Team: 01 Francis Opoku, Fabian Reiber

Aufgabenaufteilung:

1. Klassendiagramm, Sequenzdiagramm (vsp4_empfangen), sowie zugehörige Doku.
2. Klassendiagramm, Sequenzdiagramm (vsp4_start, vsp4_senden), sowie zugehörige Doku. Sender-, Messagegen- und Starter-Modul

Quellenangaben: Vorlesungsfolien und Aufgabenstellung

Begründung für Codeübernahme: keine

Bearbeitungszeitraum:
Entwurf:
03.06.15: Fabian ca. 3 Std. für ersten Entwurf und erstes Klassendiagramm.
06.06.15: Fabian ca. 3 Std. Entwurf weiter ausgearbeitet und erstes Sequenzdiagramm erzeugt.
09.06.15: Francis ca. 3 Std. Entwurf: Anforderungsanalyse.
10.06.15: Francis und Fabian ca. 5 Std. gemeinsam Entwurf ausgearbeitet.
11.06.15: Fabian ca. 2 Std. weitere Bearbeitung des Entwurfs.
11.06.15: Francis 3 Std. Sequenzdiagram vsp4_empfangen.

Implementierung:
11.06.15: Fabian ca. 1 Std. Initialisierungsphase
12.06.15: Fabian ca. 3 Std. Sender und Messagegen
13.06.15: Francis ca. 3 Std. Receiver, SlotReservation und TimeSync
15.06.15: Fabian ca. 2 Std. Testen und Bug fixing
16.06.15: Francis und Fabian ca. 3 Std. Testen und Bug fixing
16.06.15: Fabian ca. 3 Std. Dokupflege
17.06.15: Francis und Fabian ca. 6 Std. Bug fixing.

Aktueller Stand: Entwurf fertig. Implementierung soweit auch, allerdings treten noch einige Fehler auf und das Senden der Pakete zur richtigen Zeit stimmt nicht ganz.

Änderungen im Entwurf:

1) Einige Methoden wurden in den jeweiligen Modulen nachträglich eingefügt, welche in den Diagrammen und der Dokumentation nach gepflegt wurden.
2) Einige Nachrichtenformate sind ebenfalls hinzugekommen oder wurden geändert.

Entwurf: siehe folgende Seiten

1. Einleitung

1.1. Ziel

Ziel ist es ein spezielles Zeitmultiplexverfahren (STDMA), welches für die Datenübertragung über Funk genutzt wird, zu implementieren, indem ein IP-Multicast genutzt wird, anstelle eines Funkkanals. Mit einem solchen Verfahren ist es möglich, dass verschiedene Sender bzw. Stationen innerhalb eines Frames ihre Nutzdaten versenden können, die über einen einzigen Kanal gehen. Dabei ist ein Frame in mehrere Zeitschlitze (Slots) unterteilt. In einem Slot sendet immer genau eine Station. Die Vergabe der Slots soll dezentral geschehen, was die STDMA-Verfahren so speziell machen. Unabhängig vom Senden der Nutzdaten, sind die Stationen ständig empfangsbereit um Kollisionen erkennen zu können und um zu ermitteln in welchen freien Slot sie im nächsten Frame senden dürfen. Damit alle Stationen zeitlich synchronisiert sind, muss ebenfalls ein Synchronisations-Algorithmus implementiert werden (siehe **2.3 Verwendeter Synchronisations-Algorithmus**).

1.2. Rahmenbedingungen

a) Die Programmiersprache ist frei zu wählen, sofern diese im Labor installiert ist.
b) Es ist die Datenquelle (*DataSource*) zu verwenden.
c) Als Datensenke kann ein log-File oder die Konsolenausgabe genutzt werden.

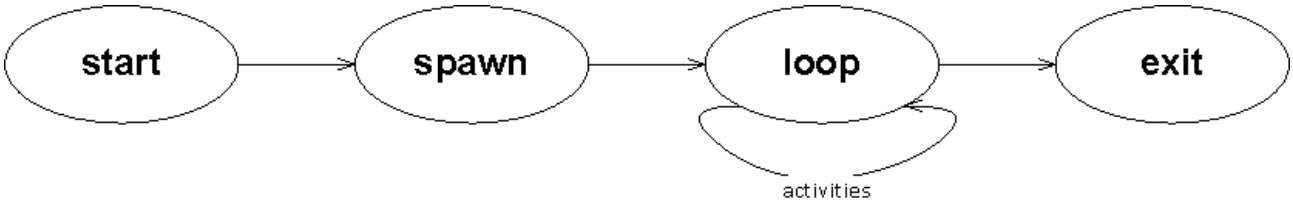
1.3. Konventionen

Der Entwurf lehnt sich an die Programmiersprache Erlang an, sodass sich Erlang-Strukturen wie z.B. Listen, Tupel, Atome, Variablen etc. in diesem Dokument wieder finden lassen. Unter Punkt **3 Anforderungen** sind die in der Aufgabenstellung angegebenen Anforderungen nochmals strukturiert zusammengefasst. In den Beschreibungen der einzelnen Module werden vereinzelnd Verweise auf die Anforderungen im Entwurf zu finden sein.

