HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

INSTITUT INFORMATIK

internal report 17-01

Camera Framework

Jens Fey, Sven Schlaghecke, Patrick Vöing, Darius Malysiak, Uwe Handmann

1 INHALT

2 GRUNDLAGEN

2.1 ÜBERSICHT

Das CameraFramework wurde entwickelt, um mittels Socket-Kommunikation [1] als Middleware zwischen verschiedenen Kamerainstanzen mit eigenen Kameratreibern und Clienten zu fungieren. Über diesen Kommunikationsweg ist es möglich Clienten nicht nur lokal, sondern auch über das Netzwerk mit Kameradaten zu versorgen.

Um neue Kameras mit dem Framework nutzen zu können, muss die Implementierung gewissen Regeln folgen, was durch ein vorgegebenes Basis-Interface (abstrakte Basis-Klasse in C++ [2]) fast vollständig sichergestellt ist. Neue Kameras werden zur Laufzeit über dynamische Bibliotheken geladen.

Parameter für Kameras sind über ein XML-File [3] einzustellen. Funktionen zur Übergabe von neuen Kameradaten sind implementiert und müssen durch den Entwickler der einzelnen Kamerainterfaces aufgerufen werden.

Die Zuordnung von Kameradaten zum passenden Nutzer übernimmt das Framework. Jeder Client erhält seinen eigenen konfigurierbaren Ringbuffer [4] um unabhängig von anderen Nutzern und Kameras zu sein.

Die Aufgaben des Frameworks sind auf verschiedene Module, wie in Abbildung 1 dargestellt, aufgeteilt.

2.2 GEGENÜBERSTELLUNG

Die grundlegende Idee ein einheitliches Interface für die Ansteuerungen von verschiedenen Kameratypen bereit zu stellen findet sich auch in anderen bereits existierenden Projekten wie beispielsweise GenICam [5] von EMVA oder der open-source Library libdc1394 [6]. Letzteres bietet lediglich die Möglichkeit Kameras, welche auf der 1394-based Digital Kamera Spezifikation basieren, anzusteuern. Im Gegenzug dafür bietet die Library Unterstützung für die meisten gängigen Desktop-Betriebssysteme, wohingegen die Anwendung unseres Frameworks auf Linux beschränkt ist. Wie auch in unserer Implementation lassen sich mehrere Kameras simultan betreiben, jedoch ist das libdc1394 Framework durch die Begrenzung auf den Firewire-Anschluss stark eingeschränkt. Der Vorteil unseres Frameworks liegt darin, dass eine Abstraktionsschicht den Anschluss jedes Kameratypen zulässt. Diese Methodik greift GenICam ebenfalls auf. Der Unterschied zum GenICam System liegt darin, dass wir die Daten der Kamera nicht in ein spezifisches Bildformat wandeln, sondern jedem Programmierer freistellen, wie er mit den Daten des Geräts umgehen möchte. Dies bietet die Möglichkeit Geräte in das System aufzunehmen, die andere Daten als nur reine Bilder zur Verfügung stellen. Beide Systeme unterscheiden sich zu unserem Framework noch zusätzlich darin, dass sie auf derselben Maschine laufen müssen, auf denen das Bild vorliegen soll. Unser System bietet die Flexibilität einer Netzwerkschnittstelle, über welche die Daten von einem entfernten Client abgegriffen werden können.

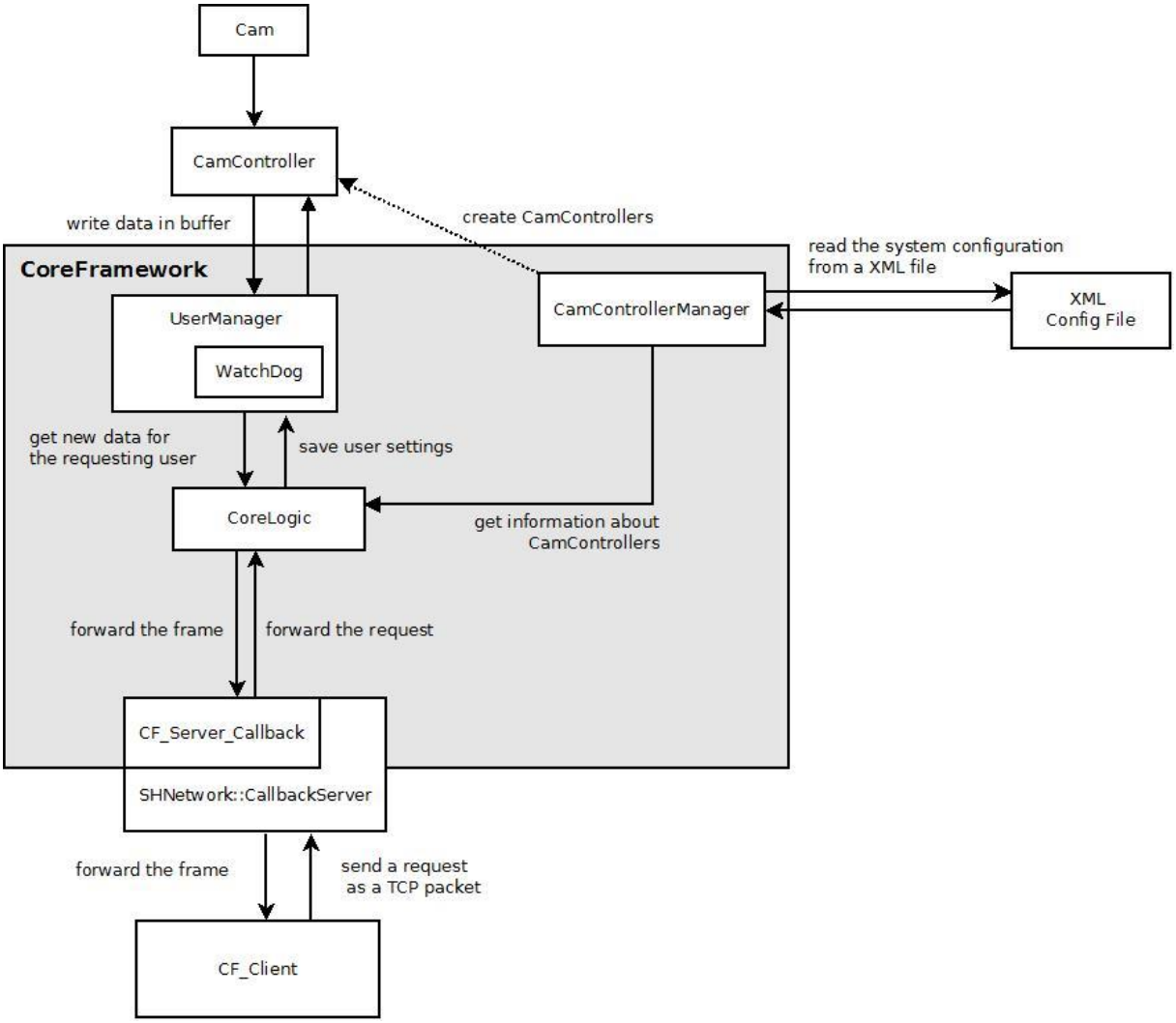

Abbildung 1: CameraFramwork Übersicht

2.3 MODULE

2.3.1 CAMCONTROLLER

CamController werden von der abstrakten Basisklasse „**CamController**“ abgeleitet. Das Framework nutzt alle **CamController**-Instanzen über Basisklassen-Zeiger um polymorphes Verhalten zu nutzen. Dies ermöglicht es dem Framework sämtliche **CamController** zu steuern, auch wenn diese neu erstellt werden und zur Compile-Zeit des Frameworks noch nicht bekannt sind. Die CamController sind dafür vorgesehen die Ansteuerung der Hardware zu übernehmen. Dabei sind Art der Hardware und Form der Daten irrelevant. Die fachgerechte Einbindung der Hardware obliegt dem Entwickler.

