E2EAUTHZ@MTC	WP4
		TCSC_E2EAUTHN@MTC		WP4	
		TCSC_E2ELOGISTICS@MTC		WP4	
		MTC_VIEWER		WP3	
		TCSCCX@MTC (all components)	WP3		

3.3 ARCHITECTURAL SUBSYSTEMS (*)

In the MobiThin architecture, the following major subsystems can be distinguished:

- Monitoring, comprising all X_MON components
- Authentication and authorization, comprising all X_AUTHN and X_AUTHZ components
- Administration, comprising all X_ADM components
- Thin client protocol, comprising all TCSCCX components.
- Self logistics management, comprising all X_SLM components

In accordance with Table 2, only the subsystems for self logistics management and for the thin client protocol are within the scope of WP3. These subsystems are described in the following sections of this chapter. The other subsystems are described in D4.4.

3.3.1 Thin Client protocol architecture

Figure 2 presents the overall architecture of the User Session level (US) on the MobiThin Thin Client Server (TCS). Compared to Figure 1, the TCSCC component (Core of the Thin Client Server Core) has been renamed to Content Manager (CM). This new name should better represent the intended functionality. The CM provides persistent scene state management between the application input listeners and the output encoders. It is necessary in order to be able to reconfigure the output encoding mechanisms in response to requests received from the Self Logistics Manager. The CM takes all necessary steps to relay the application content to the client, and the user input to the applications.

The Application Listener offers an interface to the application to generate its audiovisual content, while the encoded output is generated by the TCSCCX components. If necessary, an intermediate conversion is performed by the appropriate Content Converter. Generic interfaces have been defined to communicate with the Self Logistics Management and the Session Manager.

Three components are involved in the content transmission, namely the Content Manager (CM), the Self Logistics Manager (SLM) and the Session Manager (SESMAN). The other components, not directly related to content transmission, are added to the figure with a dashed line. The following sections provide brief descriptions of both the Content Manager and the Self Logistics Manager. The Session Manager is described in WP4 deliverables.

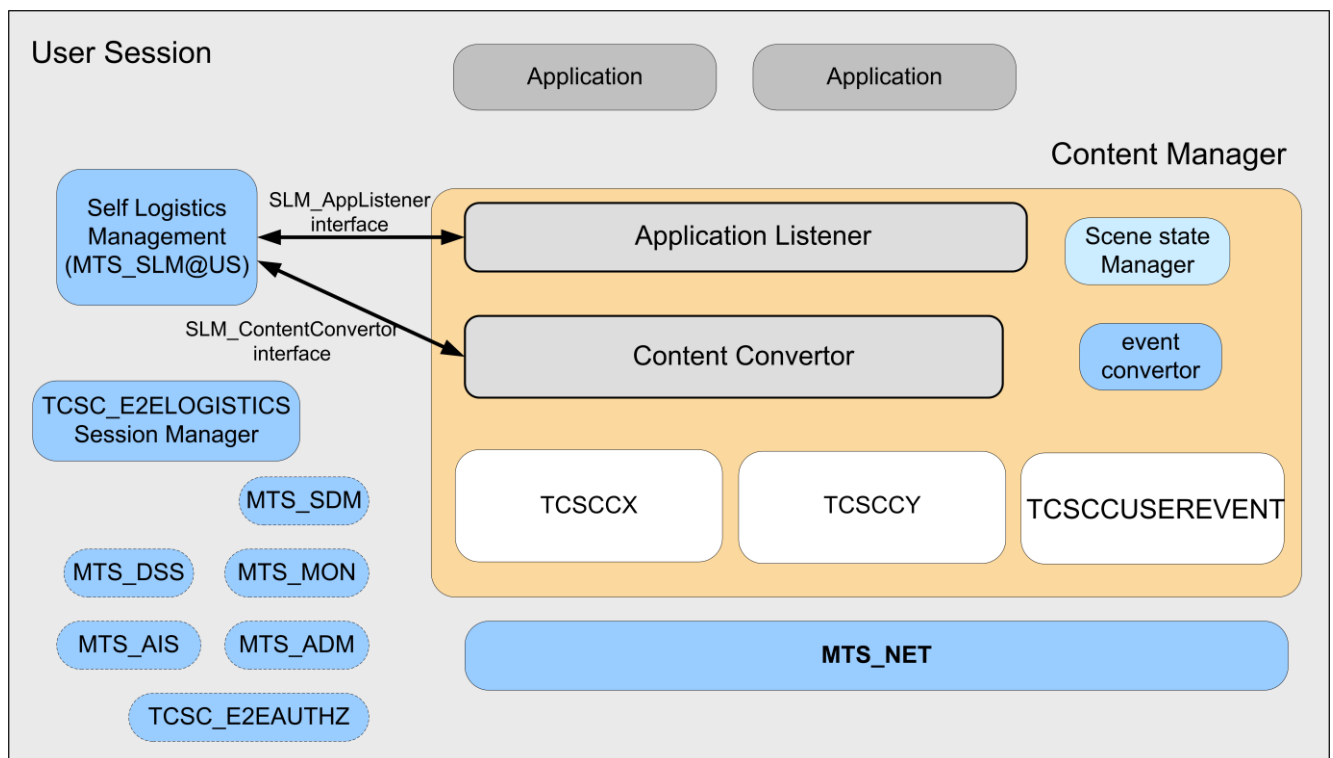


Figure 2 – Reworked architecture of the User Session architectural level, together with their newly defined interfaces. The dashed components are not involved with image transmission.

3.3.1.1 APPLICATIONS

Various different application types are intended for use by the MobiThin Content Manager system and they may differentiate in terms of the nature of the encoding of their graphical user interface output as follows:

X.11

X.11 Legacy applications are a primary target for direct integration to the architecture of the MobiThin Content Manager and the various listener and conversion tools will be provided from the outset.

VRML

VRML (Virtual Reality Modelling Language) production applications are intended to be supported by the architecture of the MobiThin Content Manager since this is the source language used by BiFS, one of the intended encoding formats of the back end and as such would require little overhead in terms of conversion.

SVG

SVG (Scaled Vectorial Graphics) production applications are intended to be supported by the architecture of the MobiThin Content Manager since this is the source language of LASer one of the intended encoding formats of the back end and as such would require little overhead in terms of conversion.

STREAMING AUDIO and VIDEO

Both audio and video applications streams are to be intercepted directly for handling by the MobiThin Content Manager at a high level in order to ensure preservation of the highest degree of abstraction and semantic information.

3.3.1.2 CONTENT MANAGER

This section describes the architecture of the MobiThin Content Manager within the MobiThin Thin Client Server. The Content Manager, working in conjunction with information provided by the Session Manager, is responsible for the coordination, persistence and correspondence of graphical and aural application output for the MobiThin Client Device. The Content Manager handles two types of communication channel, between the server and the client, the downstream channel towards the client display (TSCCX), and the upstream channel from the client endpoint to the Content Manager (TSCCUSEREVENT). The former, downstream channel, is comprised solely of graphical user interface description information whilst the latter, upstream TSCCUSEREVENT channel, comprises both user keyboard and mouse events. Coordination of these two channel components is performed by the Scene State Manager (SSM).

3.3.1.3 SCENE STATE MANAGER

The scene state manager component is the heart of the Content Manager and is responsible for the coordination of information in both upstream and downstream channels. The initial scene description will provide information that will allow a standard MobiThin thin client device to be configured to forward all user key board and mouse events back up to the MobiThin Content Manager via the upstream channel.

3.3.1.4 DOWNSTREAM

The downstream manager component comprises two sections. An input section captures application output through specialised listener components and performs conversion to the appropriate scene description format to be managed by the scene state manager. The output section receives scene description information from the scene state manager to be encoded using the required output format and packetised prior to transport over the network to the client endpoint device.

3.3.1.4.1 Listeners

Specialised listening device processes will be developed for the capture of the various application input types described above.

X.11

The X.11 listening processes capture application graphical requests over the socket to the X.11 Display Server. X.11 packet requests captured this way will be converted to elements in an internal scene description format and placed on the scene management tree for the calling display.

SVG

The SVG listening process simply pipes incoming information “as is” to the internal Scene State Manager.

VRML

As for the preceding listener, the VRML listener will transfer, unchanged, input directly to the internal Scene State Manager.

AUDIO/VIDEO

The audio and video codec listening devices will intercept requests to the codec by the application process and transfer high level information to the Scene State Manager allowing best optimization of these heavy weight content providers. The primary objective here is to avoid the decoding and re-encoding of the audio or video bit stream by redirecting the original bit stream content or source description to the Scene State Manager.

3.3.1.4.2 Converters

Behind each application input type listening device we find the corresponding converter responsible for the conversion of the input form to that of the internal scene tree. The scene tree format is a hybrid description language offering an aggregate of entity types for each of the differing entity types to be encountered in the various application input streams.

3.3.1.4.3 Encoders

When a particular scene is to be refreshed on the client terminal the Scene Tree Manager will make use of the output encoder selected during configuration of the client session. Encoders are intended to be provided for both BiFS and LAsER encodings, certain scene tree entities however will not be possible unless the output encoding permits.

3.3.1.4.4 Packetizers

After encoding the packetizers are responsible for bit stream compression prior to output of the scene tree description over the “wire” to the client terminal.

3.3.1.5 UPSTREAM

The upstream channel management component is responsible for the reception of user keyboard and mouse events from the client endpoint device and for their delivery for processing to the appropriate application program. This is performed with assistance from the scene state manager component.

3.3.1.5.1 Event Handler

Client side mouse and keyboard events will be received by the upstream event handler after decoding to be made available to the application in the manner appropriate to each of the application operational types.

3.3.1.5.2 Event DECODER

Events are transmitted over the wire in a MobiThin specific format and are decoded on input from the device prior to input to the event handler.

3.3.1.5.3 Event CONVERTOR

Events to be made available to the application program require conversion to the appropriate format. For the moment only the X.11 legacy application will be capable of handling upstream events.

3.3.2 Self Logistics Management

The Self Logistics Management (SLM) components are vital components of the MobiThin system architecture. They are the steering components, making intelligent decisions on the configuration of thin client protocol parameters to optimize user experience, and to ensure proper functioning of the MobiThin system. The SLM components are present in all architectural levels, from inside a channel up to the management server level. The hierarchical SLM architecture is presented in Figure 3.

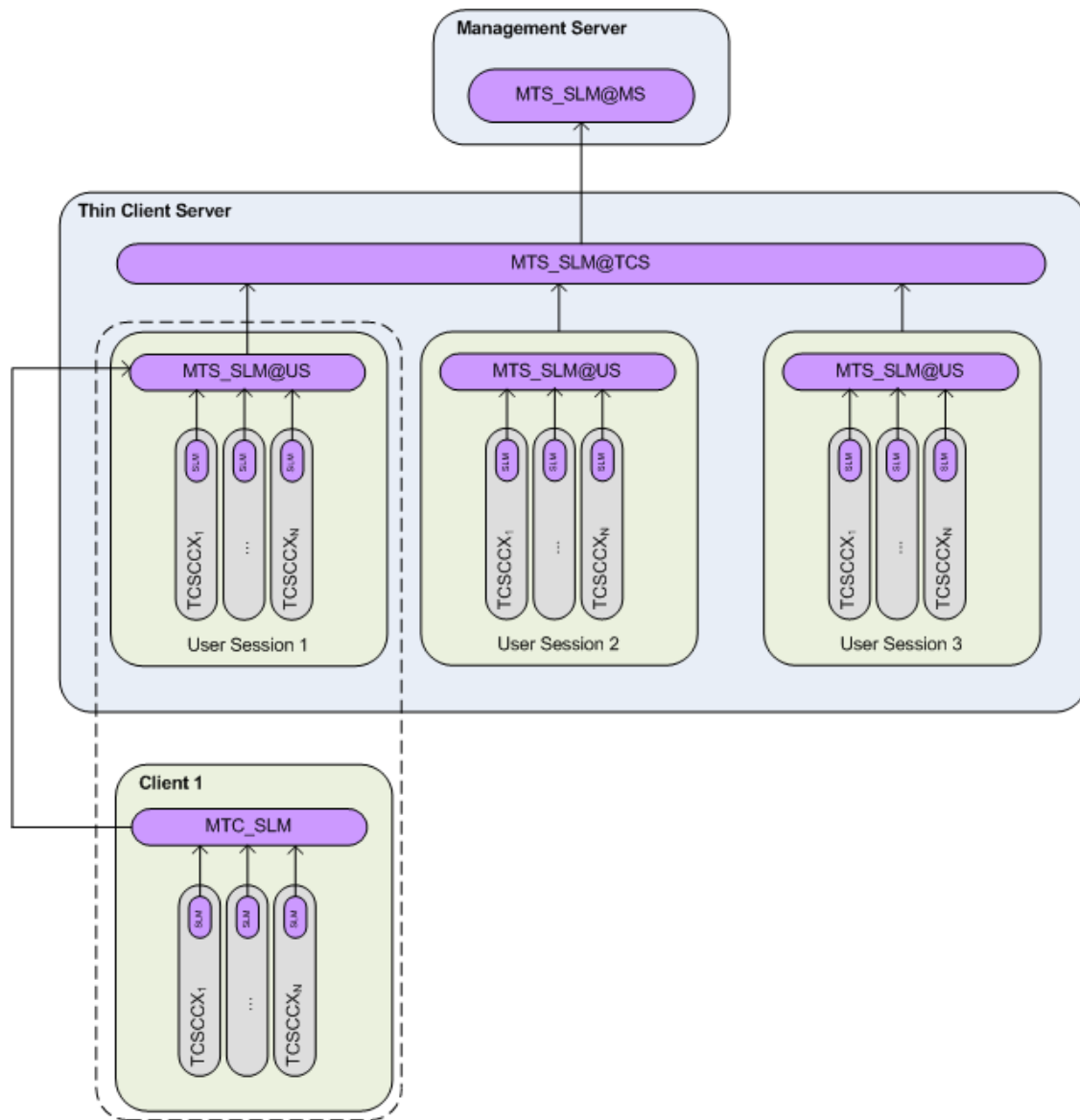


Figure 3 - Self Logistics Management Architecture

An SLM component ensures the proper working of the level it is situated at. The SLM components anticipate on the varying state of channels or the environment (client terminal state, server state, network state), or react on sudden changes in the environment (new application, bandwidth drop on wireless link). When an SLM component is triggered to solve a problem (e.g. through a monitoring component), it tries first to solve the problem on its own level. When it decides that it cannot solve the problem at this level, it will trigger the SLM component at a higher level. At the Management Server, the SLM component should be able to solve the problem. If not, the MobiThin system cannot optimize anything for the user perceiving bad QoS. It is clear that this situation will be the worst case for MobiThin.

When an SLM-component has found a solution that needs to set parameters at a lower level, then an SLM report is sent from the SLM-component to all SLM-components on the lower levels with the required changes.

The arrows between the different SLM components in Figure 3 indicate the information flow between SLM components at different levels.

In phase I of the MobiThin project, the SLM components will only be triggered when an alert was sent from one of the monitoring components. In phase II, the SLM components will make pro-active decisions (e.g. based on a user’s profile).

To clarify to the reader the functionality of the SLM components at different levels, some examples of decisions are given in the next sections. Please note that the examples given, are only a subset of possible decisions made by the SLM components, since the SLM components need further investigation throughout the project to identify new problems with new solutions and decisions.

3.3.2.1 NETWORK TRANSPORT BANDWIDTH

Detected changes in network bandwidth would allow selection of a more efficient encoding protocol to improve user experience. This would then lead to a scene tree update of the current scene state to be transmitted to the client terminal device using the newly established encoding and packetising techniques.

3.3.2.2 CLIENT TERMINAL CHANGE

An explicit change in client terminal software or hardware would require the current scene tree to be refreshed on the new client terminal device.

3.3.2.3 APPLICATION OUTPUT REQUIREMENTS

Changes in application content requirements would require negotiation for the activation of an output encoding more suitable to the subsequent application output. Scene tree refresh will be performed using the newly established encoding and packetising techniques.