2. Rahmenbedingungen

2.1. allg. Ablauf (Prozesse)

Eine Bedingung, um den Entwurf genauer verstehen zu können, ist, dass man sich den allgemeinen Ablauf der Prozess-Erstellung genauer ansehen muss. Dieser wird im folgenden Diagramm angegeben und kurz erläutert:



Durch eine zentrale Instanz, dem **Starter-Modul** wird ein **Receiver-** und **Sender-Prozess** gestartet (spawn). Der **Receiver-Prozess** startet wiederum den **TimeSync-** und **SlotReservation-Prozess**. Durch den **Sender-Prozess** wird der **MessageGen-Prozess** gestartet.

2.2. verwendete Bibliotheken/externe Module/bereitgestellte Tools

Es wird das Modul „werkzeug.erl“ vom Professor Klauck verwendet, da dieses nützliche Methoden bereitstellt. Weiterhin wird eine Datenquelle (*DataSource*) verwendet, damit der Sender immer die aktuellen Nutzdaten zur Verfügung hat die in das Datenpaket eingebunden werden. Um die Stationen kontrolliert starten zu können, wird ein weiteres Start-Skritp (*startStations.sh*) verwendet. Zur Überprüfung des Netzwerk-Datenverkehrs wurde ebenfalls ein Sniffer (*STDMASniffer*) zur Verfügung gestellt.

2.3. Verwendeter Synchronisations-Algorithmus

Für die Synchronisation der Uhren der A-Stationsklassen wird der **Berkeley (Unix) Algorithmus** verwendet. Im folgenden ein kurzer Ablauf aus den bereitgestellten Unterlagen:

Ein Rechner ist ein **Koordinator** und „jeder ist Koordinator“ und die Uhrzeit wird aus den Nachrichten extrahiert. Algorithmus:

1. Zeit-Server (= Koordinator) fragt periodisch alle Rechner nach ihrer Uhrzeit
2. Aus den erhaltenden Antworten werden die lokalen Zeiten durch Schätzung der Nachrichtenlaufzeiten ermittelt. Antworten, die zu lange auf sich warten lassen, werden ignoriert.
3. Aus den geschätzten lokalen Zeit wird das arithmetische Mittel gebildet.
4. Die jeweiligen Abweichungen vom Mittel werden als neuer aktueller Uhrenwert den Rechnern zurückgesendet.

2.4. STDMA-Verfahren

a) Jede Station sieht alle Slots aus dem letzten Frame (k-1).
b) Aus diesem Frame ermittelt sie die freien Slots, indem die Station das Byte 25 (in dem die Nummer des Slots welches im nächsten Frame (k) belegt sein wird steht) aller Datenpakete auswertet.
c) Im aktuellen Frame (k), schreibt sie in die zu versendende Nachricht den freien Slot, in dem sie im nächsten Frame (k+1) senden wird.

3. Anforderungen

3.1. Beschreibung

Anforderungs-Nr	Beschreibung
1.0	Alle Stationen sind ständig empfangsbereit.
2.0	Jede Station soll genau einmal pro Frame senden.
2.1	Die Station soll in der Mitte ihres gewählten Slots senden.
3.0	Die Slot-Vergabe ist dezentral.
3.1	Jede Station sucht daher selbstständig einen eigenen freien Slot zum Senden im nächsten Frame.
3.2	Mögliche freie Slots sollen zu jedem Zeitpunkt zur Verfügung gestellt werden.
4.0	Die Stationen werden in die Klassen **A** und **B** eingeteilt.
4.1	Die Uhr einer Station der Klasse **A** ist hinreichend genau.
4.2	Die Uhren dieser Stationen sollen sich im Rahmen der Möglichkeiten untereinander synchronisieren.
4.3	Eine anfängliche Abweichung von der UTC soll beim Start einstellbar sein.
4.4	Die Uhr einer Station der Klasse **B** ist **nicht** hinreichend genau.
4.5	Die Uhren dieser Stationen müssen sich mit den Uhren der Stationen der Klasse **A** synchronisieren.
5.0	Kollisionen, die a priori (grundsätzlich) zu erkennen sind, sollen vermieden werden.
5.1	Tritt eine Kollision auf, gelten die beteiligten Pakete als nicht empfangen bzw. nicht auswertbar.
5.2	Der kollisionsfreie Betrieb muss erreicht und darf bei unveränderter Stationsanzahl nicht wieder verlassen werden.