2.3.2 CAMCONTROLLERMANAGER

Der **CamControllerManager** ist für das Erstellen und Ausführen der **CamController** zuständig. Zu Programmstart wird die Konfiguration aus der XML-Dateil gelesen. Anschließend werden alle **CamController** mit den entsprechenden dynamischen Bibliotheken geladen, mit Parametern aus der XML-Datei initialisiert und daraufhin in einem extra Thread gestartet.

2.3.3 USERMANAGER

Der **UserManager** verwaltet die Nutzer des Frameworks. Dazu hat jeder Nutzer eine einzigartige ID, einen eigenen Ringbuffer sowie eine getroffene Kameraauswahl. Meldet sich ein Nutzer am

CameraFramework an, so hat er zudem die Möglichkeit den Ringbuffer zu konfigurieren. Erst nachdem ein Nutzer eine Kameraauswahl getroffen hat ist es für ihn möglich Daten dieser Kamera zu empfangen. Zusätzlich zur normalen Abmeldefunktion über die Nutzeranfrage „Logout“ gibt es eine WatchDog-Funktion im **UserManager**. Diese sorgt dafür, dass alle Nutzer, die in einem gegebenen Zeitintervall keine Aktivität gezeigt haben, aus dem **UserManager** entfernt werden.

2.3.4 CoreLogic

Die **CoreLogic** verarbeitet die Anfragen der Nutzer, welche über den TCP-Server an die **CoreLogic** weitergeleitet werden. Hier wird die Art der Anfrage bestimmt und die entsprechende Bearbeitung eingeleitet.

2.3.5 CF_Server_Callback / SHNetwork::CallbackServer

Diese beiden Klassen formen zusammen den TCP-Server. Der CallbackServer aus dem SHNetwork fungiert als TCP-Listener, welcher Anfragen an die Instanz des **CF_Server_Callbacks** jeweils in einem eigenen Thread weitergibt. Die **CF_Server_Callback**-Instanz leitet die Anfrage dann an die **CoreLogic** weiter.

2.3.6 CF_Client

Die **CF_Client**-Klasse ist ein Wrapper für die Kommunikation mit dem CameraFramework. Es stellt die vier Grundfunktionen *(getRdyCamList, selectCam, getNextFrame, logout)* des Nutzers als öffentliche Funktionen zur Verfügung und kapselt sämtlichen Overhead der TCP-Kommunikation. Weitere Details in Kapitel. Der **CF_Client** ist nicht teil des **CoreFrameworks** sondern dient als Beispiel-Benutzerinterface für die Kommunikation mit dem Server.

2.4 Thread Management

In Abbildung 2 ist das Thread Management grafisch dargestellt. Wie in der Abbildung zu erkennen, kann man die Threads in 5 verschiedenen Kategorien unterteilen.

Callbackserver

Der Callbackserver hat einen dauerhaft laufenden Thread(t_Listener), um eingehende TCP-Verbindungen entgegenzunehmen.

2.4.1 CamController

Jeder **CamController** läuft in einem eigenen Thread. Diese Threadobjekte werden im **CamControllerManager** neben den **CamControllern** gespeichert. Dies ist nützlich, um zu jeder Zeit Zugriff auf den laufenden Thread eines jeden **CamControllers** zu haben.

2.4.2 Client

Der Client läuft als eigenständige Anwendung und ist vom restlichen Framework getrennt. Die Anbindung zu Framework geschieht über ein TCP Interface, daher sind sämtliche Clients externe Threads und können sogar auf einem verteilten System operieren.

2.4.3 Framework

Für jede Anfrage an den Server wird ein neuer Worker-Thread gestartet, welcher die Bearbeitung übernimmt. Dabei werden verschiedene Module des Frameworks durchlaufen, jedoch werden keine

weiteren neuen Threads mehr erstellt.

2.4.4 WATCHDOG

Der WatchDog ist ein kontinuierlich laufender Thread im **UserManager**, welcher inaktive User entfernt.

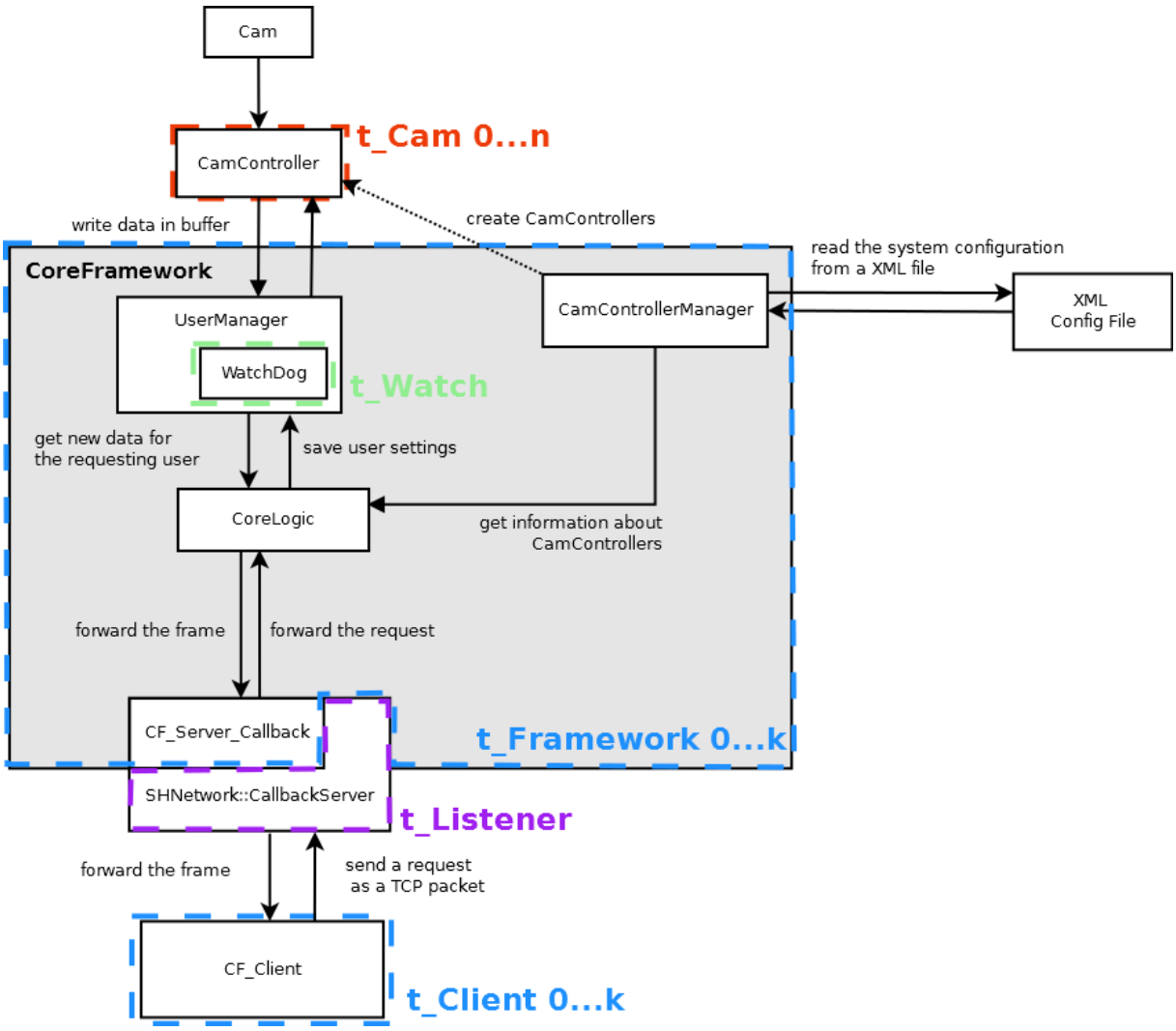

Abbildung 2: Thread-Management

2.5 XML-FILE

Das XML-File wird genutzt um dem **CameraFramework** Einstellungen für die **CamController** zu hinterlegen. In der Struktur ist es vereinfacht und entspricht nicht der offiziellen XML-Spezifikation. Das Wurzelelement ist `<CamControllers>` gefolgt von beliebig vielen Tags von `<CamController>`. Jedes dieser **CamController** Tags stellt einen **CamController** dar, der durch das Framework geladen werden soll. Jeder **CamController** muss die Attribute `<name>` und `<dll>` aufweisen.