4. WP3 COMPONENT FUNCTIONALITY

From Table 2

Table 2, it can be derived that the WP3 components are located at the following architectural levels:

- User Session (US)
- Channel at User Session side (TCSCCX@US)
- MobiThin Client (MTC)
- Channel at MobiThin Client side (TCSCCX@MTC)

This chapter gives a detailed overview of all components in scope of WP3. A functionality description of the other components of Figure 1 can be found in D4.4.

4.1 WP3 COMPONENTS ON THIN CLIENT SERVER (*)

Figure 4 gives an overview of the MobiThin components deployed on a Thin Client Server. The WP3 components are situated at the User Session and the Channel level. Their functionality is described in the next two sections. The functionality of the other components can be found in D4.4. Channel X refers to a channel established between User Session and MobiThin Client, e.g. TCSCCVideo, TCSCCAudio, TCSCCMouse, etc. The channel concept was introduced and discussed in D2.2.

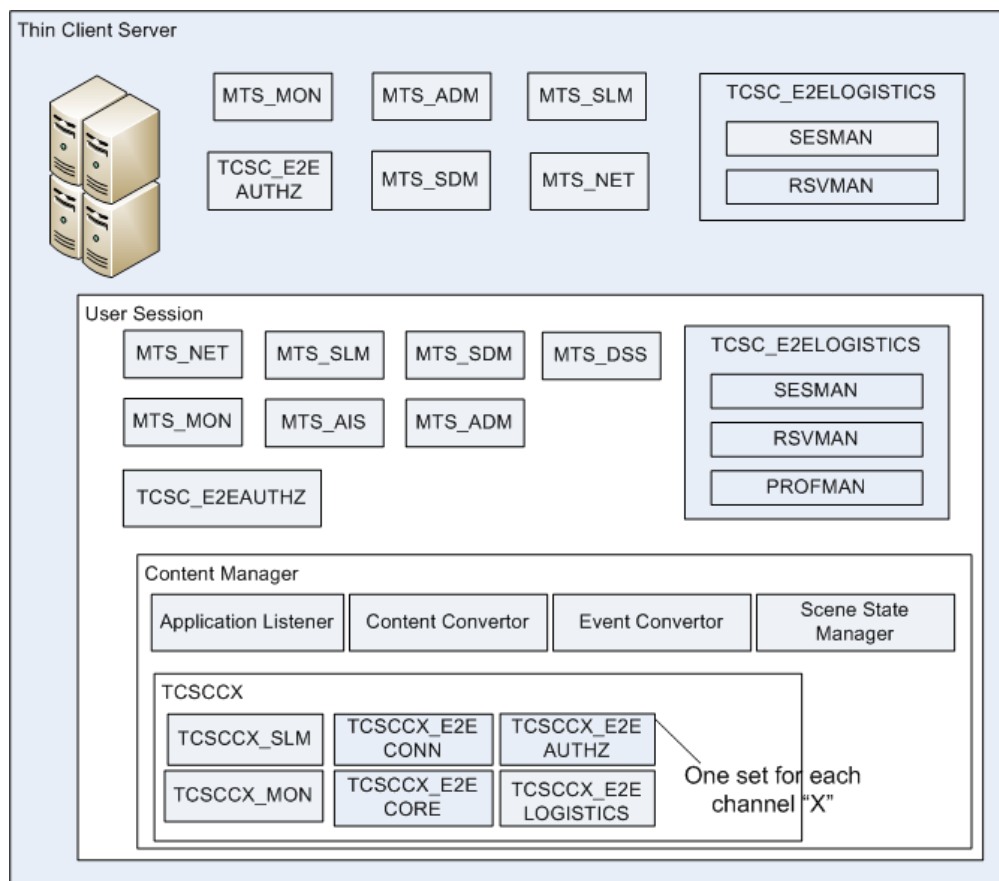


Figure 4 - MobiThin components deployed on the Thin Client Server

4.1.1 WP3 components at User Session level

The User Session is the environment running on the MobiThin infrastructure in which users execute their applications. At this level, user input is processed and audiovisual application output is forwarded to the client.

High level functionalities to be covered at User Session level include:

- Provide access to the user’s private data at the Data Storage Service (DSS) and to the Application Image Service (AIS)
- Monitor the user session state
- Management of the thin client protocol channels
- User session management
- Resource Reservation per channel
- Profiling management

At User Session level, the following components are in scope of WP3:

- MTS_NET@US
- MTS_SLM@US
- Content Manager

MTS_NET @ US: Network	
Functionality	<ul style="list-style-type: none"> ▪ This component hides OS specific network stack implementation from the other components. It offers a generic interface to establish network connections, to change layer parameters, etc. ▪ Offers an interface to set-up and configure network connections. ▪ Offers an interface to the network stack of each TCSCCX channel. ▪ Interfaces with the network for QoS negotiation and configuration (phase II)
Is used by	<ul style="list-style-type: none"> ▪ All components on channel level: TCSCCX_E2ECONN@US TCSCCX_E2ECORE@US TCSCCX_E2ELOGISTICS@US TCSCCX_E2EAUTHZ@US TCSCCX_SLM@US TCSCCX_MON@US ▪ All components on user session level: MTS_MON@US MTS_SLM@US MTS_SDM@US MTS_ADM@US MTS_DSS@US ContentManager
Uses	NONE

MTS_SLM @ US: Self Logistics Manager	
Functionality	<ul style="list-style-type: none"> ▪ Ensures the User Session self-management: resilience, repairing malfunctioning components... ▪ Handles monitoring alerts at the user session level. ▪ Handles alerts received from lower level SLM components: MTC_SLM, TCSCCX_SLM. ▪ Cross-layer controller of the channels and their network stacks based on policy and monitoring information. ▪ Configures parameters at User Session level. ▪ Ensures appropriate application content encoding by configuring the ApplicationListener and ContentConvertor components. ▪ Instruct monitoring components to provide additional information, e.g. to increase the monitor frequency.

	<ul style="list-style-type: none"> ▪ Solves alerts than cannot be handled by MTC_SLM and instructs MTC_SLM. ▪ Notifies higher level components (MTS_SLM@TCS) when the available algorithms/mechanisms are insufficient to resolve monitoring alerts. ▪ Executes instructions received from higher level SLM components (MTS_SLM@TCS)
Is used by	<ul style="list-style-type: none"> ▪ TCSCCX_SLM@US (support for monitoring alerts that cannot be solved at channel level) ▪ MTS_SLM@TCS (send SLM decision report, send parameters to change, etc.) ▪ MTC_SLM (when client SLM cannot solve a problem) ▪ MTS_MON@US (report monitoring alerts)
Uses	<ul style="list-style-type: none"> ▪ MTS_SLM@TCS (when the available algorithms/mechanisms at User Session level are insufficient to resolve monitoring alerts, to guarantee QoS, etc.) ▪ MTC_SLM (server-side instructions for MTC_SLM) ▪ TCSCCX_SLM@US (instructions, parameters to change, etc.) ▪ TCSC_E2ELOGISTICS_SESMAN@US (channel information) ▪ MTS_SDM@US (retrieving and storing of data for self management, such as policy and monitoring information) ▪ MTS_MON@US (monitoring instructions, e.g. which parameters to monitor and at what frequency, request for possibly filtered and summarized monitoring information) ▪ MTS_NET@US (setting parameters of the network stack layers) ▪ MTS_ADM@US (to configure parameters at User Session level) ▪ ApplicationListener (instruct to listen for specific format of application content) ▪ ContentConvertor (configure the conversion between input format from ApplicationListener and output format of the TCSCCX component)

The ApplicationListeners constitute the interface offered to the application. The following list gives an overview of the ApplicationListeners that possibly will be implemented. This list is only meant to give the reader better insight in the component's functionality. In no means, it is an indication of which ContentConvertors will actually be implemented.

- ApplicationListener offering the X.11 interface [1]
- ApplicationListener offering the ALSAlib interface [5]
- ApplicationListener offering the LAsER interface [4]

ApplicationListener (*)	
Functionality	<ul style="list-style-type: none"> ▪ Captures a specific type of graphical output of the applications' user interface ▪ Multiple ApplicationListener components can run in parallel, each capturing a different type of graphical output from the application (X11, VRML, SVG, streaming audio/video...)
Is used by	<ul style="list-style-type: none"> ▪ MTS_SLM@US (start and stop the component)
Uses	<ul style="list-style-type: none"> ▪ ContentConvertor@US (forward the graphical output of the application)

ContentConvertor (*)	
Functionality	<ul style="list-style-type: none"> ▪ Performs the required parsing and conversion to a format appropriate as input for one of the TSCCX components, e.g. parsing X11 content (delivered by the ApplicationListener) and calling the appropriate functions call for BiFS encoding
Is used by	<ul style="list-style-type: none"> ▪ ApplicationListener to push the capture graphical application content to ▪ MTS_SLM@US (start and stop the component, retrieve context information)
Uses	<ul style="list-style-type: none"> ▪ TCSCCX component to encode the output

SceneStateManager (*)	
Functionality	<ul style="list-style-type: none"> ▪ Responsible for the coordination of information in both upstream and downstream channels

	(TCSCCX). <ul style="list-style-type: none"> Contains the current Scene State at the appropriate level of detail. Provides information about the mapping from user events to the appropriate application
Is used by	<ul style="list-style-type: none"> EventConvertor to get information to which application the incoming events should be reported to MTS_SLM@US (exchange scene information)
Uses	<ul style="list-style-type: none"> MTS_SLM@US (exchange scene information)

EventConvertor (*)	
Functionality	<ul style="list-style-type: none"> Interprets the user events incoming via the TCSCCUSEREVENT channel. Transcodes to the appropriate format to report the user event to the application.
Is used by	<ul style="list-style-type: none"> TCSCCUSEREVENT_E2ECORE@US to forward the incoming user events
Uses	<ul style="list-style-type: none"> SceneStateManager to retrieve information to which application the events must be delivered

4.1.2 WP3 Components at Channel Level

Inside each User Session, several channels can be active. The components described in this section are instantiated for each channel inside the user's session. High level functionalities at the channel level include:

- Channel monitoring
- Connection portal between channel on the client device and channel in the user session (set-up, authentication, authorization)
- Self Logistics Management (channel specific optimization)
- End-To-End Core communication (encoding/decoding channel specific messages)

The WP3 components at channel level are:

- TCSCCX_MON@US
- TCSCCX_SLM@US
- TCSCCX_E2ECONN@US
- TCSCCX_E2EAUTHZ@US
- TCSCCX_E2ECORE@US
- TCSCCX_E2ELOGISTICS@US

On purpose, the channel specifications have been kept as minimal as possible. More details are only given for the User Event channel. This should enable external developers to create their own channel to be integrated in the MobiThin framework. The internal channel operation is left to the channel designer. Besides the cooperation between the channel monitoring and self logistics, only the interfaces to the User Session level have been detailed.

TCSCCX_MON@US: Monitoring	
Functionality	<ul style="list-style-type: none"> Monitors channel state Filters and summarizes channel monitoring information. Notifies TCSCCX_SLM@US of monitoring alerts. Reports to higher layer components (MTS_MON@US).
Is used by	<ul style="list-style-type: none"> MTS_MON@US (request up-to-date information) TCSCCX_SLM@US (request up-to-date information) TCSCCX_E2ECORE@US (report monitoring information)
Uses	<ul style="list-style-type: none"> MTS_MON@US (provide monitoring information) TCSCCX_SLM@US to report monitoring alerts directly (traps, alerts) MTS_NET@US (send/receive bytes from the network)

	<ul style="list-style-type: none"> TCSCCX_E2ECORE@US (monitoring)
--	--

TCSCCX_SLM@US: Self Logistics Manager	
Functionality	<ul style="list-style-type: none"> Maintains the channel protocol state Ensures the proper working of the channel (e.g. handling channel protocol retransmissions) Handles monitoring alerts at the TCSCCX level. Instruct TCSCCX_MON@US components to provide additional information, to increase the monitor frequency, etc. Notifies higher level components (MTS_SLM@TCS) when the available algorithms/mechanisms are insufficient to resolve monitoring alerts.
Is used by	<ul style="list-style-type: none"> TCSCCX_MON@US (report problems) MTS_SLM@US (send SLM decision report, send parameters to change, etc.)
Uses	<ul style="list-style-type: none"> MTS_SLM@US (when channel specific mechanisms are insufficient to ensure QoS/QoE according to defined policies) TCSCCX_MON@US (request monitoring information) TCSCCX_E2ECORE@US (parameter configuration) MTS_NET@US (send/receive bytes from the network)

TCSCCX_E2ECORE@US: End-To-End Core	
Functionality	<ul style="list-style-type: none"> Coding and decoding of channel specific messages
Is used by	<ul style="list-style-type: none"> TCSCCX_SLM@US (parameter configuration) The user application (to generate audiovisual output) TCSCCX_MON@US (monitoring)
Uses	<ul style="list-style-type: none"> MTS_NET@US (send/receive bytes from the network) TCSCCX_MON@US (report monitoring information)

TCSCCX_E2ECONN@US: End-To-End Connection	
Functionality	<ul style="list-style-type: none"> Set up connection for channel X between client device and the User Session.
Is used by	<ul style="list-style-type: none"> TCSC_E2ELOGISTICS_SESMAN@US (instructs channel to start/stop etc.)
Uses	<ul style="list-style-type: none"> TCSCCX_E2EAUTHZ@US (check if the user is allowed to start this channel) MTS_NET@US (socket creation during channel set-up)

TCSCCX_E2EAUTHZ@US: End-To-End Authorization	
Functionality	<ul style="list-style-type: none"> Authorization (e.g. is the user allowed to use the print channel?)
Is used by	<ul style="list-style-type: none"> TCSCCX_E2ECONN@US (check if the user is allowed to start this channel)
Uses	<ul style="list-style-type: none"> MTS_NET@US (send/receive bytes from the network) TCSC_E2EAUTHZ@US (check whether the sessionTicket sent by the user is valid)

Note: This authorization component only applies to internal components. For external services, such as DSS, authorization will be obtained via the authorization methods provided by that service.

The E2ELOGISTICS components shall contain some negotiation component between client and server end points. They will discover for instance their respective versions, their settings and according to those settings they may need some additional “on-the-fly” resources e.g. when a channel requires more bandwidth, or a lower maximum round-trip delay.

TCSCCX_E2ELOGISTICS@US: End-To-End Logistics	
Functionality	<ul style="list-style-type: none"> ▪ Takes in charge the negotiation of needed resources at both end points of a channel. This is to ensure correct end-to-end session. ▪ Requests the reservation of needed resources at the channel level
Is used by	<ul style="list-style-type: none"> ▪ Any other component that needs resource reservation at the channel level
Uses	<ul style="list-style-type: none"> ▪ MTS_NET@US (send/receive bytes from the network)

SPECIAL CASE: TCSCCUSEREVENT (*)

A specific channel is the TCSCCUSEREVENT channel. It will mostly transport upstream traffic from client to server, containing the encoded user events to be delivered to the application.

The TCSCCUSEREVENT_E2ECORE@US component will decode the received user events, and forward them to the appropriate EventConvertor.

TCSCCUSEREVENT_E2ECORE@US	
Functionality	<ul style="list-style-type: none"> ▪ Decodes the user events received from the client ▪ Forwards the user events to the EventConvertor
Is used by	<ul style="list-style-type: none"> ▪ SceneStateManager
Uses	<ul style="list-style-type: none"> ▪ EventConvertor to forward the decoded user event data.

4.2 WP3 COMPONENTS ON MOBITHIN CLIENT

The MobiThin Client is the software to be installed on the client’s terminal device. MTC contains both the thin client viewer (MTC_VIEWER) and some management components (monitoring components, administration components, etc.). In Figure 5, the architecture of MTC that was presented in D2.2, is shown.