5.3	Sofern die Anzahl der Stationen kleiner als die Anzahl der Slots im Frame ist, darf eine neu hinzukommende Station keinerlei Kollisionen verursachen.
5.4	Bei kleineren Änderungsraten der Referenzzeit (bis ¼ Slotlänge pro Frame) darf eine neu hinzukommende Station keinerlei Kollisionen verursachen.
6.0	Erkannte Kollisionen sollen von den Stationen protokolliert werden.
6.1	Hat Station selbst in einen solchen Slot gesendet haben wird dies ebenfalls protokolliert.
6.2	Anzahl der Frames in denen gesendet wurde soll protokolliert werden.
6.3	Anzahl der Frames in denen **nicht** gesendet wurde soll protokolliert werden.
7.0	Das bereitgestellte Startskript wird zum Starten der Stationen verwendet.
7.1	Multicas-Adresse, Empfangsport und Netzwerkinterface müssen per Parameter beim Programmstart einstellbar sein.
8.0	Bereitstellung der Nutzdaten wird durch eine Datenquelle übernommen die zur Verfügung steht.
8.1	Die Station nimmt immer die aktuell verfügbaren Nutzdaten.
9.0	Ein Frame dauert 1 Sekunde.
9.1	Ein Frame besteht aus 25 Slots. Somit ist jeder Slot genau 40 ms lang.
9.2	Die Frames sind in Sekunden seit dem 1.1.1970, 00:00 Uhr nummeriert.
10	TTL der Multicast-Pakete ist auf 1 zu setzen, um unnötige Netzlasten und -störungen zu vermeiden.

3.2. Format und Semantik der Nachrichtenpakete

Byte	**Beschreibung**
0	Stationsklasse (**A** oder **B**)
1-24	Nutzdaten (Byte 1-10: Name der sendenen Station
25	Nummer des Slots in dem die Station im nächsten Frame senden wird.
26-33	Zeitpunkt zu dem dieses Paket gesendet wurde. Einheit: Millisekunden seit dem 1.1.1970 als 8-Byte Integer, Big Endian
Summe: 34 Bytes	

3.3. Multicast-Adressen und Ports

Nachfolgende Werte sollen aus Interferenzgründen verwendet werden:

Adresse:	255.10.1.2
Empfangsport:	15000 + <Teamnummer> := 15001

4. Modularisierung

4.1. Beschreibung der Module

Im folgenden werden die einzelnen Module kurz beschrieben, sowie die angefertigten Diagramme kurz erläutert.

a) Klassendiagramm

Eine Station wird in folgende 6 Module eingeteilt:

1. Starter
2. Receiver

3. Sender
4. TimeSync
5. MessageGen
6. SlotReservation

Ihre Aufgaben werden im folgenden Punkt **b)** kurz erläutert.

b) Verantwortlichkeiten der Module:

1. **startStations:** Durch dieses bereitgestellte Skript von Herrn Schulz, wird die Station sowie die Datenquelle gestartet und der Strom von Datenpaketen in einem festen Zeitintervall aus dieser Quelle an die Station geleitet.
2. **Starter:** Mit diesem Starter werden die einzelnen Module gestartet.
3. **Receiver:** Der Receiver empfängt die über Multicast gesendeten Nachrichten [den Datenstrom über den Multicast] und entscheidet, ob es sich um eine Kollision handelt. Bei Kollision findet keine Auswertung statt. Bei keiner Kollision wird die Zeit und die Stationsklasse aus dem Datenpaket an das Modul *TimeSync* weitergeleitet und der Slot an das Modul *SlotReservation*.
4. **Sender:** Der *Sender* wartet nun auf den Empfang einer zu versendenden Nachricht vom *MessageGen*-Modul. Hat er diese empfangen muss nun noch überprüft werden, ob auf diesem Slot bis zu diesem Zeitpunkt bereits gesendet wurde. Daher wird eine Nachricht an das *Empfänger*-Modul gesendet die die entsprechende Information übermittelt. Wurde keine Kollision erkannt wird der aktuelle Timestamp (eigene Uhr) geholt und der Nachricht angehängt.
5. **TimeSync:** Über einen bestimmten Zeitraum hinweg (z.B. 1 Sekunde) werden die Zeiten aus den einzelnen Datenpaketen der A-Stationsklassen gesammelt. Darüber wird dann das arithmetische Mittel gebildet. Mit dieser Information und der eigenen Uhr (Systemzeit) kann nun die eigene Abweichung ermittelt werden, welche entweder auf die eigene Uhr addiert oder von der eigenen Uhr subtrahiert wird. Handelt es sich um eine B-Stationsklasse synchronisieren diese sich über die A-Stationsklassen.
6. **SlotReservation:** Zu jedem Zeitpunkt wird ein zufällig gewählter freier Slot im nächsten Frame bereitgestellt. Vom *Sender* kann hier erfragt werden, ob eine Kollision auf einen jeweiligen Slot möglich ist.
7. **MessageGen:** Bis auf den Timestamp (wann die Nachricht tatsächlich verschickt wurde) wird hier die Nachricht für das Senden vorbereitet. **Die Nutzdaten aus der Datenquelle werden hier ausgelesen und weiter verarbeitet.** Dafür werden die Nutzdaten in einem parallel laufenden Prozess empfangen, der jedoch blockiert solange keine neuen Nutzdaten von der Datenquelle eingetroffen sind. Sobald neue Nutzdaten eingetroffen sind, sendet dieser Prozess die aktuellen Nutzdaten an einen Puffer-Prozess. Dieser Puffer-Prozess kann von der MessageGen angesprochen werden, wenn diese neue Nutzdaten anfordert um eine Nachricht zu präparieren. Ebenso muss, auf Grundlage des ermittelten Slots die Sendezeit ermittelt werden damit der *Sender* rechtzeitig die Nachricht mitgeteilt werden kann. Die aktuelle Uhrzeit wird über das TimeSync-Modul ermittelt mit der u.a. die Sendezeit ermittelt werden kann. Ist die ermittelte Sendezeit schon vorbei, muss dem *Sender* dies ebenfalls mitgeteilt werden.

c) Nachrichtenformate

Damit die einzelnen Prozesse innerhalb einer Station sich verständigen können, müssen vorher Nachrichtenformate definiert werden. Im folgenden je eine Tabelle für Befehle die von den Modulen jeweils versendet und empfangen werden müssen.

Starter:

Befehl (Senden)	Beschreibungen
kill	kill-Befehl an Sender und Receiver schicken, damit diese sich, und alle anderen selbst erstellten „Hilfs-Prozesse“, terminieren.
Befehl (Empfangen)	**Beschreibungen**
kill	Erhält kill – Kommando über die Konsole und initiiert somit die Terminierung des Receivers und Senders indem er ein kill an die jeweiligen Prozesse schickt.

Receiver:

Befehl (Senden)	Beschreibungen
{times, StationClass, TimeInSlot}	Versendet den, im Datenpaket, enthaltenen die Stationsklasse (Byte 0) und den Timestamp (Byte 26-33) an den **TimeSync-Prozess.**
{getTime, self()}	Sendet eine Anfrage für den Timestamp an den **TimeSync-Prozess**.
{slot, NextSlot}	Sendet die Nummer des Slots, aus dem Datenpaket (Byte 25), in dem im nächsten Frame gesendet werden kann an den **SlotReservation-Prozess**.
{getPID, self()}	Benötigt die MessageGen-PID und fordert diese beim **Sender-Prozess** an.
kill	Sendet ein kill an den **SlotReservation-** und **TimeSync-Prozess**.
Befehl (Empfangen)	**Beschreibungen**
{pid, MessageGenPID}	Bekommt die MessageGen-PID vom **Sender-Prozess**.
{currentTime, TimeStamp}	Erhält den aktuellen Timestamp des **TimeSync-Prozesses**.
kill	Damit alle „Hilfs-Prozesse“ und sich selber zu beenden. Wird von Starter initiiert.