<name>: Name des **CamControllers** im laufenden Framework. Über diesen Namen kann die Kamera später vom Nutzer ausgewählt werden. Jeder Name ist einmalig. Sollten zwei **CamController**-Elemente den identischen Namen haben, so wird nur der erste erstellt.

<dll>: Kombinierter string aus Pfad und Name der dynamischen Bibliothek, in welcher die Implementierung des **CamControllers** liegt.

Weitere Attribute sind optional und werden der zum **CamController**-Tag passenden **CamController**-Instanz in der *Init()*-Methode als std::vector von **CamereFeatures** übergeben. Die Verarbeitung der **CameraFeatures** steht dem Entwickler frei. Ein Beispiel XML-File ist in Abbildung 3 zu sehen:

Es werden zwei **CamController** angelegt. Während der zweite sich nur auf die nötigsten Elemente beschränkt, nutzt der **CamController** mit Namen „CCHDD“ drei weitere Parameter, welche jeweils als std::string-Objekte in das **CameraFeature** gelesen bzw. geschrieben werden.

```
<CamControllers>
    <CamController>
        <name>CCHDD</name>
        <dll>CamControllerHDD.so</dll>
        <dir>/home/projekt1team/img</dir>
        <timeInterval>2000</timeInterval>
        <waitForProspect>true</waitForProspect>
    </CamController>

    <CamController>
        <name>CCVB</name>
        <dll>libCamControllerVimbaBasic.so</dll>
    </CamController>
</CamControllers>
```

Abbildung 3 XML-Beispiel

2.6 SPEICHERVERWALTUNG

2.6.1 NEUE DATEN

Das Konzept sieht vor, dass die Daten von allen Controllern an den Usermanager weitergereicht werden, sobald ein neuer Datensatz von der Kamera eintrifft. Dazu stellt der UserManager die Methode

insertNewData(const std::string& name, unsigned char data, unsigned long long size);*

zur Verfügung, welche als Übergabeparameter den Namen des aufrufenden CamControllers, die Adresse der Daten sowie Länge der Daten in Bytes erwartet. Den Entwicklern von CamControllern steht die Methode *forwardNewData()* im CamController zur Verfügung, welche sich um den Aufruf im UserManager kümmert.

2.6.2 NUTZER-SPEZIFISCHER DATENSPEICHER

Der **UserManager** erzeugt für jeden angemeldeten Nutzer einen eigenen Ringbuffer(Abbildung 4). In diesen Speicher werden der Reihe nach die neu ankommenden Daten der Kamera gespeichert, für die sich der entsprechende Nutzer registriert hat. Jeder Datensatz erhält eine fortlaufende Sequenznummer, die es dem Nutzer ermöglicht festzustellen wenn Daten verworfen wurden. Dies kann zum Beispiel dadurch entstehen, dass ein **CamController** schneller Daten liefert als der Nutzer

Daten liest. Deswegen gibt es zudem die Möglichkeit, den Ringbuffer in verschiedenen Modi arbeiten zu lassen.

2.6.3 RINGBUFFER-MODI

2.6.3.1 Overwrite

In diesem Modus werden neue Daten in jedem Fall in den Ringbuffer geschrieben. Falls der Ringbuffer voll ist wird der älteste Datensatz überschrieben. Der Lese-Index wird auf den nun ältesten Datensatz gesetzt.

Sollten kontinuierlich schneller Daten generiert werden als der Nutzer diese abfragt, kann es dazu führen, dass beim Nutzer kontinuierlich Lücken im Datenstrom auftreten.

2.6.3.2 No_Overwrite

Im Modus **no_overwrite** werden keine neuen Daten in den Buffer geschrieben wenn dieser vollständig gefüllt ist. Werden neue Daten durch den CamController bereitgestellt, werden diese Verworfen und die Sequenznummer trotzdem entsprechend hochgezählt.

2.6.3.3 Clear_on_overtake

Dieser Modus ähnelt dem Modus **overwrite**, allerdings werden beim Einfügen eines neuen Datensatzes in einen vollständig gefüllten Buffer alle bisher gespeicherten Datensätze verworfen. Dies ermöglicht es dem Nutzer nur einen (großen) Sprung im Datenstrom zu erhalten um anschließend wieder genug freie Kapazitäten für einen fortlaufenden Datenstrom bereitzustellen.

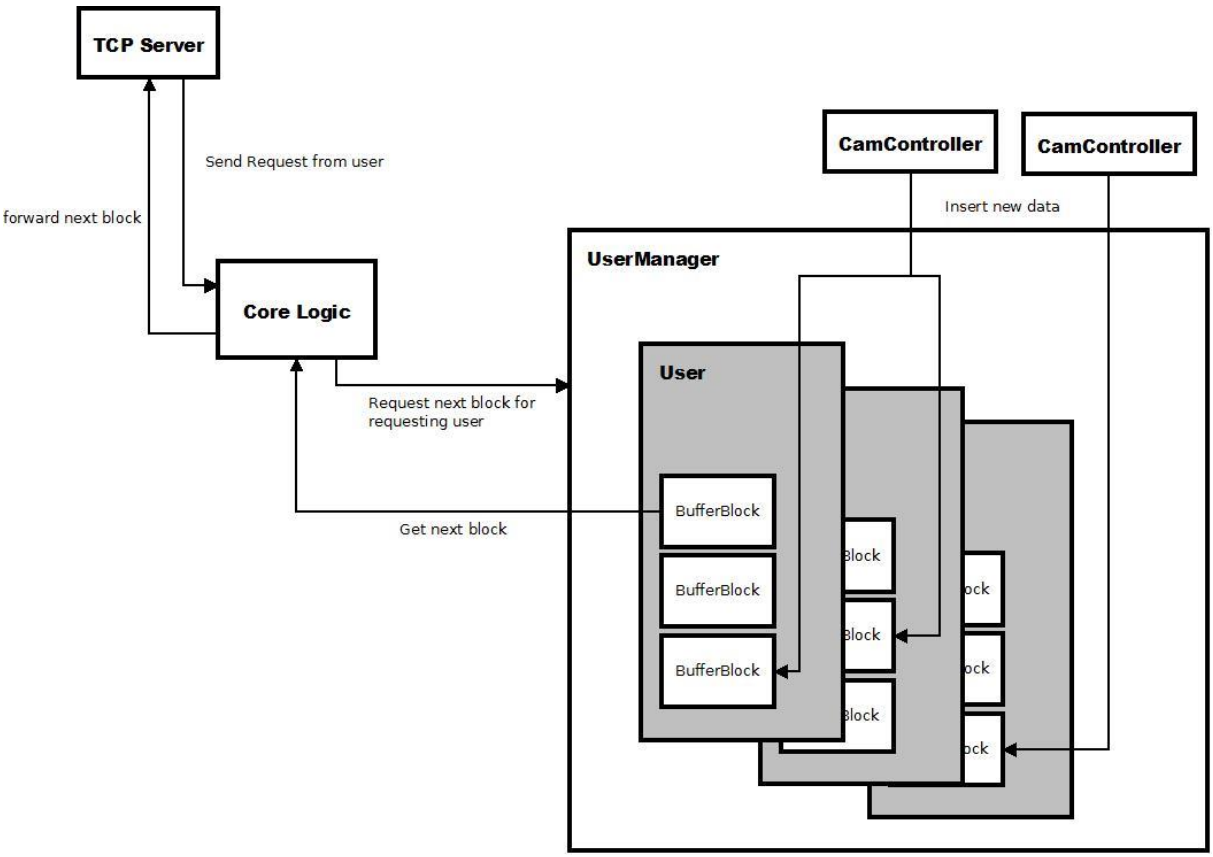

Abbildung 4: Speicherverwaltung

2.7 INITIALISIERUNGS-PROZESS

Bei Start des CameraFrameworkes werden einige initiale Prozesse getätigt. Dazu zählen:

- Instanziierungen der einzelnen Module
- Auslesen der XML-Datei
- Laden und Starten aller **CamController**
- Starten des **TCP-Servers**