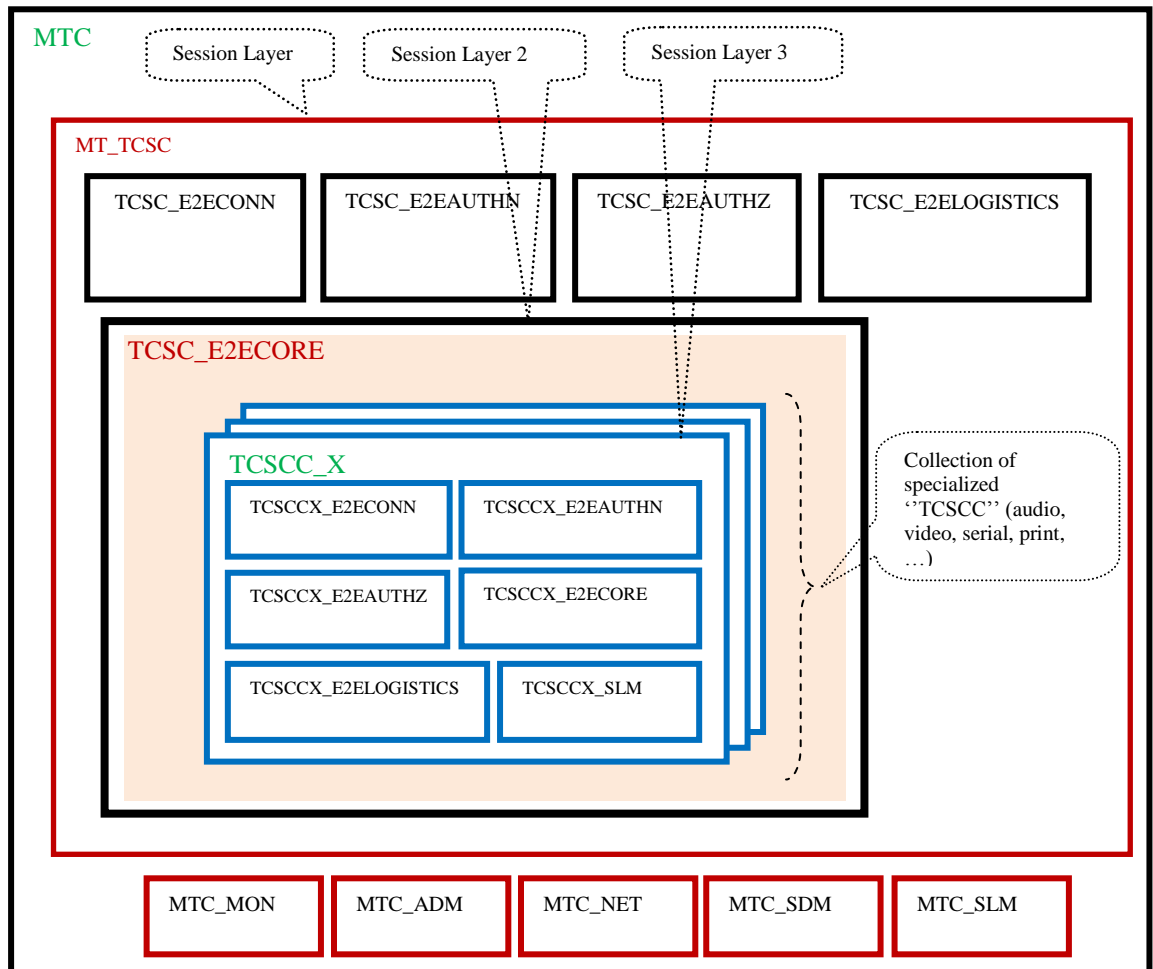


Figure 5 - MobiThin Client Architecture, as defined in D2.2

High level functionalities of the MobiThin client software include:

- Monitoring device state (CPU load, memory usage, battery status, wireless link statistics, etc.)
- Administrative tasks (setting parameters on the client device by the user or a remote administrator)
- Self Logistics Manager (e.g. energy-efficient transmission over the wireless link)
- Self Data Manager (e.g. containing user and/or device profile)
- Network communication
- Connection set-up with the MobiThin Management Server, authentication and authorization
- End-To-End Logistics (e.g. session management, resource reservation on the client device)
- Channel connection set-up, authentication and authorization with MobiThin Thin Client Server
- End-To-End Core Communication (encoding and decoding of channel specific messages)

Figure 6 presents the MobiThin software components to be installed at the client. Compared to Figure 5, a MobiThin Client Viewer was added. This component renders the graphics on the display and captures user events such as keystrokes and pointer movements. For the rendering of the display, the MobiThin Client Viewer will cooperate tightly with the underlying mobile device operating system (MTS_ENV, see D2.2 – System Architecture). The “Channel X” in Figure 5 refers to a channel established between the Terminal Device and the User Session, e.g. TCSCCVideo, TCSCCAudio, TCSCCMouse, etc. The channel concept was introduced and discussed D2.2.

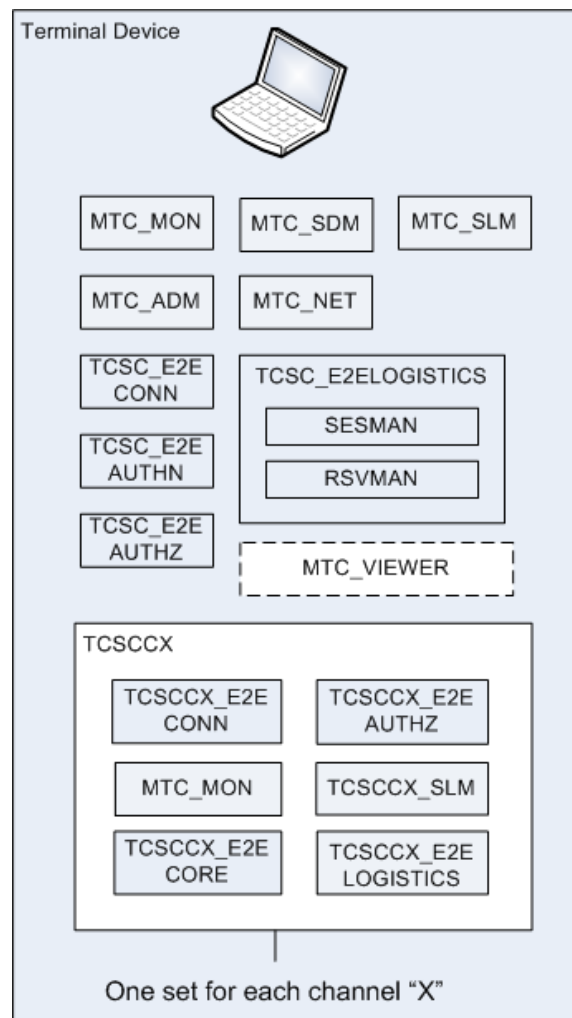


Figure 6 - Components to be installed at the Terminal Device. A MobiThin Client Viewer was added, responsible for presenting the graphical output and capturing the user events.

Two categories of MobiThin Client components can be distinguished. The first category comprises the general components for administration, monitoring and session management. The second category contains per-channel components (TCSCCX) to transport user events, peripheral data and audiovisual data in both directions between client and server. In the next sections, a functionality description is given for each WP3 component in these two categories. The functionality description of the other components can be found in D4.1.

The WP3 components at MobiThin Client are:

- General components
 - MTC_SLM
 - MTC_NET
- Components per channel
 - TCSCCX_E2ECONN@MTC
 - TCSCCX_E2EAUTHZ@MTC
 - TCSCCX_E2ECORE@MTC
 - TCSCCX_SLM@MTC
 - TCSCCX_MON@MTC
 - TCSCCX_E2ELOGISTICS@MTC

4.2.1 General Components

MTC_SLM: Self Logistics Manager	
Functionality	<ul style="list-style-type: none"> ▪ Ensures the MobiThin client self-management: resilience, repairing malfunctioning component. ▪ Handles monitoring alerts at the MTC level. ▪ Handles alerts received from TCSCCX_SLM@MTC components. ▪ Cooperates with MTS_SLM@US when necessary ▪ Cross-layer controller of the channels and their network stacks based on policy and monitoring information. ▪ Configures parameters at MobiThin Client level ▪ Instruct monitoring components to provide additional information, e.g. to increase the monitor frequency. ▪ Notifies higher level components (MTS_SLM@US) when the available algorithms/mechanisms are insufficient to resolve monitoring alerts. ▪ Tries to execute instructions received from higher level SLM components (MTS_SLM@US)
Is used by	<ul style="list-style-type: none"> ▪ MTC_MON (to report monitoring alerts) ▪ TCSCCX_SLM@MTC (when the channel specific algorithms are insufficient to resolve monitoring alerts or to ensure the QoS/QoE policy definitions, under the given circumstances of device/network resources, application characteristics...) ▪ MTS_SLM@US (for server-side instructions of the cross-layer optimization)
Uses	<ul style="list-style-type: none"> ▪ MTC_ADM (setting parameters at MobiThin Client level) ▪ MTC_SDM (retrieving/storing self data such as policy, monitoring information) ▪ MTC_MON (instruct the monitoring component, e.g. to increase the monitoring frequency, to monitor other parameters) ▪ MTS_SLM@US (when the locally available algorithms/mechanisms are insufficient to resolve monitoring alerts or to guarantee QoS) ▪ TCSCCX_SLM@MTC (to give instructions on the layer optimization) ▪ MTC_NET (setting parameters of the transport, network and wireless link layer)

MTC_NET: Network	
Functionality	<ul style="list-style-type: none"> ▪ This component hides the device OS specific network stack implementation from the other MobiThin Client components. It offers a generic interface to establish network connections and to change layer parameters. ▪ Offers an interface to set-up and configure network connections. ▪ Offers an interface to the network stack of each TCSCCX channel. ▪ Network control (QoS-related parameters).
Is used by	<ul style="list-style-type: none"> ▪ TCSC_E2ECONN@MTC (for setting up initial connection) ▪ MTC_ADM (system administration) ▪ MTC_SLM (changing layer parameters, e.g. of TCP connections, wireless channel) ▪ TCSCCX_E2ECORE@MTC (core components of the channels, writing to and reading from the network) ▪ TCSCCX_E2ECONN@MTC (for socket creation during channel set-up)
Uses	NONE

4.2.2 Channel Components

On purpose, the channel specifications have been kept as minimal as possible. The internal channel operation is left to the channel designer. Besides the cooperation between the channel monitoring and self logistics, only the interfaces to the MobiThin Client level have been detailed.

TCSCCX_SLM@MTC: Self Logistics Manager	
Functionality	<ul style="list-style-type: none"> ▪ Maintains the channel protocol state ▪ Ensures the proper working of the channel (e.g. handling channel protocol retransmissions) ▪ Handles monitoring alerts at the TCSCCX level. ▪ Instruct TCSCCX_MON@MTC components to provide additional information, to increase the monitor frequency, etc. ▪ Notifies higher level components (MTC_SLM) when the available algorithms/mechanisms are insufficient to resolve monitoring alerts.
Is used by	<ul style="list-style-type: none"> ▪ MTC_SLM (for setting channel specific protocol parameters)
Uses	<ul style="list-style-type: none"> ▪ MTC_SLM (when the channel specific mechanisms are insufficient to ensure the QoS/QoE policy definitions, under the given circumstances of device/network resources, application characteristics...) ▪ TCSCCX_E2ECORE@MTC (layer configuration)

TCSCCX_MON@MTC: Monitoring	
Functionality	<ul style="list-style-type: none"> ▪ Monitors channel state. ▪ Filters and summarizes channel monitoring information. ▪ Notifies MTC_SLM of monitoring alerts. ▪ Reports to higher layer components (MTC_MON).
Is used by	<ul style="list-style-type: none"> ▪ TCSCCX_E2ECORE (report monitoring information)
Uses	<ul style="list-style-type: none"> ▪ MTC_MON to report monitor information ▪ TCSCCX_E2ECORE (monitoring)

TCSCCX_E2ECORE@MTC: End-To-End Core	
Functionality	<ul style="list-style-type: none"> ▪ Coding and decoding of channel specific messages
Is used by	<ul style="list-style-type: none"> ▪ TCSCCX_SLM@MTC (parameter configuration) ▪ TCSCCX_MON@MTC (monitoring)
Uses	<ul style="list-style-type: none"> ▪ MTC_NET@MTC (send/receive bytes from the network) ▪ TCSCCX_MON@MTC (report monitoring information) ▪ Viewer (for rendering graphical output)

TCSCCX_E2ECONN@MTC: End-To-End Connection	
Functionality	<ul style="list-style-type: none"> ▪ Setting up a connection for channel X with the server-side channel X
Is used by	<ul style="list-style-type: none"> ▪ TCSC_E2ELOGISTICS_SESMAN@MTC (for starting/stopping the channel)
Uses	<ul style="list-style-type: none"> ▪ MTC_NET (for socket creation during channel set-up) ▪ TCSCCX_E2EAUTHZ@MTC (to check if user is authorized to start channel)

TCSCCX_E2EAUTHZ@MTC: End-To-End Authorization	
Functionality	<ul style="list-style-type: none"> ▪ Authorization (e.g. is the user allowed to use the print channel?)
Is used by	<ul style="list-style-type: none"> ▪ TCSCCX_E2ECONN@MTC (check if the user is allowed to start this channel)
Uses	<ul style="list-style-type: none"> ▪ MTC_NET@MTC (send/receive bytes from the network) ▪ TCSCCX_E2EAUTHZ@MTC (get the sessionTicket and use it for starting a channel connection)

TCSCCX_E2ELOGISTICS@MTC: End-To-End Logistics	
Functionality	<ul style="list-style-type: none"> ▪ Takes in charge the negotiation of needed resources at both end points of a channel. This is to ensure correct end-to-end session. ▪ Requests the reservation of needed resources at the channel level
Is used by	<ul style="list-style-type: none"> ▪ Any other component that needs resource reservation at the channel level
Uses	<ul style="list-style-type: none"> ▪ MTC_NET (send/receive bytes from the network)

5. SEQUENCE DIAGRAMS

5.1 IMAGE TRANSMISSION

Sequence Diagram: see Figure 7

Scenario: User interacting with the remote application in the User Session on the Thin Client Server

Precondition: A user session is established. The user is authenticated and authorized and has launched the MobiThin viewer on the client device, presenting the display of a running application.

Assumption: For the sake of clarity, both user events and graphic updates are transported over the same channel (TCSCCVideoGraphics). In a real deployment, separate channels for user events and graphic updates will be established, each having their own E2ECORE component.

Event: User generates an event, such as a keystroke or a mouse click.

Action:

- The user event is captured by MTC_VIEWER and forwarded to MTC_TCSCCVideoGraphics_E2ECORE.
- MTC_TCSCCVideoGraphics_E2ECORE encodes the user event in a channel specific protocol message.
- The bytes are forwarded to MTC_NET to be sent over the network to the server.
- MTS_TCSCCVideoGraphics receives the channel specific protocol message, decodes it and forwards the user event to the application.
- The application processes the user event. In response, the application user interfaced must be updated and the application issues the appropriate graphic calls to MTS_TCSCCVideoGraphicsE2ECORE
- The call is encoded in a channel specific protocol message and sent, via MTS_NET and MTC_NET over the network to the client.
- MTC_TCSCCVideoGraphicsE2ECORE decodes the channel specific protocol message and issues the appropriate rendering calls to the Viewer.

5.2 PROTOCOL ADAPTIVITY

Sequence Diagram: see Figure 8

Situation: The thin client protocol parameters are adapted to variations in the network environment.

Event: interference on wireless link results in bandwidth drop and increasing delay

Action:

- MTC_MON remarks the higher delays, and notifies MTC_SLM by generating a MonitorReport
- MTC_SLM tries to optimize the wireless link communication by changing a parameter via MTC_NET
- After a while, MTC_MON notices that the delay is still too high and issues a MonitorReport to MTC_SLM
- MTC_SLM decides that action is required at the server side. It generates an ActionReport and sends it, through a dedicated SLM channel to MTS_SLM@US.
- MTS_SLM receives the report and decides to change the parameters of the codec in the video channel TCSCCX. It contacts TCSCCX_SLM to modify some of its parameters. TCSCCX_SLM acknowledges the change of parameters.
- The delay decreases, and MTC_MON and MTS_MON@US will not generate any Monitoring Alert anymore.

5.3 STARTING UP CONTENTMANAGER COMPONENTS (*)

Sequence Diagram: see Figure 9

Situation: The MobiThin management framework is setting up a new User Session.