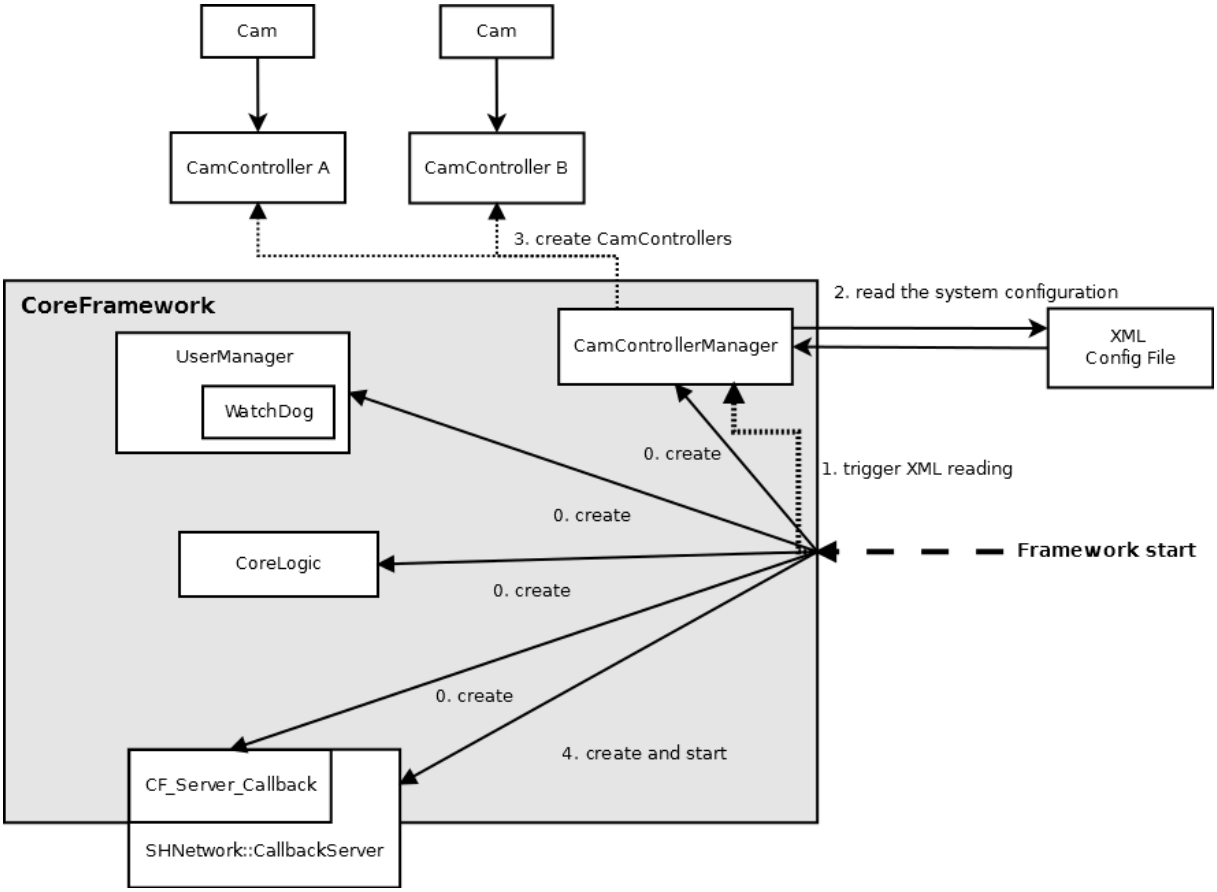

Abbildung 5: Framework-Start

Alle Module, bis auf den **SHNetwork::CallbackServer,** sind Member-Objekte der **CoreFramework**-Klasse und werden implizit erstellt. Das Auslesen der XML-Datei wird durch den **CamControllerManager** veranlasst. Anschließend werden alle **CamController** der Reihe nach mit den entsprechenden ausgelesenen Parametern instanziiert. Dazu wird die entsprechende dynamische Bibliothek geladen, der **CamController** erstellt und seine *init()*-Methode aufgerufen. Sollte der Constructor des **CamControllers** eine Exception werfen wird diese gefangen und der **CamController** wird nicht erstellt. Ebenso wird kein **CamController** im Framework aufgenommen, wenn seine *init()*-Methode eine Exception wirft oder sie false zurückliefert.

Die Möglichkeit Exceptions im Constructor zu werfen bietet den Komfort zur Laufzeit des **CamControllers** auf gewissen Gegebenheiten setzen zu können (Stichwort: class invariant). Nach der Erstellung aller **CamController** wird jeder durch den Aufruf seiner *run()*-Methode gestartet. Dabei wird ein erstellter Wrapper (**ExecutionFunctor**) um den Methodenaufruf genutzt, damit auch Exceptions aus der *run()*-Methode gefangen werden können und die Ausführung des Frameworks nicht terminiert wird.

Der **SHNetwork::CallbackServer** wird schlussendlich expliziert instanziiert und gestartet um fortan nutzeranfragen entgegen zunehmen.

2.8 KAMERAAUSWAHL-PROZESS

Um als Nutzer Daten der CamController aus dem CameraFramework zu beziehen muss das Framework zunächst intern einen Nutzer anlegen, den RingBuffer instanziieren sowie eine getroffene Kameraauswahl von diesem Nutzer speichern. Der übliche Ablauf ist in Abbildung 6 abgebildet.

2.8.1 VERFÜGBARE KAMERAS ABRUFEN

Der Nutzer hat die Möglichkeit vom CameraFramework eine Liste alle verfügbaren Kameras zu erfragen. Die Anfrage wird an das **CoreLogic**-Modul weitergeleitet, welches aus dem entsprechenden Modul - dem **CamControllerManager** - eine Liste aller Namen von verfügbaren Kameras bezieht. Diese Liste wird dem Nutzer zugesandt. Sollten keine Kameras verfügbar oder anderweitig ein Fehler aufgetreten sein, so wird der entsprechende **CF_ErrorCode** in der Server-Antwort gesetzt.

2.8.2 KAMERA AUSWÄHLEN

Eine Kamera kann über ihren Namen ausgewählt werden. Dazu schickt der Nutzer an den Server die Anfrage, eine Kamera auszuwählen. Einhergehend damit muss er den Namen der Kamera, die gewünschten Ringbuffer Kapazität sowie den gewünschten Ringbuffer Modus senden. Auch diese Anfrage geht in die **CoreLogic** und wird dort ausgewertet. Es wird geprüft, ob die gewünschte Kamera verfügbar ist und falls dem so ist, wird im System ein neuer Nutzer angelegt. Das SimpleHydra-Server-Interface ordnet jeder TCP-Verbindung eine eindeutige ID zu, über welche die Nutzer im CameraFramework identifiziert werden. Falls der Nutzer vorher schon eine andere Kamera ausgewählt hatte, wird diese Einstellung überschrieben. Wird der Auswahlprozess erfolgreich abgeschlossen, so wird der **CF_ErrorCode** „ok“ zurück geschickt und der **UserManager** beginnt Daten für den Nutzer in seinen Ringbuffer abzulegen. Diese können nun vom Nutzer abgegriffen werden.