Precondition: Initialization at higher architectural levels (MS and TCSC) has been successfully completed. This process is described in full detail in the sequence diagrams in D4.1 – Phase I Component Interfaces and D4.4 – Phase II Component Interfaces. The “startSession” call has been issued to the TCSC_SESMAN@US component.

Action:

- TCSC_SESMAN@US starts the required ApplicationListeners. The ApplicationListener components are created through the ApplicationListenerFactory and started by invoking the start()-call.
- TCSC_SESMAN@US starts the required Convertors, for both events and graphics content. The process is similar to the ApplicationListeners: first, the objects are created through a ConverterFactory and then actually started by invoking the start()-call.
- Lastly, the channels are started by invoking startChannel() for every channel.

5.4 A NEW APPLICATION IS STARTED BY THE USER (*)

Sequence Diagram: see Figure 10

Situation: The user has started a new application, which is detected by the ContentManager.

Precondition: User Session components have been successfully started and initialized, according to section 5.3.

Action:

- The user starts a new application that generates graphics in the X11 format.
- X11GraphicsListener notices the new application and registers this application to the TCSC_SLM@US component. The graphics protocol (“X11”) is passed as an argument.
- X11GraphicsListener needs to know to which ContentConvertor it has to forward the graphical content for possible conversion. The appropriate ContentConvertor is retrieved through the “requestConvertor” call to TCSC_SLM@US.
- X11GraphicsListener sets up a connection with X11toBIFSCovertor through internal socket communication. X11GraphicsListener transcodes X11 content to BIFS.
- X11toBIFSCovertor encodes the BIFS to a bitstream ready for network transmission by TCSCC_BIFS_E2ECORE.

5.5 FORWARDING OF USER EVENTS TO THE APPLICATION (*)

Sequence Diagram: see Figure 11

Situation: The user has started a new application, which is detected by the ContentManager.

Precondition: User Session components have been successfully started and initialized, according to section 5.3.

Action:

- The user starts a new application that generates graphics in the X11 format.
- X11GraphicsListener notices the new application and registers this application to the TCSC_SLM@US component. The graphics protocol (“X11”) is passed as an argument.
- X11GraphicsListener needs to know to which ContentConvertor it has to forward the graphical content for possible conversion. The appropriate ContentConvertor is retrieved through the “requestConvertor” call to TCSC_SLM@US.
- X11GraphicsListener sets up a connection with X11toBIFSCovertor through internal socket communication. X11GraphicsListener transcodes X11 content to BIFS.
- X11toBIFSCovertor encodes the BIFS to a bitstream ready for network transmission by TCSCC_BIFS_E2ECORE.

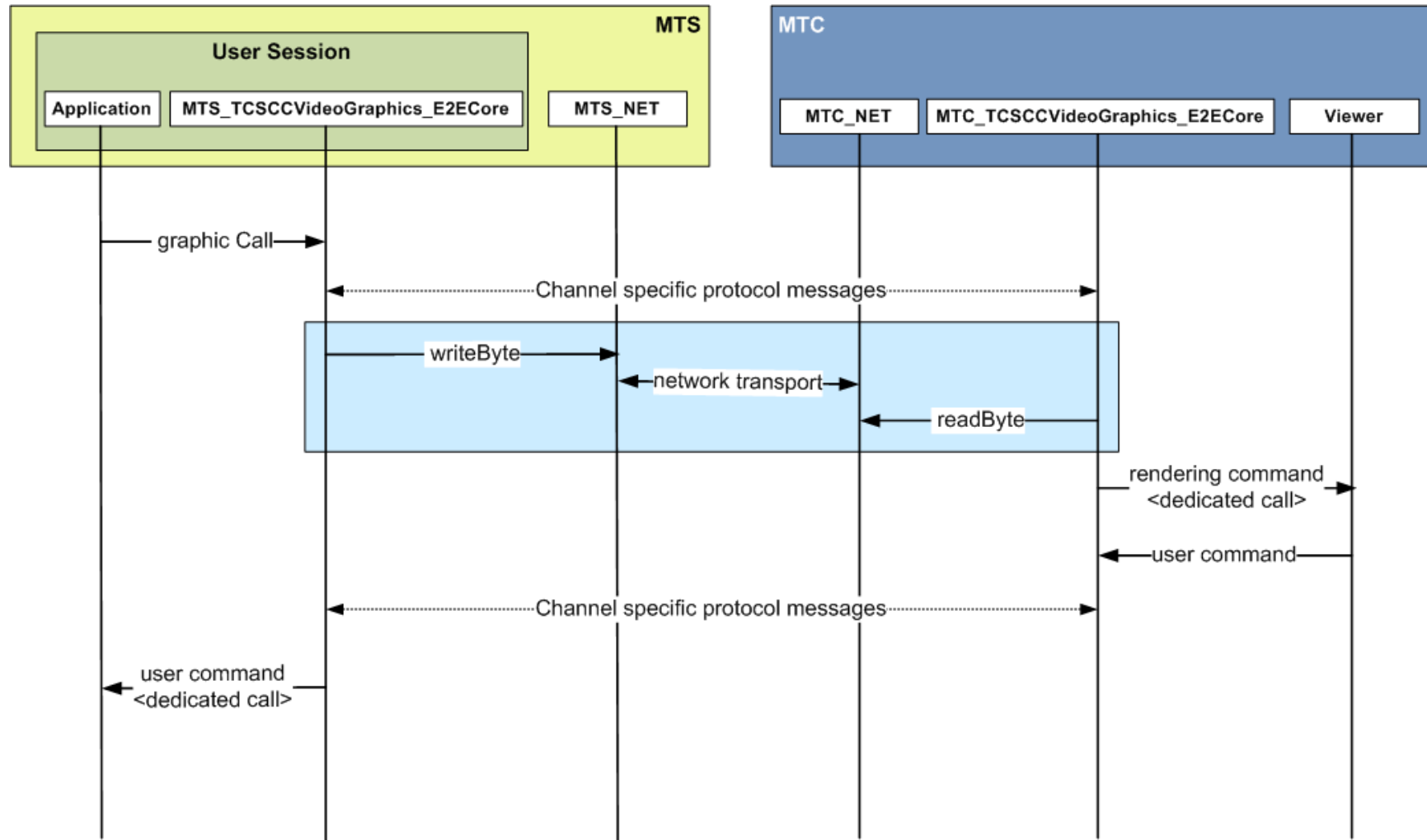


Figure 7 - Sequence diagram for the scenario "Image Transmission"

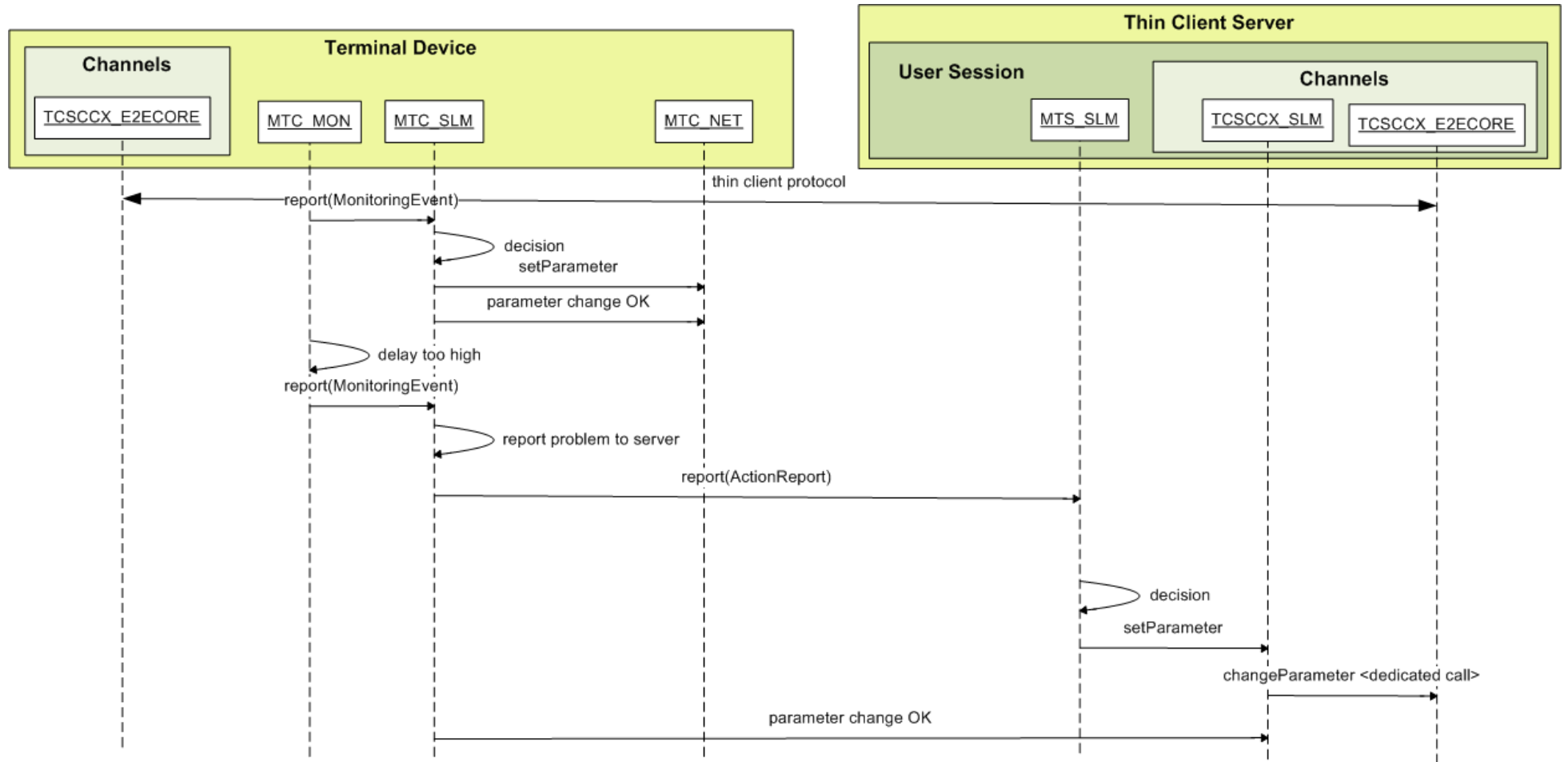


Figure 8 - Sequence diagram "Protocol Adaptivity"

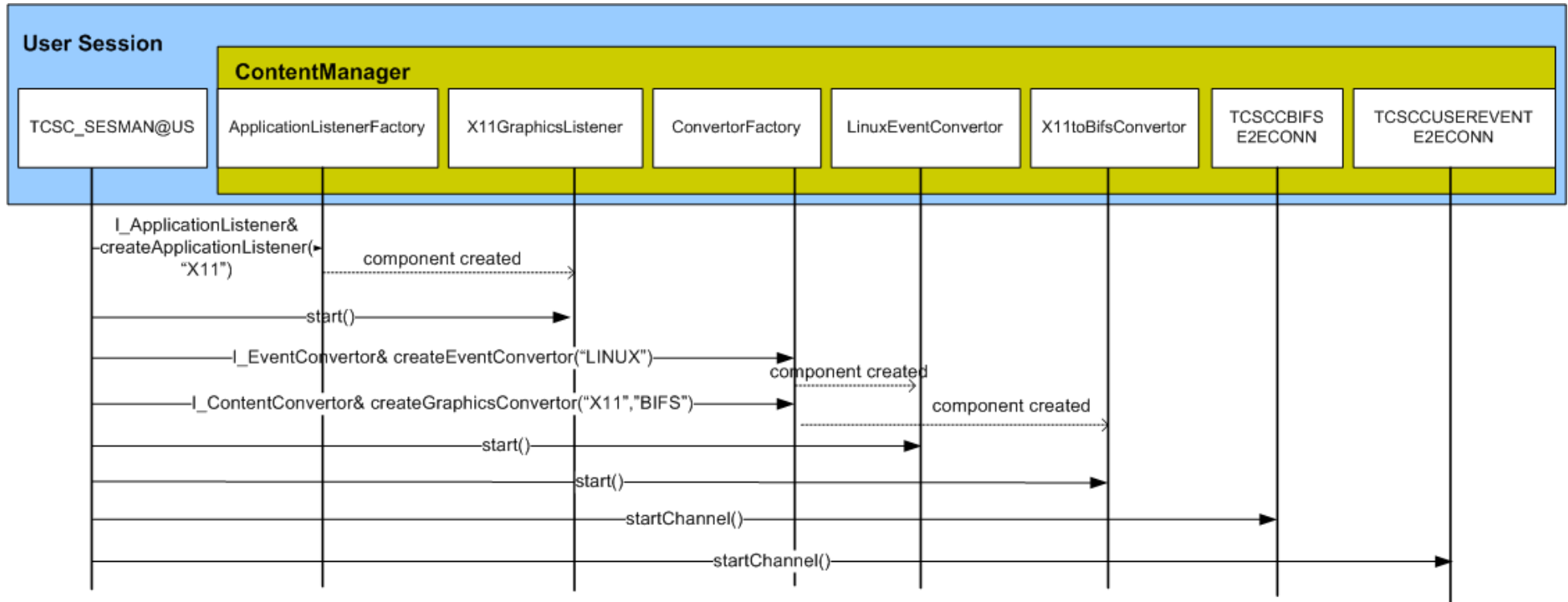


Figure 9 - Sequence Diagram "Starting ContentManager components"

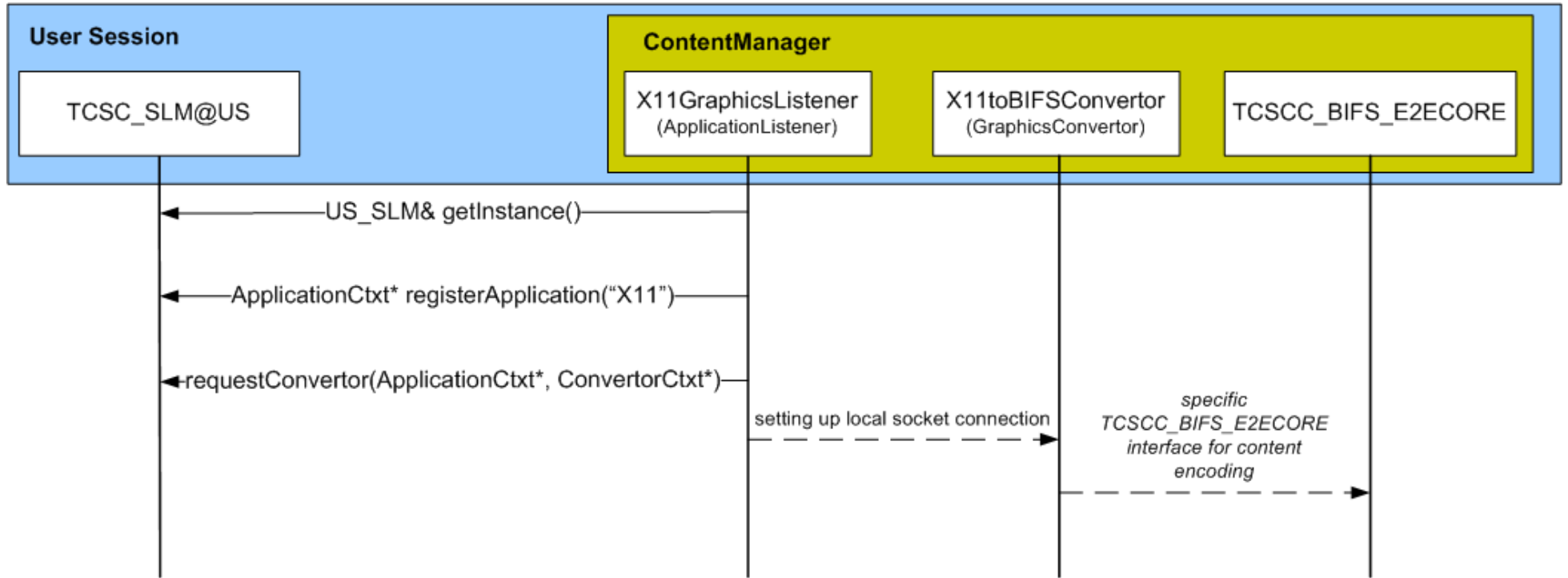


Figure 10 – Sequence Diagram “Handling a new application”

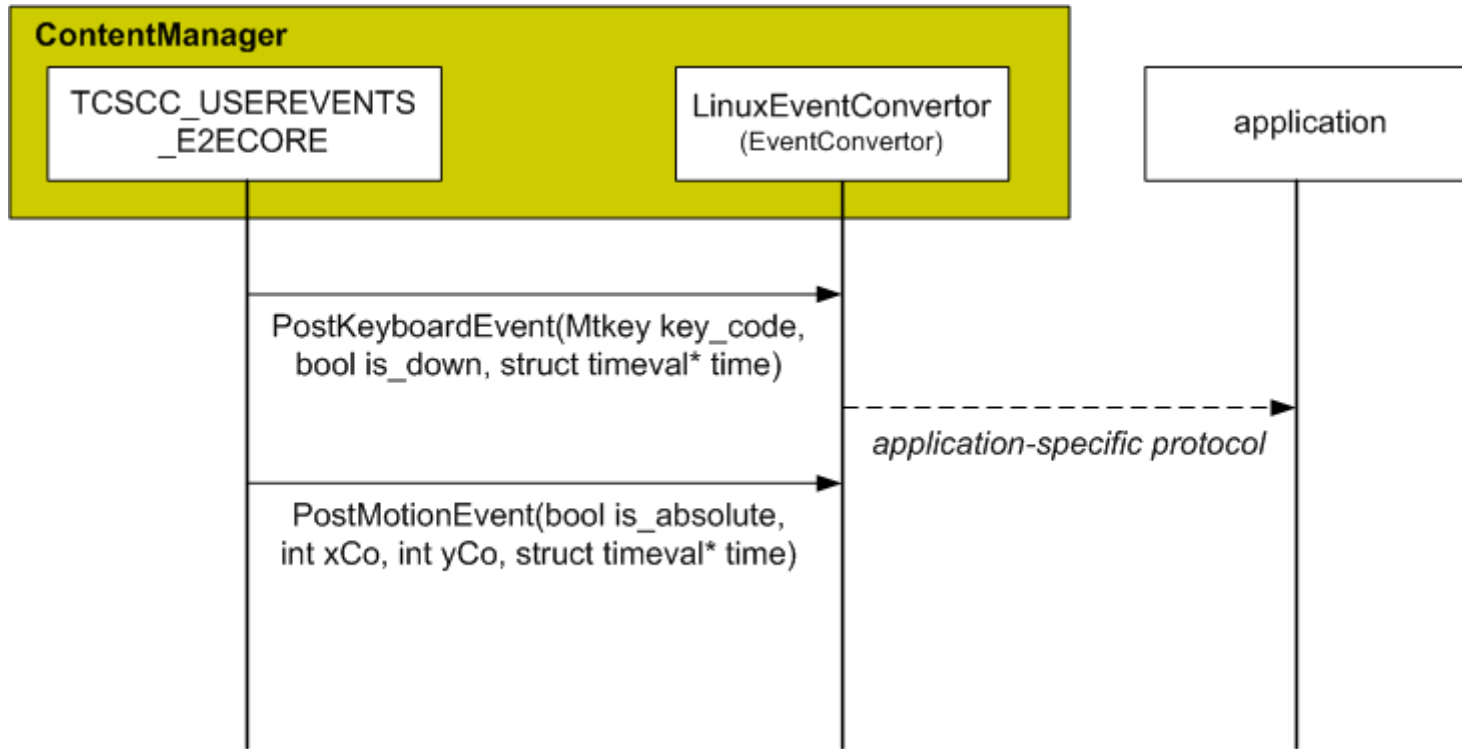


Figure 11 - Sequence Diagram "Forwarding User Events"

6. INTERFACE DEFINITION

This chapter details the interfaces between WP3 components. In chapter 3, the different architectural subsystems were identified, before detailing the functionality of each component. This chapter takes a similar approach. It starts with the description of interface of the WP3 SLM subsystem in section 6.1. The description of the generic interfaces of the WP4 architectural subsystems (administration, monitoring, entity subscription service, authentication, authorization), as well as the detailed interface description for every WP4 component, can be found in D4.1.

In section 6.2, the interfaces of WP3 components on the Thin Client Server are defined. Section 6.3 contains the interface definitions of WP3 components on the MobiThin Client.

6.1 GENERIC SLM INTERFACES

6.1.1 High-level view of Self Logistics Management

Self Logistic Management means ensuring the sane operation of all MobiThin components, in accordance to the configured policy. From a conceptual point of view, the MobiThin system can be seen as a large set of parameters: number of concurrent users, bandwidth used by channel X of user Y, cpu load of Thin Client Server X, etc. This is shown in Figure 12.

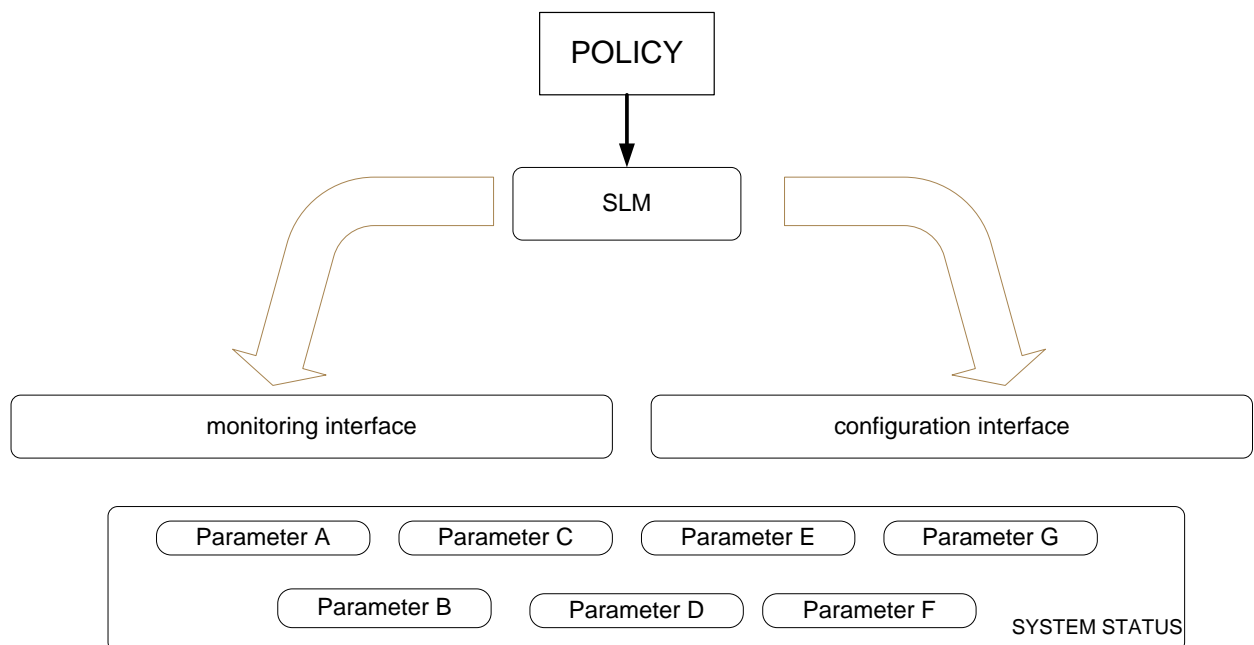


Figure 12 – The SLM monitors and configures system parameters.

The basic SLM operation is a MAPE loop: Monitor – Analyze – Plan – Execute. System status parameters are monitored and analyzed. If policy rules are violated, some parameters will be reconfigured. This means that the SLM only sees “parameters”, and two interfaces are required: a monitoring and a configuration interface.

The underlying monitoring framework, described in D4.1, continuously gathers system status information. This process is often referred to as “logging”. The logging is configured by the system administrator, or possibly by the SLM. A low-level configuration interface is described in D4.1

However, the information provided by the logging might not be sufficient for proper operation of the SLM. Some examples:

- The SLM might need summarized monitoring data (e.g. the average of parameter X over the past 10 minutes). Thus, the monitoring framework must be able to compute statistics over the logging information.
- In order to evaluate the effectiveness of some parameter reconfiguration, the SLM might want to intensify the monitoring of some parameter (e.g. every 1 second instead of every 10 seconds) in order to evaluate the effectiveness.

Besides a low-level interface to configure the logging, a higher level interface “SLM_MON” is required. Being WP4 scope, a complete description of the monitoring framework and both interfaces can be found in D4.1. To keep this deliverable consistent, a copy of the definition of high-level SLM_MON interface is included in section 6.1.1.1.

Besides the SLM_MON interface, in Figure 12 a configuration interface is presented. This interface offers the functionality to set system parameters. As presented in section 3.3.2, the SLM architectural subsystem is hierarchically structured. A SLM component can either configure parameters at the architectural level it is situated, or can instruct the SLM located on a lower level. Thus, the configuration interface of Figure 12 is composed of two interfaces:

- inter-SLM: to report to a higher level SLM when a problem cannot be solved, or to instruct a lower layer SLM
- setting parameters at the level of the SLM:
 - o at channel level, the SLM@TCSCCX instructs the TCSCCX. This interface is channel specific.
 - o at all other levels, the “Administration” interface described in D4.1 is used.

Figure 13 gives an overview of all the interfaces in which the SLM is involved.

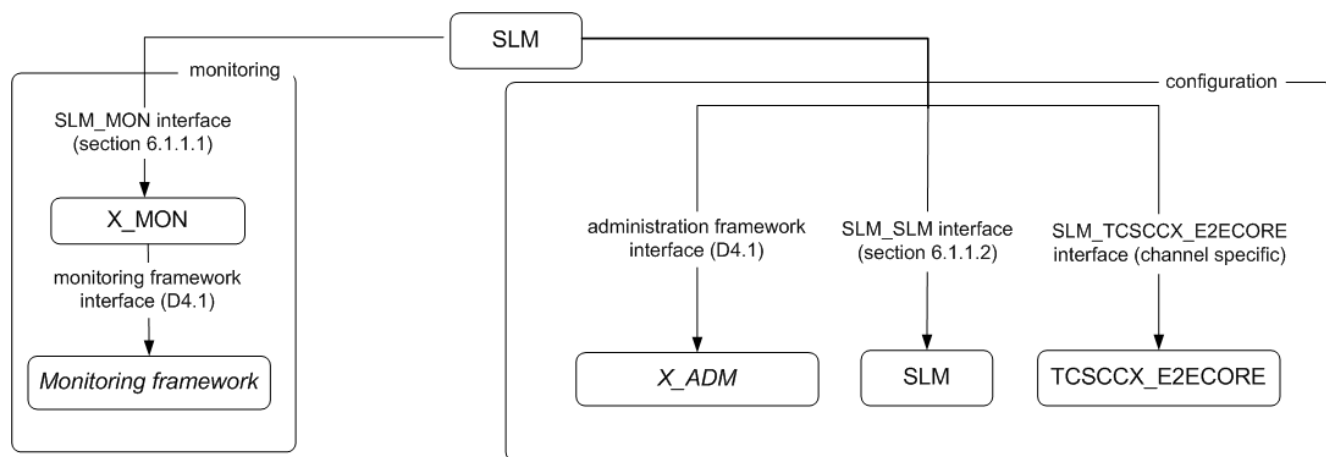


Figure 13 - The interfaces of the SLM for monitoring and configuration. For monitoring, the SLM_MON interface is used. For configuration purposes, the interface is dependent on the component to be configured: a parameter on the same level as the SLM through the administration framework, communication with another SLM component through the SLM interface, or channel configuration through the SLM_TCSCCX_E2ECORE interface.

In the next two sections the SLM_MON and SLM_SLM interfaces are defined. Both interfaces make use of the same helper classes that are detailed in Appendix A. This reflects the tight coupling of both interfaces around the “Parameter” concept, as previously illustrated in Figure 12.

6.1.1.1 SLM_MON INTERFACE

This interface is constructed according to the “event-listener” paradigm, illustrated in Figure 14. A Monitoring Agent monitors multiple parameters, such as packet error rate, bitrate, etc. At regular intervals, or when some threshold is exceeded, it sends MonitoringEvents to the MonitoringEventListeners that are registered to the MonitoringAgent. Every MonitoringEventListener declares to the MonitoringAgent of which monitored parameters it would like to receive MonitoringEvents. Furthermore, for each parameter-eventlistener combination, a report mode must be specified. This report mode can be periodic, threshold based, or a combination of both. In the case of threshold based event reporting, a MonitoringEvent is only generated when a monitored parameter exceeds a certain threshold.

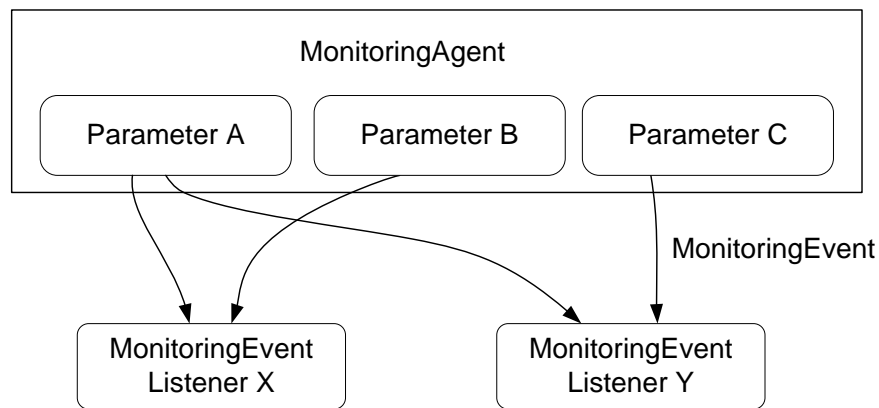


Figure 14 - Interfaces MonitoringAgent and MonitoringEventListener. Listener X has registered itself to the MonitoringAgent to receive MonitoringEvents about Parameters A and B, while listener Y has registered itself to receive MonitoringEvents about Parameters A and C.

6.1.1.1.1 MonitoringAgent

Parameter[] getMonitorableParameters()	
Functionality	Returns a list of Parameters that the Monitoring System can monitor (the list will actually be restricted to the “visible” parameters, according to the node where the SLM component is placed).
Arguments	None
Return value	An array of Parameters

void monitor(IN Parameter p, IN boolean start)	
Functionality	Starts or stops the monitoring the Parameter p
Arguments	Parameter p Boolean start
Return value	None
Remark	Throws a “IsAlreadyMonitoredException” when the parameter is already monitored.

boolean isMonitored(IN Parameter p)	
Functionality	Check if a parameter is already monitored.
Arguments	Parameter p
Return value	True when the Monitoring System already monitors this parameter

boolean isMonitorable(IN Parameter p)	
Functionality	Returns true when the Monitoring System can monitor this Parameter
Arguments	Parameter p
Return value	True if the Monitoring System can monitor this parameter.

void setMeasurementInterval(IN Parameter p, IN int measurementInterval)		
Functionality	Sets the measurement interval of Parameter	
Arguments	Parameter p int measurementInterval	The parameter to be monitored The time in milliseconds between subsequent measurements
Return value	None.	

int getMeasurementInterval(IN Parameter p)		
Functionality	Returns the time in ms between subsequent measurements of the specified Parameter p	
Arguments	Parameter p	The parameter to be monitored
Return value	The time between subsequent measurements of the specified Parameter (in milliseconds)	

void addReportListener(IN Parameter p, IN ReportListener listener, IN ReportMode mode)		
Functionality	Registers a ReportListener to receive MonitoringReports about Parameter p. Report mode is specified in ReportMode.	
Arguments	Parameter p ReportListener listener ReportMode mode	Parameter to be monitored Listener for MonitoringReports ReportMode for MonitoringReports to be sent to the client
Return value	None	

void removeReportListener(IN ReportListener listener)		
Functionality	Deregisters a ReportListener so it does not receive any MonitoringReports anymore.	
Arguments	ReportListener listener	Listener to be deregistered
Return value	None	

void setHistoryPeriod(IN Parameter p, IN int historyPeriod)		
Functionality	Sets how long monitoring measurements of Parameter p are archived	
Arguments	Parameter p int historyPeriod	Parameter to be monitored Period, expressed in milliseconds, during which a monitoring value must be archived in the sensor.
Return value	None.	

int getHistoryPeriod(IN Parameter p)		
Functionality	Returns the archive period of monitoring values of Parameter p	
Arguments	Parameter p	Parameter to be monitored
Return value	Archive period of monitoring values of Parameter p, expressed in milliseconds	

MonitoringReport calculateStatistics(IN Parameter p, IN String operation, IN String measurementPeriod)	
Functionality	Instructs the Monitoring System to perform an aggregation operation on its monitor data of Parameter p. The calculator will use the specified operation (ADD, AVERAGE, MIN, MAX) on the data available over the specified period. This aggregate value will then be returned in a MonitorReport.