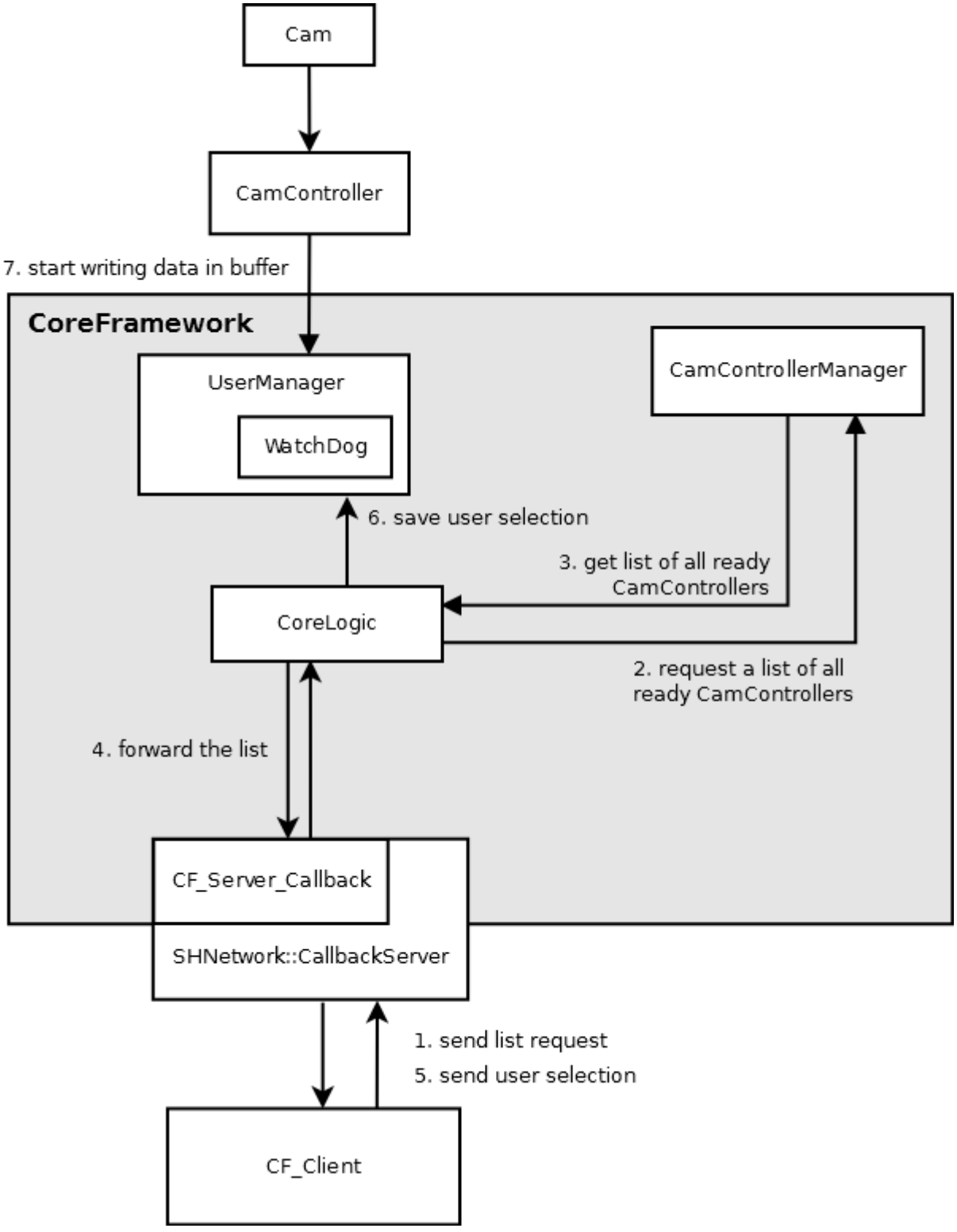

Abbildung 6: Kameraauswahl-Prozess

2.9 Daten Abrufen

Nachdem sich ein Nutzer am **CameraFramework** angemeldet hat (siehe Kapitel: Kameraauswahl-Prozess) kann er Daten beziehen. Der Ablauf ist in Abbildung 7 illustriert. Die Anfrage geht an den Server und wird im **CoreLogic**-Modul bearbeitet. Durch die eindeutige Nutzer-ID kann der **UserManager** den dazugehörigen Ringbuffer auswählen und eventuell verfügbare Daten bereitstellen. Die **CoreLogic** prüft dazu vor dem Zugriff auf den Ringbuffer, ob dieser leer ist. Sind

keine (neuen) Daten vorhanden wird ein entsprechender **CF_ErrorCode** an den Nutzer zurückgegeben. Ansonsten wird der Payload mit den Daten gefüllt.

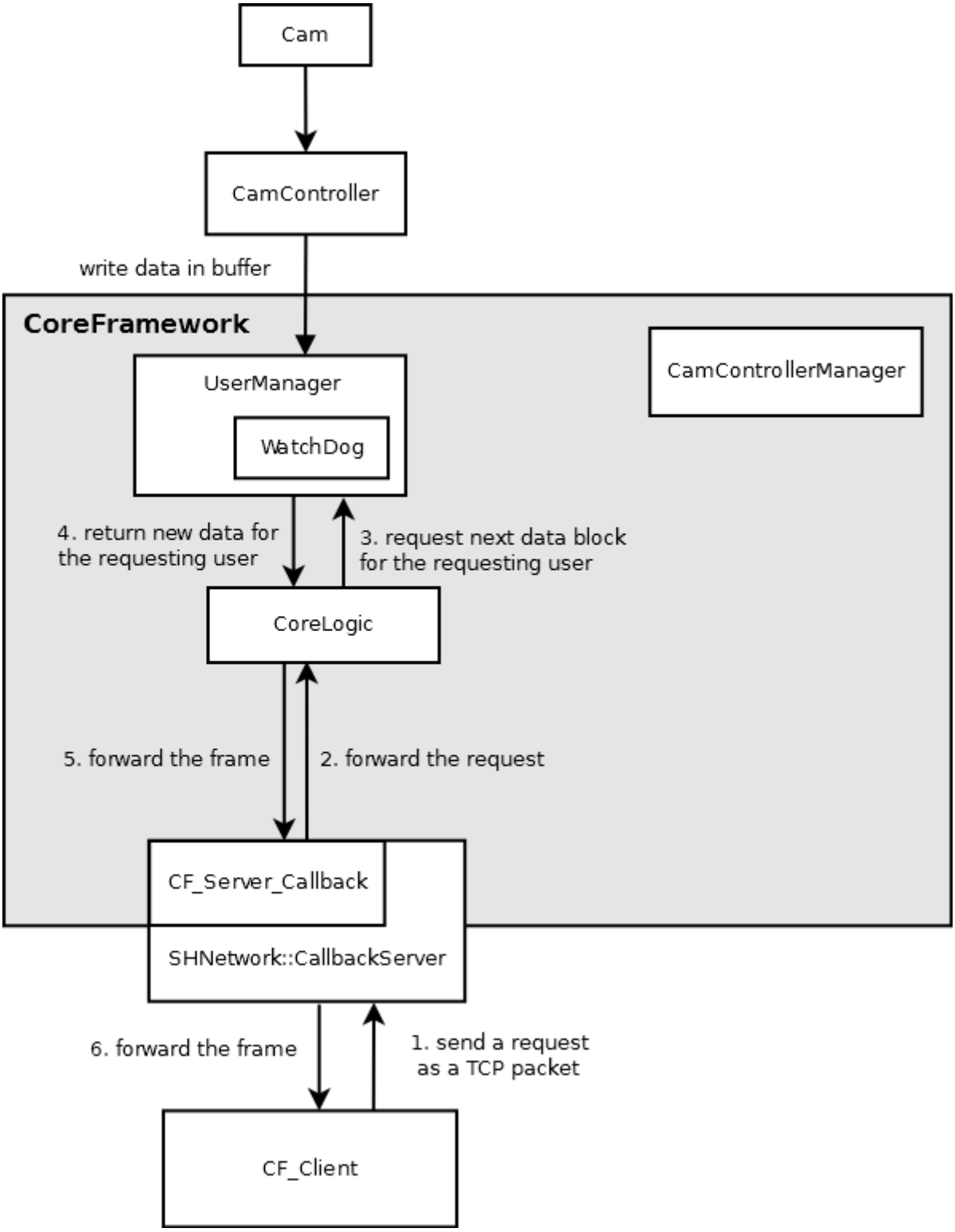

Abbildung 7: Daten-Abrufen

2.10 Kommunikations Struktur

Um die Kommunikation zwischen Client und Server zu strukturieren, sind im Header verschieden

Steuerelemente implementiert. Hierbei muss zwischen dem Weg vom Server und zurück unterschieden werden.

2.10.1 Client zu Server

2.10.1.1 Header

Der Kommunikationsweg beginnt immer beim Client. Jeder von ihm ausgehende Request hat folgende Struktur:

0	…	3	4	5	6	…	n
SH Enum			CF_Request Type	CF_Request ID	Payload		

SH Enum: Steuerzeichen des SimpleHydra Frameworks.

CF_Request Type: Art der Anfrage(„GetReadyCamList“, „SelectCam“, „GetNextFrame“, „Logout“).

CF_Request ID: Fortlaufende Nummer zur Zuordnung am Client

Die ersten 4 Bytes eines Paketes sind als Steuerzeichen von der Callback Struktur des SimpleHydra Frameworks belegt, welche für die TCP Verbindung verwendet wird. Danach folgt ein Byte „CF_Request Type“. Dieser beschreibt von welchem Type die Anfrage ist. „CF_Request ID“ ist eine fortlaufende Nummer und dient dazu eine erhaltende Antwort auf einen Request zuordnen zu können. Obwohl das Protokoll eine Ping-Pong-Kommunikation vorsieht kann es auf Grund von verschieden langen Bearbeitungszeiten oder Netzwerkverzögerungen zu einer Asynchronität kommen. Über die Request ID, welche im Server gespiegelt wird, kann der Client das Paket eindeutig zu einer gestellten Anfrage zuordnen. Der Rest des Paketes ist variabler Payload, der wiederum - abhängig vom Request-Type – verschiedene Strukturen haben kann.

2.10.1.2 Payload SelectCam

0	…	n	n+1	…	n+4	n+5
Cam Name\0			Buffer Size			Buffer Mode

Cam Name: String variabler Länge, terminiert durch den '\0' character.

Buffer Size: Die maximale Kapazität des Ringbuffers.

Buffer Mode: Der Modus des Ringbuffers.

Am Anfang des Payloads befindet sich der Name der Kamera als ASCII String in variabler Länge. Das Ende des Strings ist durch den '\0' character gekennzeichnet.

2.10.2 Server zu Client

0	…	5	6	7	…	n
Request Header			CF Error code	Payload		

Request Header: Gleicher Aufbau wie bei der Anfrage. Wird im Server gespiegelt.

CF Error code: Gibt Aufschluss über den Bearbeitungserfolg im Server

Die Antwort des Servers weist eine leicht erweiterte Struktur auf. Der Header des Request Paketes wird unverändert übernommen und um ein Byte „CF Error Code“ erweitert. Der Wert 0 bedeutet, dass kein Fehler aufgetreten ist und die Anfrage wie gewünscht ausgeführt wurde. Die restlichen Werte beschreiben verschiedene Fehlertypen.

Wie schon beim Request, so ist auch der Payload bei der Server-Response typenabhängig aufgebaut.

2.10.2.1 Payload getReadyCamList

0	…	n	n+1	…	m	m+1	…	l	…
Number of Elements\0			Element 1\0			Element 2\0			Element…

Number of Elements: Die Anzahl der Listeneinträge als string.

Element x: Der Name des **CamControllers** so wie er im Framework hinterlegt ist.

An erster Stelle steht die Anzahl der Listenelemente als Dezimalzahl in einem String. Hierdurch ist die Länge der Liste nicht begrenzt. Im Anschluss ist der Name jedes CamControllers aufgezählt und wird jeweils durch den '\0' character terminiert.

2.10.2.2 Payload getNextFrame

0	…	7	8	…	n
Frame ID			Data		

Frame ID: Fortlaufende Nummer kodiert als unsigned long long

Data: Die Daten die der **CamController** zur Verfügung stellt. (z.B. Bild Daten)

3 INSTALLATION

Das Framework wurde für Linux entwickelt und verwendet daher auch Linux spezifische Aufrufe. Ein Support für Windows ist nicht vorgesehen. Da die Daten über Sockets versendet werden, kann ein Client diese theoretisch auch auf einem Windows System entgegennehmen und verarbeiten. Der von uns bereitgestellte Client ist jedoch für Linux programmiert.

Bei der Programmierung des CameraFrameworks wurde auf Elemente des SimpleHydra-Frameworks zurückgegriffen. Daher ist es erforderlich, dieses auf dem System zu installieren. Die dafür erforderlichen Schritte sind in der SimpleHydra [7] Installationsbeschreibung erläutert.

Um das CameraFramework selbst zu kompilieren, müssen folgende Librarys dazu gelinkt werden:

- simpleHydraCore
- simpleHydraXML
- simpleHydraNetwork
- pthread
- dl

4 CF_CLIENT-KLASSE

Wie im Kapitel CF_Client beschrieben ist die CF_Client-Klasse ein Kommunikations-Wrapper für die Benutzung das CameraFrameworks. Es bietet ein Interface für die vier Grundfunktionen:

- getRdyCamList
- selectCam
- getNextFrame
- logOut

Diese Methoden nehmen den Nutzer den Aufwand ab, TCP-Pakete selbst du generieren und entsprechend auszulesen. Einmal eine Verbindung zum CameraFramework-Server erstellt, wird die Verbindung stets aufrechterhalten und es werden Fehler bei Paket-Zuordnung vermieden. Alle Funktionen sind Blocking-Calls mit einer Timeout-Dauer von 1000ms. Sollte bis dahin keine Antwort vom Server empfangen worden sein wird „**no_server_response“** als **CF_ErrorCode** zurückgegeben.

Es wird dabei auf Funktionalitäten der SimpleHydra-Network-Bibliothek zurückgegriffen.

4.1 VERBINDUNG HERSTELLEN

Der Constructor der CF_Client-Klasse nimmt die IP-Adresse als std::string-Objekt sowie den Port als unsigned integer entgegen. Daraufhin wird selbstständig eine Verbindung hergestellt. Ist dies geschehen, kann der Nutzer die vier Basis-Funktionen ausführen.

4.2 GETRDYCAMLIST

Funktions-Signatur: *CF_ErrorCode getRdyCamList(std::vector<std::string>& camList);*

Diese Methode setzt die Anfrage für das CameraFramework auf, sendet sie und wartet auf eine Antwort mit der Liste aller Kameranamen, die im CameraFramework verfügbar sind. Falls mindestens eine Kamera verfügbar ist, wird als CF_ErrorCode „ok“ zurückgeliefert und der übergebene vector mit allen Kameranamen gefüllt. Sollten Fehler auftreten, gibt der ErrorCode Aufschluss über die Ursache.

4.3 SELECTCAM

Funktions-Signatur: *CF_ErrorCode selectCam(const std::string& camName, unsigned int buffer_capacity, RingBuffer_Mode rb_mode);*

Durch den Aufruf der *selectCam()*-Methode lässt sich eine Kamera am CameraFramework auswählen. Dabei ist camName der Name der Kamera im Framework, buffer_capacity ist die Ringbuffer Kapazität und RingBuffer_Mode ist ein Enum, welches den Modus des Ringbuffers angibt. Alle Parameter werden korrekt an das CameraFramework übertragen und es wird auf eine Antwort des Frameworks gewartet. Sollte das Auswählen der Kamera gelingen, dann gibt die Funktion CF_ErrorCode „ok“ zurück.

4.4 GETNEXTFRAME

Funktions-Signatur: *CF_ErrorCode getNextFrame(BufferBlock& bb);*

Die Methode *getNextFrame()* schickt dem CameraFramework eine Anfrage für einen neuen Datensatz und wartet auf eine Antwort. Wenn ein neuer Datensatz empfangen wird, werden die Daten in den übergebenen BufferBlock kopiert und es wird CF_ErrorCode „ok“ zurückgeliefert. Sollte der **CF_ErrorCode** nicht „ok“ entsprechen, so wird der BufferBlock nicht verändert.

4.5 LOGOUT

Funktions-Signatur: *CF_ErrorCode logOut();*

Diese Funktion sendet dem Framework eine Anfrage mit dem Request-Type „LogOut“. Das Framework löscht den Nutzer intern mitsamt seinem Ringbuffer. Möchte der Nutzer wieder Daten empfangen, so muss er sich erst wieder mit *selectCam* anmelden.