Arguments	Parameter p String operation String measurementPeriod	The parameter to be monitored. The operation to be executed on the monitoring data. This parameter can be ReportMode.STAT_ADD, ReportMode.STAT_AVERAGE, ReportMode.STAT_MIN, ReportMode.STAT_MAX. The period over which the statistics have to be calculated.
Return value	MonitoringReport containing the requested data	
Remark	Throws a BadMeasurementPeriodException when the specified measurementPeriod argument is greater than the HistoryPeriod set through setHistoryPeriod.	

MonitoringReport getImmediateReport(IN Parameter p)		
Functionality	Instructs the Monitoring System to return the latest version of the value of Parameter p.	
Arguments	Parameter p	The parameter to be monitored.
Return value	MonitoringReport containing the requested data	

6.1.1.1.2 MonitoringEventListener

void report(IN MonitoringReport r)		
Functionality	Called by Monitoring System when a MonitoringReport is available.	
Arguments	MonitoringReport r	MonitoringReport to be delivered to the MonitoringReportListener
Return value	None.	

6.1.1.2 SLM_SLM INTERFACE

The SLM_SLM interface is used for inter-SLM communication.

Value getParameterValue(Parameter p)		
Functionality	Returns the current value of the Parameter p	
Arguments	Parameter p	Parameter p of which the Value is requested
Return value	Returns the current Value of the Parameter p	

void setParameter (Parameter p, Value v)		
Functionality	Sets the value of Parameter p to the specified Value v	
Arguments	Parameter p Value v	Parameter of which the value has to be updated Value to which the parameter must be set
Return value	none	
Remark	Throws a “BadParameterException” when the Value does not match with the specified Parameter.	

void requestAction(ActionReport report)		
Functionality	Request an action from the SLM component. The ActionReport contains additional information.	
Arguments	ActionReport report	ActionReport specifying the action requested
Return value	None	

6.2 THIN CLIENT SERVER INTERFACES

On the Thin Client Server, WP3 components are situated at User Session (section 6.2.1) and the Channel level (section 6.2.2). For every component, a figure is presenting the other components to which it communicates. The arrows point to the component that is called or receives a protocol message. The arrows that point to a component, indicate an interface that is exported by that component. For each component in this section, all exported interfaces are defined.

6.2.1 Components at user session level

6.2.1.1 MTS_NET@US

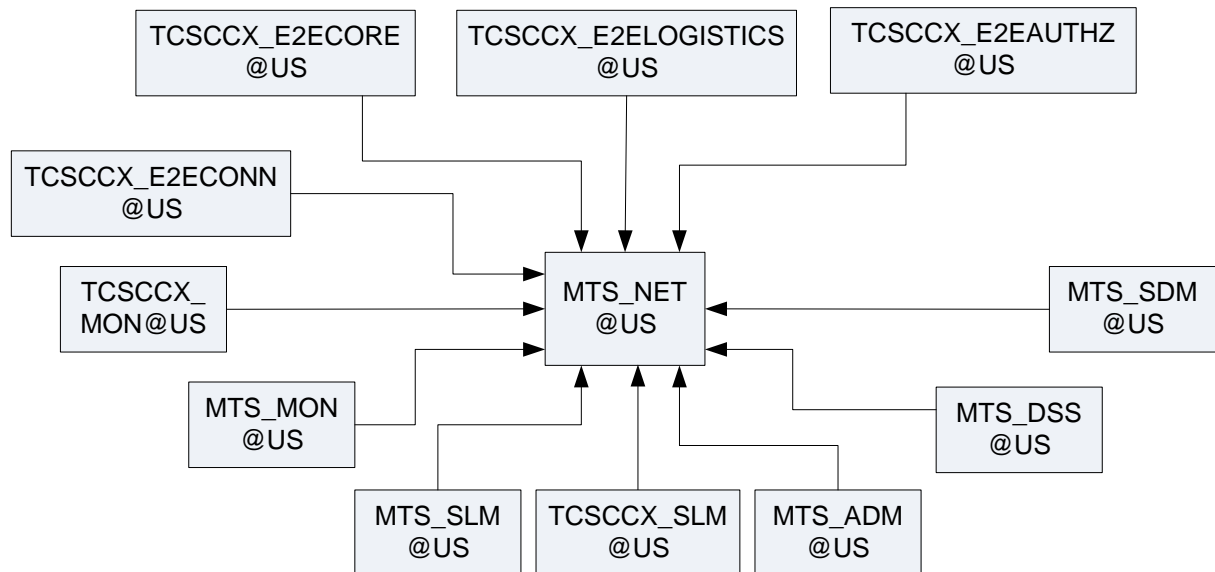


Figure 15 - Interfaces exported by MTS_NET@US

Two interfaces are exported by MTS_NET@US

- the generic “administration” interface is exported to MTS_ADM@US. This interface is described in D4.1
- a “networking” interface is exported to TCSCCX_E2ECORE@US, TCSCCX_E2ELOGISTICS@US, TCSCCX_E2EAUTHZ, TCSCCX_E2EAUTHN@US, MTS_SDM@US, MTS_ESS@US, MTS_SLM@US, MTS_MON@US, TCSCCX_MON@US, TCSCCX_E2ECONN@US

For MTS_NET@US, a general network communication API will be used. Some candidates have already been identified:

- the Adaptive Communication Environment (ACE) [6]
- the BOOST library [7]

6.2.1.2 MTS_SLM@US

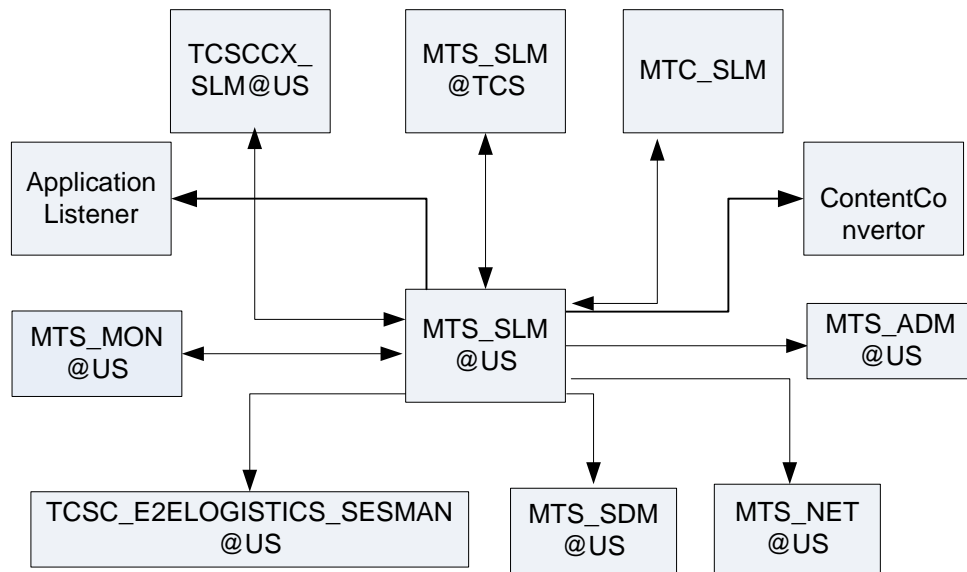


Figure 16 - Interfaces exported by MTS_SLM@US

MTS_SLM@US exports the “SLM_SLM” interface, described in section 6.2.1.2, to the following components:

- TCSCCX_SLM@US
- MTS_SLM@TCS
- MTC_SLM

MTS_SLM@US exports the “SLM_MON” interface, described in section 6.1.1.1, to MTS_MON@US.

6.2.1.3 APPLICATIONLISTENER@US (*)

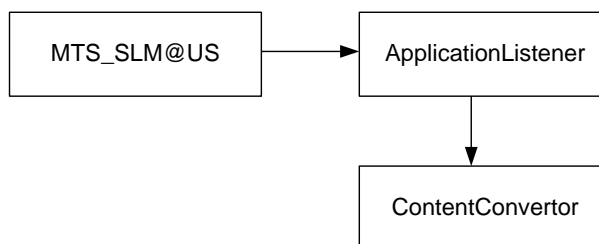


Figure 17 - Interfaces exported by ApplicationListener@US

ApplicationListener components must at least implement the following interface.

int start(void)	
Functionality	Starts and initializes the ApplicationListener component
Arguments	none /
Return value	0 if successful, -1 on error

int stop(void)	
Functionality	Stops the ApplicationListener component
Arguments	none /
Return value	0 if successful, -1 on error

6.2.1.4 EVENTCONVERTOR@US (*)

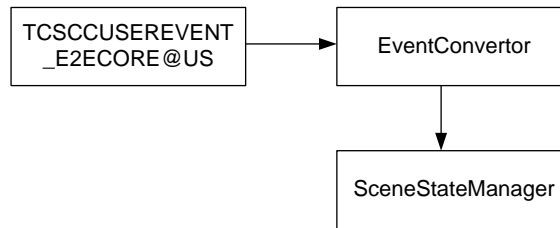


Figure 18 - Interfaces exported by EventConvertor

EventConvertor components must at least implement the following interface:

int start(void)	
Functionality	Starts and initializes the EventConvertor component
Arguments	none /
Return value	0 if successful, -1 on error

int stop(void)	
Functionality	Stops the EventConvertor component
Arguments	none /
Return value	0 if successful, -1 on error

int PostKeyboardEvent(MTkey key_code, int is_down, struct timeval* time)		
Functionality	posts a keyboard event to the renderer	
Arguments	MTkey key_code	the code of the key pressed/released, for a list of codes: see appendix B
	int is_down	indicates if the key was pressed or released, possible values: MTKEY_DOWN, MTKEY_UP (defined in MTinputevent.h)
	struct timeval* time	Pointer to struct timeval, containing the moment at which the key event occurred
Return value	0 if successful, -1 on error	

int PostMotionEvent(int is_absolute, int X, int Y, struct timeval* time)		
Functionality	posts a motion event of the pointing device to the renderer	
Arguments	int is_absolute	Indicates if the (X,Y) coordinates provided are absolute or relative. Possible values are MTRELATIVE, MTABSOLUTE (defined in MTinputevent.h)
	int X, int Y	new coordinates of the pointing device
	struct timeval* time	Pointer to struct timeval, containing the moment at which the key event occurred
Return value	0 if successful, -1 on error	

int PostButtonEvent(MTbutton button_code, int is_down, struct timeval* time)		
Functionality	posts a button event (of the pointing device) to the renderer	
Arguments	MTbutton button_code	the code of the button pressed/released, for a list of codes: see appendix B
	int is_down	indicates if the key was pressed or released, possible values: MTBUTTON_DOWN, MTBUTTON_UP (defined in MTinputevent.h)
	struct timeval* time	Pointer to struct timeval, containing the moment at which the key event occurred
Return value	0 if successful, -1 on error	

6.2.1.5 CONTENTCONVERTOR@US (*)

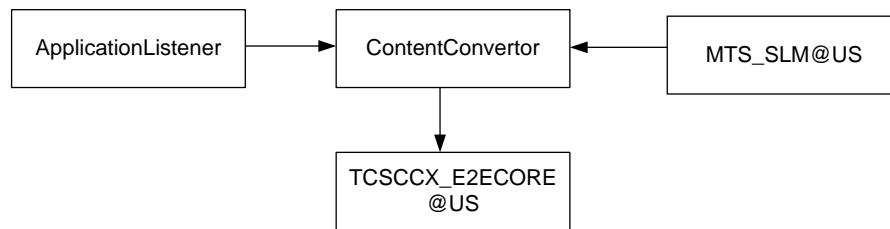


Figure 19: Interfaces exported by ContentConvertor

ContentConvertor components should at least implement the following interface.

int start(void)		
Functionality	Starts and initializes the ContentConvertor component	
Arguments	none	/
Return value	0 if successful, -1 on error	

int stop(void)		
Functionality	Stops the ContentConvertor component	
Arguments	none	/
Return value	0 if successful, -1 on error	

void getConvertorCtxt(ConvertorCtxt* ctxt)		
Functionality	Acquires information about the context of the convertor	
Arguments	ConvertorCtxt* ctxt	Pointer to struct ConvertorCtxt to be filled in. The pointer should point to initialized memory.
Return value	none	

6.2.2 Components at channel level

The interface below is mandatory for all channels. MobiThin compatible channels should at least include these interfaces.

6.2.2.1 TCSCCX_SLM@US

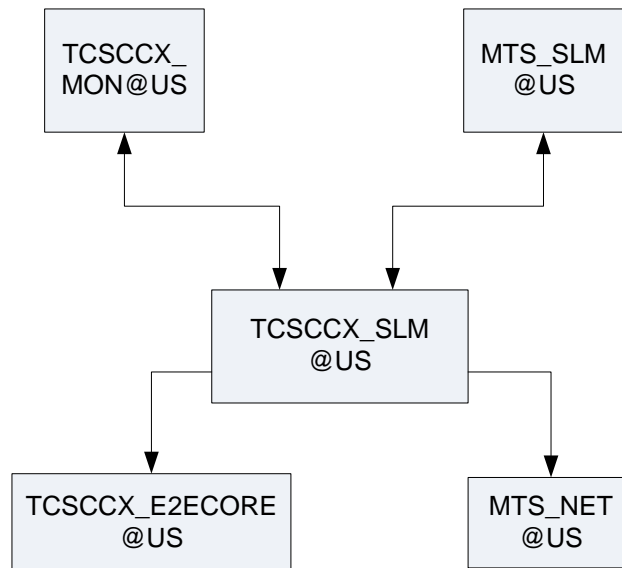


Figure 20 - Interfaces exported by TCSCCX_SLM

TCSCCX_SLM@US exports the “SLM_MON” interface, described in section 6.1.1.1 to TCSCCX_MON@US.

TCSCCX_SLM@US exports the “SLM_SLM” interface, described in section 6.2.1.2, to MTS_SLM@US.

6.2.2.2 TCSCCX_E2ECORE@US

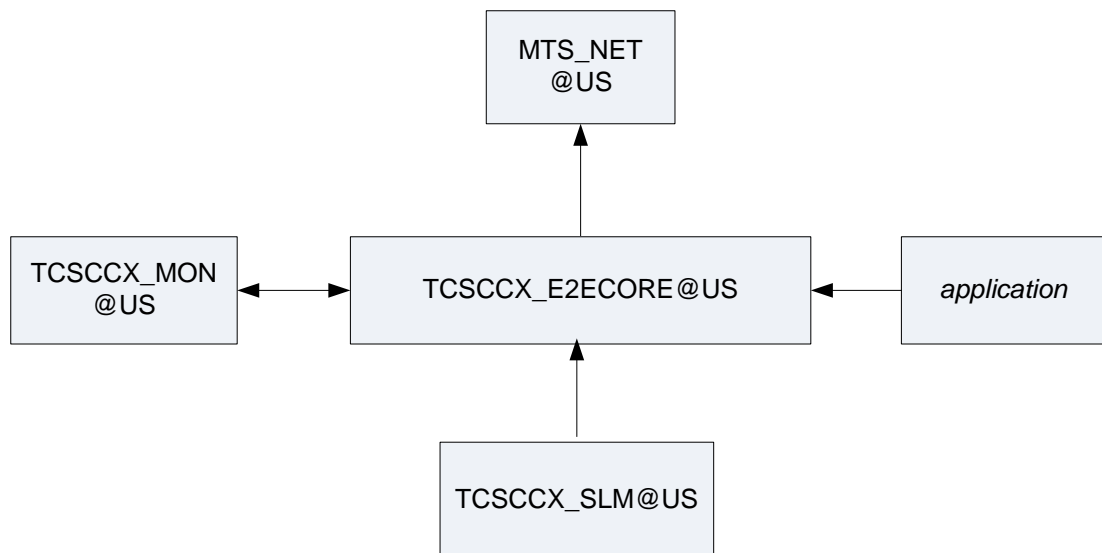


Figure 21 - Interfaces exported by TCSCCX_E2ECORE

TCSCCX_E2ECORE@US only exports interfaces to two components:

- the application, for audio/video rendering, for user events, etc..
- TCSCCX_SLM@US, for channel specific parameter configuration

Since both interfaces are channel specific, MobiThin does not define these interfaces.