5 Neuen CamController erstellen

5.1 Einleitung

Die Einbindung einer neuen Kamera in das Framework geschieht über dynamische Bibliotheken. Funktionen wie beispielsweise zum Abgriff von Bildern einer Hardwarekamera müssen in eine Bibliothek verpackt werden. Diese Bibliotheken werden im Rahmen unseres Frameworks als CamController betitelt. Die Aufgabe eines solchen CamControllers ist es, Daten von der entsprechenden Hardware abzugreifen und an das Framework weiterzuleiten. Hierzu muss der Code eine gewisse Grundstruktur aufweisen und wird von der „CamController“ Basisklasse vorgegeben. Von dieser Basisklasse muss abgeleitet und die entsprechenden Methoden implementiert werden.

5.2 Constructor

Dem Construktor der Basis-Klasse sind zwei Parameter zu übergeben

- **Name des CamControllers**
- **Eine Callback-Funktion zur Weiterleitung von Daten**

Der Name wird als String-Objekt übergeben.

Die Callback-Funktion wird als Functionwrapper übergeben, dessen Funktion folgende Signatur aufweist: `bool(const std::string& name, unsigned char* data, long size)`

Für das Framework wurde diesem Functionswrapper der Name **CF_CC_Forward_Function** gegeben.

```
CamControllerHDD::CamControllerHDD(const std::string& name,
CF_CC_Forward_Function f)
:CamController(name, f)
{
        //do setup if needed
}
```

5.3 Init

Funktions-Signatur: `bool Init(const std::vector<CameraFeature>& features);`

Diese Methode dient dafür initiale Schritte abzuarbeiten, die zum Starten der Hardware nötig sind. Gleichzeitig werden mit dem Funktionsaufruf alle zusätzlichen Parameter (CameraFeatures), die aus dem XML File gelesen wurden, an den Controller übergeben. Es ist Aufgabe des CamController-Programmierers diese Parameter zu interpretieren und zu verwenden. Falls alle Schritte fehlerfrei durchlaufen wurden, sollte die Init „true“ zurückgeben, andernfalls einen Fehler durch den Wert „false“ kenntlich machen. Durch eine fehlgeschlagene Initialisierung wird der CamController im CameraFramework nicht instanziiert und somit nicht gestartet.

```
bool CamControllerHDD::Init(const std::vector<CameraFeature>& features)
{
      //do init stuff
      //handle CameraFeatures
      return status;
}
```

Die Klasse CameraFeature besteht aus einem String-Pair Name und Value.

5.4 RUN

Funktions-Signatur: `void run();`

Hier soll das zyklische Abgreifen von der Hardware implementiert werden. Da jeder Controller in einem eigenen Thread läuft, kann dies in einer Dauerschleife stattfinden. Wie die Daten von der Hardware abgegriffen werden und in welchem Format sie vorliegen sind dabei wiederum Bereiche, die dem Programmierer zugeschrieben sind. Um die Daten an das Framework zu übermitteln stellt die CameraController Basisklasse die Methode „forwardNewData“ bereit. Diese erwartet als Übergabeparameter einen Zeiger auf die Daten im Speicher sowie deren Längenangabe als long.

```
void run()
{
    while(flag_stopExecution == false)
    {
                …          //grap some data
        bool is_anyone_interested = forwardNewData(data,size);
    }
}
```

Intern wird von der „forwardNewData“ Methode die „insertNewData“ Methode des UserManager aufgerufen. Dieser Mechanismus ist bereits von der Basis-Klasse implementiert und muss vom Programmierer nicht weiter beachtet werden.

Falls es im Framework keinen Interessenten gibt, der sich auf den Controller angemeldet hat, wird dies durch die Rückgabe der Wertes „false“ von der „`forwardNewData`“ Methode ersichtlich. Der Programmierer kann diese Information verwenden um Mechanismen zu implementieren, die das Weiterleiten von ungewollten Daten verhindern.

5.4.1 FLAG_STOPEXECUTION

Dieses Flag vom Type *std::atomic<bool>* wird vom Framework verwendet um dem CamCotroller mitzuteilen, dass er den Betrieb einstellen soll, da er bald zerstört wird. Das Framework ruft dazu die interne Methode „stopExecution“ des Controllers auf, in der das Flag auf „true“ gesetzt wird. Der Programmierer sollte in der *run()*-Methode das Flag in regelmäßigen Abständen abfragen und dafür sorgen, dass die Methode dementsprechend beendet wird.

5.5 SPEZIFIKATION ZU DYNAMISCHER BIBLIOTHEK

Um aus dem Source Code eine Bibliothek erstellen zu können und diese dem Framework zugänglich zu machen, müssen einige Einstellungen getroffen werden. Zunächst muss in den Compiler-Einstellungen das Flag „fPIC“ gesetzt werden.

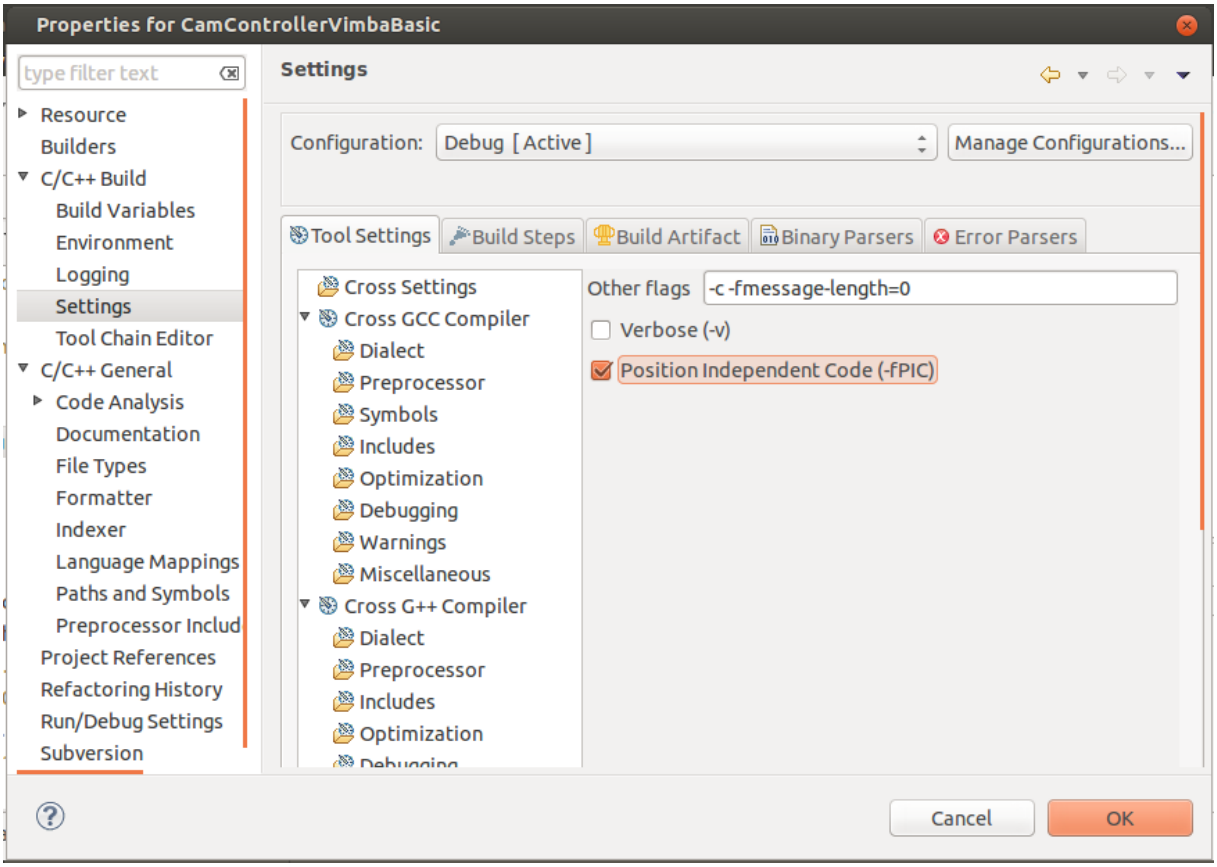

Abbildung 8: fPIC - Compilerflag

Um den CamController im Framework erstellen zu können, ist es von Nöten eine einheitliche Create Funktion bereit zu stellen. Jede Controller-Implementierung muss diese Funktion mit Namen „*create*“ implementieren. In dieser wird der CamController erstellt und ein Basisklassen-Zeiger auf das angelegte Objekt zurückgegeben. Dieser Schritt ist nötig, da das Framework nicht selbst den Construktor aufrufen kann, da der genaue Bezeichner zur Compile-Zeit nicht bekannt ist.

```
extern "C" CamController* create(const std::string& name,
CF_CC_Forward_Function f)
{
    return new CamControllerVimbaBasic(name,f);
}
```

Der Zusatz „`extern "C"`“ ist notwendig um das „name mangling“ des c++ Codes zu vermeiden und die Funktion über den Namen „create“ aus der dynamischen Bibliothek laden zu können.