6.2.2.3 TCSCCX_MON@US

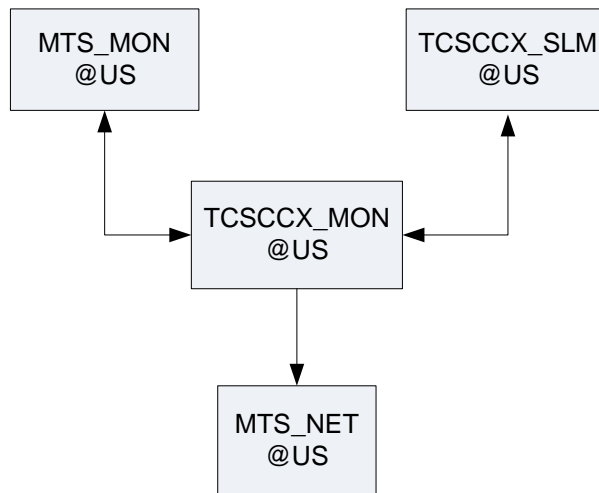


Figure 22 - Interfaces exported by TCSCCX_MON@US

TCSCCX_MON@US exports the generic monitoring interface, described in D4.1, to MTS_MON@US.

TCSCCX_MON@US exports the “SLM_MON” interfaces, described in section 6.1.1.1, to TCSCCX_SLM@US.

6.2.2.4 TCSCCX_E2ECONN@US

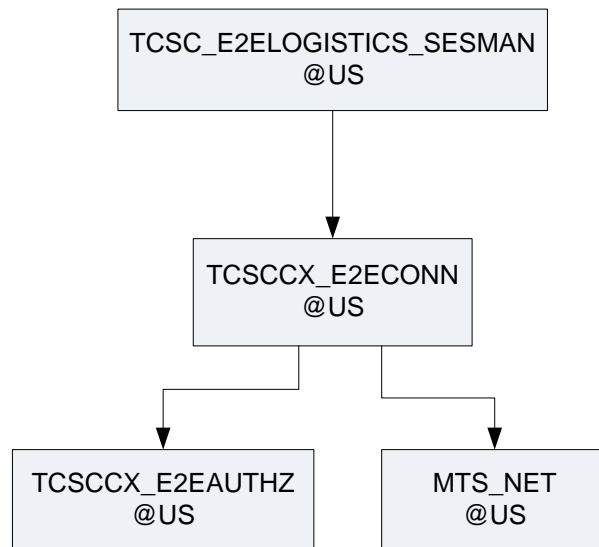


Figure 23 - Interfaces exported by TCSCCX_E2ECONN@US

TCSCCX_E2ECONN@US exports the following interface to TCSC_E2ELOGISTICS_SESMAN@US:

boolean startChannel ()	
Functionality	Initialize channel, creates listening sockets for incoming connections.
Arguments	None
Return value	Returns true if channel was successfully started.

6.2.2.5 TCSCCX_E2EAUTHZ@US

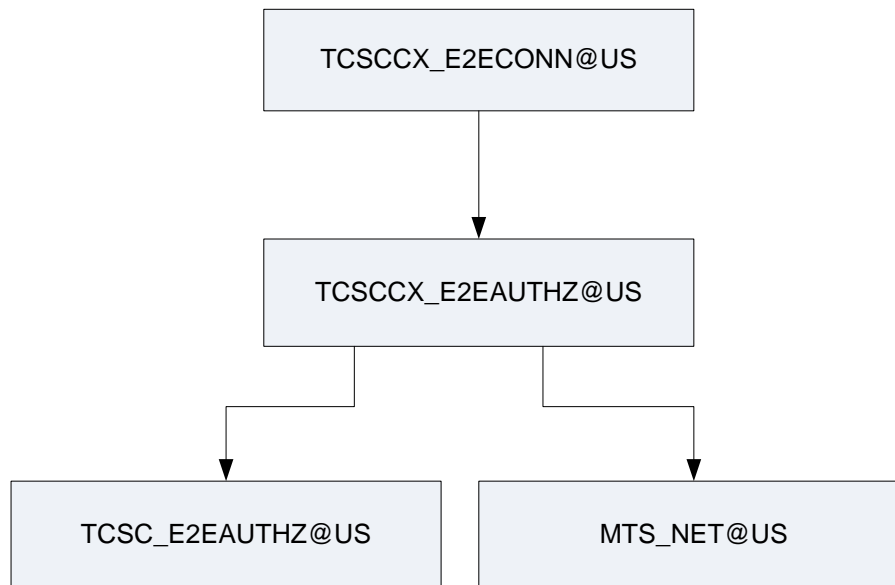


Figure 24 - Interfaces exported by TCSCCX_E2EAUTHZ@US

Authorization is one of the generic interfaces defined in section 6.2.4 in D4.1.

6.2.2.6 TCSCCX_E2ELOGISTICS@US

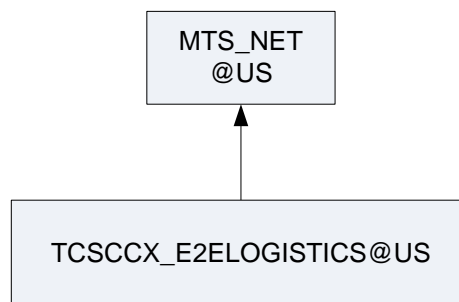


Figure 25 - Interfaces exported by TCSCCX_E2ELOGISTICS@US

TCSCCX_E2ELOGISTICS does not export any interface to a component.

6.3 MOBITHIN CLIENT INTERFACES

6.3.1 General Components

6.3.1.1 MTC_SLM

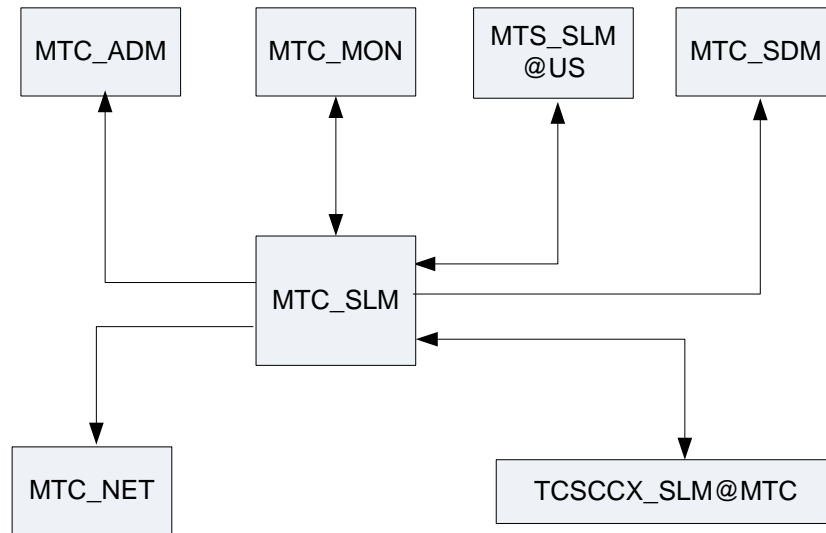


Figure 26 - Interfaces exported by MTC_SLM

MTC_SLM exports the “SLM_SLM” interface, described in section 6.2.1.2, to TCSCCX_SLM@MTC.

MTC_SLM exports the “SLM_MON” interface, described in section 6.1.1.1, to MTC_MON.

6.3.1.2 MTC_NET

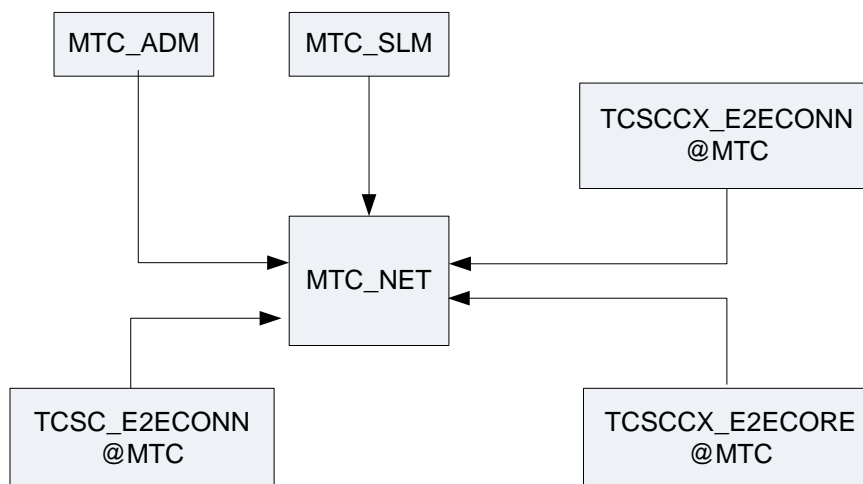


Figure 27 - Interfaces exported by MTC_NET

Two interfaces are exported by MTC_NET

- the generic “administration” interface is exported to MTC_ADM. This interface is described in D4.1
- a “networking” interface is exported to TCSCCX_E2ECORE@MTC, TCSCCX_E2ECONN@MTC and TCSC_E2ECONN@MTC.

For MTC_NET, a general network communication API will be used. Some candidates have already been identified:

- the Adaptive Communication Environment (ACE) [6]
- the BOOST library [7]

6.4 COMPONENTS PER CHANNEL

6.4.1.1 TCSCCX_E2ECONN@MTC

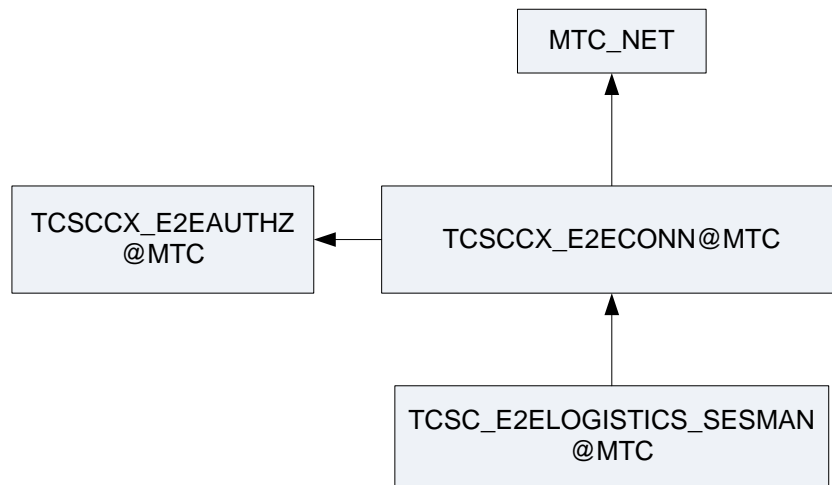


Figure 28 - Interfaces exported by TCSCCX_E2ECONN@MTC

The following interface is exported by TCSCCX_E2ECONN@MTC to TCSC_E2ELOGISTICS_SESMAN@MTC:

boolean startChannel (String serverIP)		
Functionality	Starts up the channel and connects to the server.	
Arguments	String serverIP	The IP address of the server to set-up the channel
Return value	Returns true when channel is successfully established.	

6.4.1.2 TCSCCX_E2EAUTHZ@MTC

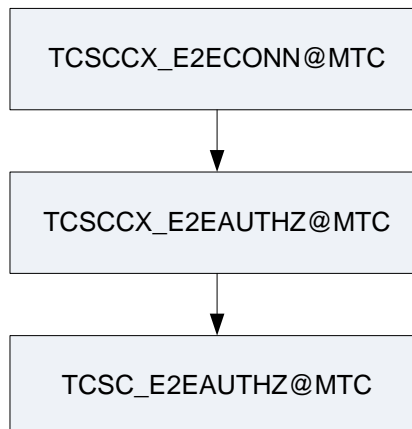


Figure 29 - Interfaces exported by TCSCCX_E2EAUTHZ@MTC

Authorization is one of the generic interfaces defined in section 6.2.4 in D4.1.

6.4.1.3 TCSCCX_E2ECORE@MTC

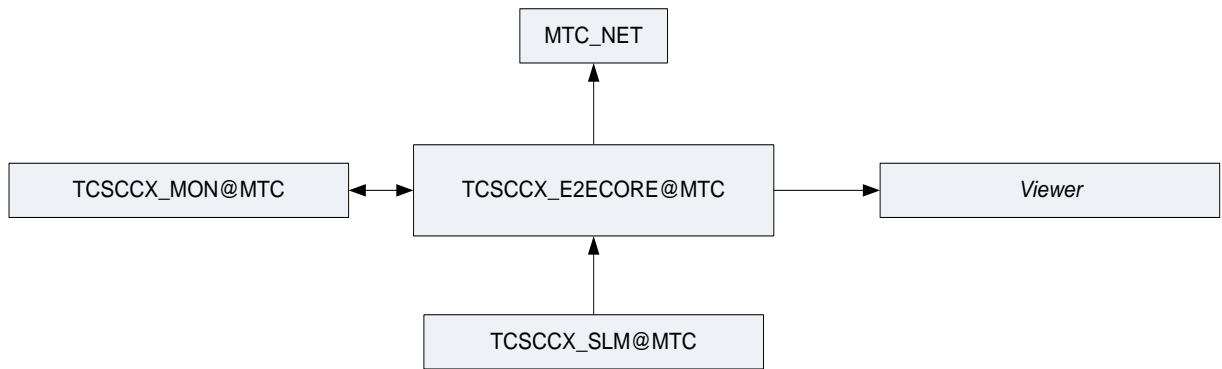


Figure 30 - Interfaces exported by TCSCCX_E2ECORE@MTC

The interface exported by TCSCCX_E2ECORE@MTC to TCSCCX_SLM@MTC is channel specific and left to the channel designers.

The interface exported by TCSCCX_E2ECORE@MTC tot TCSCCX_MON@MTC is channel specific and left to the channel designers.

6.4.1.4 TCSCCX_SLM@MTC

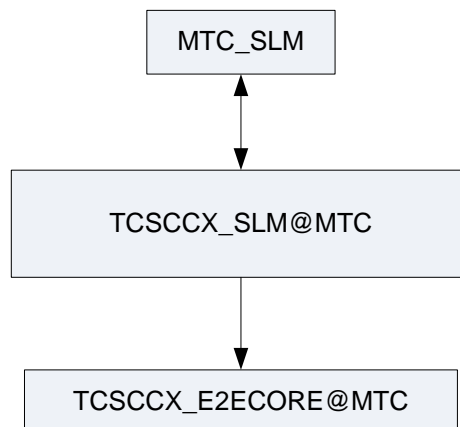


Figure 31 - Interfaces exported by TCSCCX_SLM@MTC

TCSCCX_SLM@MTC exports the “SLM_SLM” interface, described in section 6.2.1.2, to MTC_SLM.

6.4.1.5 TCSCCX_E2ELOGISTICS@MTC

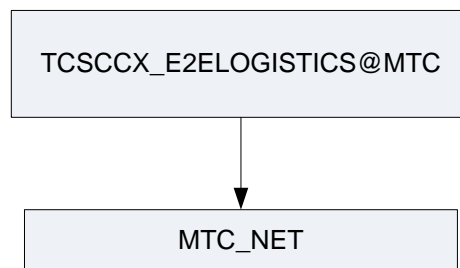


Figure 32 - Interfaces exported by TCSCCX_E2ELOGISTICS@MTC

TCSCCX_E2ELOGISTICS does not export any interface to a component.