Damit aus dem Code auch eine dynamische Bibliothek generiert wird muss dem Compiler dies auch mittgeteilt werden. Dazu muss der Artifact Type in den „Build Settings“ auf “Shared Library“ gestellt und als „Artifact extension“ die Endung „so“ gewählt werden.

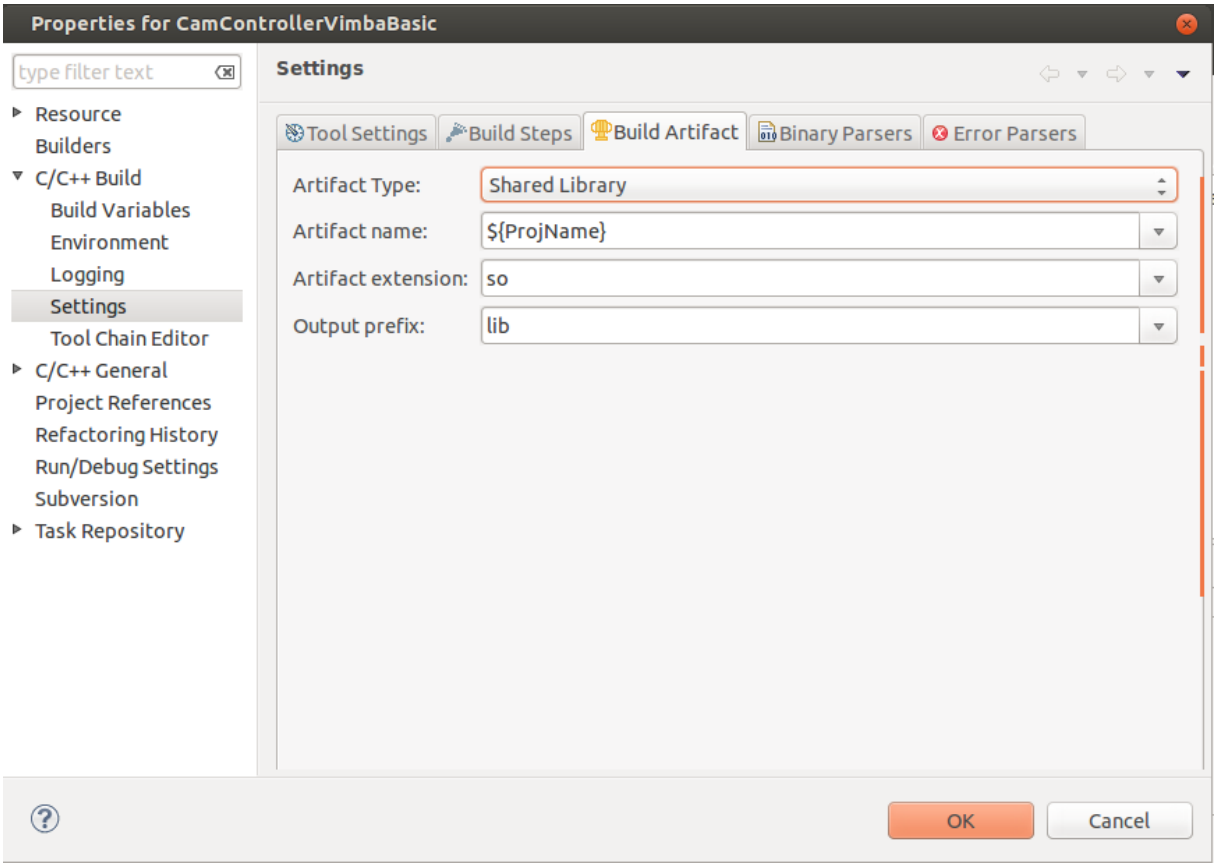

Abbildung 9: shared Library Einstellung

6 Ausblick und Erweiterungen

Das Konzept der Kameraeinbindung durch dynamische Bibliotheken ermöglicht es verschiedenste Kameratypen ohne Änderungen am Framework vornehmen zu müssen. Neue Kameras können durch das Basis-Interface des CamControllers schnell und komfortable eingebunden werden. Durch die vorgenommene Abstraktion lassen sich Kameras mit unterschiedlichsten Datenformaten und sogar gänzlich andere Hardware verwenden. Sogar der gleichzeitige Betrieb auf verteilten Systemen mit variablen Anzahl von Nutzern und mehreren Kameras ist möglich.

Der Aufbau des CameraFramework selbst ist durch seine Struktur leicht überschaubar und verteilt die Komplexität auf die einzelnen Module. Dadurch sind diese gut wart- und erweiterbar. Änderungen können sehr lokal bearbeitet werden.

Da Clients und Server über Sockets kommunizieren kann man das Framework sogar Plattform unabhängig nutzen, auch wenn das CameraFramework selbst auf Linux-Basis implementiert wurde.

Um das Framework zu verbessern bestehen noch diverse Möglichkeiten.

6.1 Denkbare Erweiterungen

Die wohl umfassendste Erweiterung wäre eine verbesserte bzw. erweiterte Schnittstelle zwischen Client und Server, mit der es ermöglicht wird, das Framework remote zu steuern. Im jetzigen Stand beschränkt sich die Kommunikation lediglich auf den Austausch von Kameradaten.

Zusätzlich sind Funktionen wünschenswert, die es Nutzern ermöglicht, CamController im laufenden Betrieb zu modifizieren um Einstellungen wie zum Beispiel Auflösung oder Bildwiederholungsrate zu setzen.

Daraus resultiert, dass man eine umfassendere Nutzerverwaltung implementieren muss. Clients müssen über Änderungen von Kameraeinstellungen informiert werden. Dies führt zudem zu einem erhöhten Workload für den Programmierer beim Erstellen neuer CamController, da diese eine deutlich erhöhte Komplexität aufweisen würden.

Um das Debugging im Fehlerfall zu verbessern kann eine Logging-Funktion in das Framework eingebunden werden. Diese sollte den regulären Betrieb nicht stören und trotzdem helfen, die Abläufe im Framework nachzuvollziehen.

Durch steigende Framework-Komplexität wäre es sinnvoll, Parameter des Frameworks ebenfalls über eine Konfigurationsdatei im XML-Format zu setzen, welche zum jetzigen Stand noch im Sourcecode zu ändern sind (zum Beispiel: WatchDog-Intervall, Anzahl-Worker-Threads, TCP-Port…).

7 LITERATURVERZEICHNIS

[1] A. Willemer, UNIX Das umfassende Handbuch, Galileo Computing.

[2] „C++ ISO,“ [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=64029.

[3] W3C, „XML,“ [Online]. Available: https://www.w3.org/TR/2006/REC-xml-20060816/.

[4] „Circular Buffer,“ [Online]. Available: http://c2.com/cgi/wiki?CircularBuffer.

[5] emva. [Online]. Available: http://www.emva.org/standards-technology/genicam/.

[6] libdc1394. [Online]. Available: http://damien.douxchamps.net/ieee1394/libdc1394/.

[7] D. Malysiak, „SimpleHydra Internal Report“.

Impressum

internal report 17-01

ISSN: 2197-6953

1. Auflage, 31.01.2017

Anschrift

Institut Informatik

Hochschule Ruhr West

Lützowstraße 5

46236 Bottrop