6.4.1.6 TCSCCX_MON@MTC

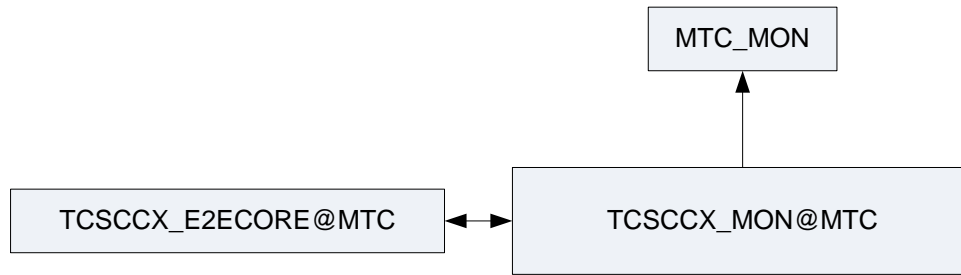


Figure 33 - Interfaces exported by TCSCCX_MON@MTC

TCSCCX_MON@MTC exports the generic monitoring interface, described in D4.1, to MTC_MON

7. REFERENCES

- [1] The XLib manual – <http://tronche.com/gui/x/xlib/>
- [2] The Win32 Debugging Application Programming Interface – <http://msdn.microsoft.com/en-us/library/ms809754.aspx>
- [3] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood & Andy Hopper, "Virtual Network Computing", *IEEE Internet Computing*, Vol.2 No.1, Jan/Feb 1998 pp33-38
- [4] Jean-Claude Dufourd, Olivier Avaro, Cyril Concolato, "An MPEG Standard for Rich Media Services," *IEEE MultiMedia* ,vol. 12, no. 4, pp. 60-68, October-December, 2005
- [5] Advanced Linux Sound Architecture (ALSA) - <http://www.alsa-project.org/alsa-doc/alsa-lib/>
- [6] Adaptive Communication Environment – <http://www.cs.wustl.edu/~schmidt/ACE.html>
- [7] Boost C++ library – <http://www.boost.org>

8. APPENDIX A: HELPER CLASSES FOR SLM_MON AND SLM_SLM INTERFACES

8.1 HELPER CLASSES

In the definition of this interface, some helper classes were mentioned. These are detailed in this section.

8.1.1 Parameter

This is a wrapper class for some parameter in the MobiThin system. Concrete parameters will be a subclass of this parameter.

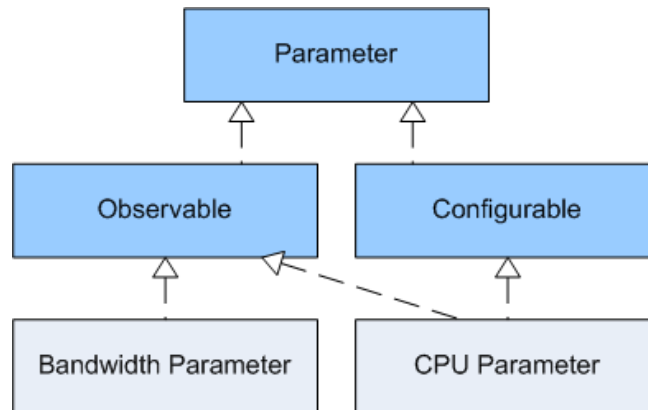


Figure 34 - Different kinds of parameters

We see parameter as an interface containing the following field (not directly accessible)

- String parameterName

There are two kinds of parameters. This results in two subinterfaces, Observable and Configurable. Observable parameters are measured and cannot be altered using the interface. Configurable parameters are only to be changed using the Configurable interface. In Figure 34, some examples are given of how to use this mechanism. Some parameters that will be used in the MobiThin system, like bandwidth, can only be observed. Therefore the parameter Bandwidth will only implement the Observable interface. The same method is used for Configurable-only parameters. When the parameter is both observable and configurable, it implements both interfaces.

String getName()	
Functionality	Returns the name of the parameter
Arguments	None
Return value	The name of the parameter

8.1.1.1 OBSERVABLE

void setHistoryLength(int interval)	
Functionality	Sets the interval over which the values are stored
Arguments	int interval Timespan in milliseconds
Return value	None

void setMeasurementPeriod(int period)	
Functionality	Sets the period between subsequent measurements

Arguments	int period	Timespan in milliseconds
Return value	None	

Value getMinimum()		
Functionality	Returns the minimum value of the observed parameter	
Arguments	None	
Return value	The minimum value measured in the past measurement interval	

Value getMaximum()		
Functionality	Returns the maximum value of the observed parameter	
Arguments	None	
Return value	The minimum value measured in the past measurement interval	

Value getCurrent()		
Functionality	Returns the current value of the observed parameter	
Arguments	None	
Return value	The current value, can lead to a forced measurement	

Value[] getHistory()		
Functionality	Returns all measurement values from the past measurement interval	
Arguments	None	
Return value	All values measured in the past measurement interval	

8.1.1.2 CONFIGURABLE

void setValue(Value v)		
Functionality	Sets the value of the configurable parameter	
Arguments	Value v	The value to set the parameter to
Return value	None	

8.1.2 Value

This is a wrapper class for the value of some parameter.

It contains the following fields (not directly accessible)

- Parameter p: the parameter of which this is the Value

Time_t getTimestamp()		
Functionality	Gets the timestamp on which the value last changed	
Arguments	None	
Return value	The timestamp of the value	

int compareTo(Value v)		
Functionality	Compares Value v with the value of the called object	
Arguments	Value v	Value to compare to
Return value	-1 if smaller, 0 if equal, +1 if greater	

8.1.3 Report, MonitoringReport, ActionReport

The class Report is a superclass of MonitoringReport and ActionReport.

By itself, the class only contains:

- String reportID: a unique identifier for the report
- Time t: time when this report was generated
- MonitoringAgent agent: originator of this report

The class MonitoringReport contains:

- an array of Values of monitored parameters

The class ActionReport contains:

- an array of Values of monitored problems, describing the problem

8.1.4 ReportMode

Every Sensor has one or more ReportListeners. Each ReportListeners specifies, through a ReportMode object, which reports it would like to receive.

A ReportMode contains the following fields:

- String trigger: either ReportMode.THRESHOLD or ReportMode.CONTINUOUS
- String operation: ReportMode.CURRENT, ReportMode.STAT_AVG, ReportMode.STAT_ADD, ReportMode.STAT_MIN, ReportMode.STAT_MAX
- Value thresholdValue
- int reportPeriod

9. APPENDIX B: DEFINITION OF KEYCODES AND BUTTONCODES (*)

The following keycodes are to be used within the MobiThin system. The same values were used as defined on the Linux platform, under linux/input.h

Keycode	Symbolic name	Keycode	Symbolic name
0	MT_KEY_RESERVED	33	MT_KEY_F
1	MT_KEY_ESC	34	MT_KEY_G
2	MT_KEY_1	35	MT_KEY_H
3	MT_KEY_2	36	MT_KEY_J
4	MT_KEY_3	37	MT_KEY_K
5	MT_KEY_4	38	MT_KEY_L
6	MT_KEY_5	39	MT_KEY_SEMICOLON
7	MT_KEY_6	40	MT_KEY_APOSTROPHE
8	MT_KEY_7	41	MT_KEY_GRAVE
9	MT_KEY_8	42	MT_KEY_LEFTSHIFT
10	MT_KEY_9	43	MT_KEY_BACKSLASH
11	MT_KEY_0	44	MT_KEY_Z
12	MT_KEY_MINUS	45	MT_KEY_X
13	MT_KEY_EQUAL	46	MT_KEY_C
14	MT_KEY_BACKSPACE	47	MT_KEY_V
15	MT_KEY_TAB	48	MT_KEY_B
16	MT_KEY_Q	49	MT_KEY_N
17	MT_KEY_W	50	MT_KEY_M
18	MT_KEY_E	51	MT_KEY_COMMA
19	MT_KEY_R	52	MT_KEY_DOT
20	MT_KEY_T	53	MT_KEY_SLASH
21	MT_KEY_Y	54	MT_KEY_RIGHTSHIFT
22	MT_KEY_U	55	MT_KEY_KPASTERISK
23	MT_KEY_I	56	MT_KEY_LEFTALT
24	MT_KEY_O	57	MT_KEY_SPACE
25	MT_KEY_P	58	MT_KEY_CAPSLOCK
26	MT_KEY_LEFBRACE	59	MT_KEY_F1
27	MT_KEY_RIGHTBRACE	60	MT_KEY_F2
28	MT_KEY_ENTER	61	MT_KEY_F3
29	MT_KEY_LEFTCTRL	62	MT_KEY_F4
30	MT_KEY_A	63	MT_KEY_F5
31	MT_KEY_S	64	MT_KEY_F6

32	MT_KEY_D	65	MT_KEY_F7
66	MT_KEY_F8	105	MT_KEY_LEFT
67	MT_KEY_F9	106	MT_KEY_RIGHT
68	MT_KEY_F10	107	MT_KEY_END
69	MT_KEY_NUMLOCK	108	MT_KEY_DOWN
70	MT_KEY_SCROLLLOCK	109	MT_KEY_PAGEDOWN
71	MT_KEY_KP7	110	MT_KEY_INSERT
72	MT_KEY_KP8	111	MT_KEY_DELETE
73	MT_KEY_KP9	112	MT_KEY_MACRO
74	MT_KEY_KPMINUS	113	MT_KEY_MUTE
75	MT_KEY_KP4	114	MT_KEY_VOLUMEDOWN
76	MT_KEY_KP5	115	MT_KEY_VOLUMEUP
77	MT_KEY_KP6	116	MT_KEY_POWER
78	MT_KEY_KPPLUS	117	MT_KEY_KPEQUAL
79	MT_KEY_KP1	118	MT_KEY_KPPLUSMINUS
80	MT_KEY_KP2	119	MT_KEY_PAUSE
81	MT_KEY_KP3		
82	MT_KEY_KP0	121	MT_KEY_KPCOMMA
83	MT_KEY_KPDOT	122	MT_KEY_HANGEUL
		123	MT_KEY_HANJA
85	MT_KEY_ZENKAKUHANKAK U	124	MT_KEY_YEN
86	MT_KEY_102 ND	125	MT_KEY_LEFTMETA
87	MT_KEY_F11	126	MT_KEY_RIGHTMETA
88	MT_KEY_F12	127	MT_KEY_COMPOSE
89	MT_KEY_RO	128	MT_KEY_STOP
90	MT_KEY_KATAKANA	129	MT_KEY_AGAIN
91	MT_KEY_HIRAGANA	130	MT_KEY_PROPS
92	MT_KEY_HENKAN	131	MT_KEY_UNDO
93	MT_KEY_KATAKANAHIRAGA NA	132	MT_KEY_FRONT
94	MT_KEY_MUHENKAN	133	MT_KEY_COPY
95	MT_KEY_KPJPCOMMA	134	MT_KEY_OPEN
96	MT_KEY_KPENTER	135	MT_KEY_PASTE
97	MT_KEY_RIGHTCTRL	136	MT_KEY_FIND
98	MT_KEY_KPSLASH	137	MT_KEY_CUT
99	MT_KEY_SYSRQ	138	MT_KEY_HELP
100	MT_KEY_RIGHTALT	139	MT_KEY_MENU
101	MT_KEY_LINEFEED	140	MT_KEY_CALC
102	MT_KEY_HOME	141	MT_KEY_SETUP

103	MT_KEY_UP	142	MT_KEY_SLEEP
104	MT_KEY_PAGEUP	143	MT_KEY_WAKEUP
144	MT_KEY_FILE	184	MT_KEY_F14
145	MT_KEY_SENDFILE	185	MT_KEY_F15
146	MT_KEY_DELETEFILE	186	MT_KEY_F16
147	MT_KEY_XFER	187	MT_KEY_F17
148	MT_KEY_PROG1	188	MT_KEY_F18
149	MT_KEY_PROG2	189	MT_KEY_F19
150	MT_KEY_WWW	190	MT_KEY_F20
151	MT_KEY_MSDOS	191	MT_KEY_F21
152	MT_KEY_SCREENLOCK	192	MT_KEY_F22
153	MT_KEY_DIRECTION	193	MT_KEY_F23
154	MT_KEY_CYCLEWINDOWS	194	MT_KEY_F24
155	MT_KEY_MAIL		
156	MT_KEY_BOOKMARKS	200	MT_KEY_PLAYCD
157	MT_KEY_COMPUTER	201	MT_KEY_PAUSECD
158	MT_KEY_BACK	202	MT_KEY_PROG3
159	MT_KEY_FORWARD	203	MT_KEY_PROG4
160	MT_KEY_CLOSECD		
161	MT_KEY_OPENCD	205	MT_KEY_SUSPEND
162	MT_KEY_EJECTCLOSECD	206	MT_KEY_CLOSE
163	MT_KEY_NEXTSONG	207	MT_KEY_PLAY
164	MT_KEY_PLAYPAUSE	208	MT_KEY_FASTFORWARD
165	MT_KEY_PREVIOUSSONG	209	MT_KEY_BASSBOOST
166	MT_KEY_STOPCD	210	MT_KEY_PRINT
167	MT_KEY_RECORD	211	MT_KEY_HP
168	MT_KEY_REWIND	212	MT_KEY_CAMERA
169	MT_KEY_PHOME	213	MT_KEY_SOUND
170	MT_KEY_ISO	214	MT_KEY_QUESTION
171	MT_KEY_CONFIG	215	MT_KEY_EMAIL
172	MT_KEY_HOMEPAGE	216	MT_KEY_CHAT
173	MT_KEY_REFRESH	217	MT_KEY_SEARCH
174	MT_KEY_EXIT	218	MT_KEY_CONNECT
175	MT_KEY_MOVE	219	MT_KEY_FINANCE
176	MT_KEY_EDIT	220	MT_KEY_SPORT
177	MT_KEY_SCROLLUP	221	MT_KEY_SHOP
178	MT_KEY_SCROLLDOWN	222	MT_KEY_ALTERASE
179	MT_KEY_KPLEFTPAREN	223	MT_KEY_CANCEL
180	MT_KEY_KPRIGHTPAREN	224	MT_KEY_BRIGHTNESSDOWN

181	MT_KEY_NEW	225	MT_KEY_BRIGHTNESSUP
182	MT_KEY_REDO	226	MT_KEY_MEDIA
183	MT_KEY_F13	227	MT_KEY_SWITCHVIDEOMODE
228	MT_KEY_KBDILLUMTOGGLE	237	MT_KEY_BLUETOOTH
229	MT_KEY_KBDILLUMDOWN	238	MT_KEY_WLAN
230	MT_KEY_KBDILLUMUP	239	MT_KEY_UWB
231	MT_KEY_SEND	240	MT_KEY_UNKNOWN
232	MT_KEY_REPLY	241	MT_KEY_VIDEO_NEXT
233	MT_KEY_FORWARDMAIL	242	MT_KEY_VIDEO_PREV
234	MT_KEY_SAVE	243	MT_KEY_BRIGHTNESS_CYCLE
235	MT_KEY_DOCUMENTS	244	MT_KEY_BRIGHTNESS_ZERO
236	MT_KEY_BATTERY	245	MT_KEY_DISPLAY_OFF

The following buttoncodes are to be used within the MobiThin system. The same values were used as defined on the Linux platform, under linux/input.h

button code	Symbolic name	button code	Symbolic name
0x110	MT_BTN_LEFT	0x115	MT_BTN_FORWARD
0x111	MT_BTN_RIGHT	0x116	MT_BTN_BACK
0x112	MT_BTN_MIDDLE		

Other symbolic constants were defined as follows.

value	Symbolic name	value	Symbolic name
1	MTKEY_DOWN	0	MTBUTTON_DOWN
0	MTKEY_UP	0	MTRELATIVE
1	MTBUTTON_UP	1	MTABSOLUTE

- End of document -