Sender:

Befehl (Senden)	Beschreibungen
{pid, MessageGenPID}	Sendet dem **Receiver-Prozess** die MessageGenPID.
{helloSlot, SlotReservationPID}	Sendet die zuvor empfangene PID des **SlotReservation-Prozesses** an den **MessageGen-Prozess**.
{helloTime, TimeSyncPID}	Leitet dem **MessageGen-Prozess** die PID des **TimeSync-Prozesses** weiter.
{getTime, self()}	Möchte vom **TimeSync-Prozess** die aktuelle Uhrzeit wissen die er dem Datenpaket anhängen

	möchte.
{collision, Slot}	Möchte von dem **SlotReservation-Prozess** wissen, ob eine Kollision für den angegebenen Slot stattfinden könnte.
kill	Sendet ein kill an den **MessageGen-Prozess**.
Befehl (Empfangen)	Beschreibungen
{getPID, ReceiverPID}	Erhält die Anfrage des **Receiver-Prozesses**, dass er ihm die MessageGenPID senden soll.
{helloTime, TimeSyncPID}	Erhält die PID des **TimeSync-Prozesses**.
{helloSlot, SlotReservationPID}	Erhält die PID des **SlotReservation-Prozesses**.
{currentTime, Timestamp}	Erhält vom **TimeSync-Prozess** die aktuelle Uhrzeit.
{message, Message, OldSlot}	Erhält vom **MessageGen-Prozess** die zu sendende Nachricht (Message).
{nomessage}	Erhält den Hinweist vom **MessageGen-Prozess**, dass die Sendezeit vorbei ist.
{collision, true/false}	Bekommt vom **SlotReservation-Prozess** mitgeteilt, ob er nun auf den jeweiligen Slot senden darf (true) oder, ob eine Kollision stattfinden würde (false).
kill	Damit alle „Hilfs-Prozesse“ und sich selber zu beenden. Wird von Starter initiiert.

TimeSync:

Befehl (Senden)	Beschreibungen
{helloTime, self()}	Teilt seine PID, direkt nach der eigenen Erzeugung, dem **Sender-Prozess** mit.
{currentTime, Timestamp}	Schickt dem **Sender-Prozess** oder **MessageGen-Prozess** die aktuelle Uhrzeit als Timestamp.
Befehl (Empfangen)	Beschreibungen
{times, StationClass, TimeInSlot}	Erhält vom **Receiver-Prozess** die Stationsklasse und den Timestamp aus dem letzten Datenpaket.
{getTime, SenderPID_ReceiverPID}	Bekommt vom **Sender-Prozess** oder **Receiver-Prozess** die Anfrage der aktuellen Uhrzeit.
kill	Beendet sich selbst.

MessageGen:

Befehl (Senden)	Beschreibungen
{getSlot, self()}	Fragt den Slot für den nächsten Frame beim **SlotReservation-Prozess** an.
{message, Message, OldSlot}	Sendet die zu sendende Nachricht (Message) an den **Sender-Prozess**.

{nomessage}	Wenn die Sendezeit vorbei ist, dann wird dies dem **Sender-Prozess** hiermit mitgeteilt.
Befehl (Empfangen)	Beschreibungen
{helloSlot, SlotReservationPID}	Erhält die PID des **SlotReservation-Prozesses** die vom **Sender-Prozess** weitergeleitet wurde.
{helloTime, TimeSyncPID}	Empfängt vom **Sender-Prozess** die PID des **TimeSync-Prozesses**.
{nextSlot, NextSlot}	Empfängt vom **SlotReservation-Prozess** den nächsten Slot.
kill	Beendet sich selbst.

SlotReservation:

Befehl (Senden)	Beschreibungen
{helloSlot, self()}	Teilt seine PID, direkt nach der eigenen Erzeugung, dem **Sender-Prozess** mit.
{nextSlot, NextSlot}	Sendet den nächsten Slot (NextSlot) an den **MessageGen-Prozess**.
{collision, true/false}	Sendet dem **Sender-Prozess** ein true oder false zurück, um ihm mit zuteilen, ob eine Kollision stattfinden würde, wenn der Sender auf den Slot schicken würde.
Befehl (Empfangen)	Beschreibungen
{slot, NextSlot}	Erhält vom **Receiver-Prozess** den Slot für den nächsten Frame.
{getSlot, MessageGenPID}	Bekommt vom **MessageGen-Prozess** die Anfrage des Slot für den nächsten Frame.
{collision, Slot}	Bekommt die Anfrage vom **Sender-Prozess**, ob eine Kollision auf dem Slot möglich ist.
kill	Beendet sich selbst.

d) Spezifikation

Alle Module besitzen weiterhin eine Debug-Methode in der einige Ausgaben auf die Konsole ausgegeben werden können. Dies wird durch das Makro „DEBUG“ gesteuert. Ebenso besitzen sie ein Logfile-Makro und einen Namen als Makro definiert.

Starter:

start(ArgsList): Startet die Station.

registerAtLocalNameservice(Name): Registriert die jeweilige Station beim lokalen Namensdienst von Erlang.

unregisterAtLocalNameservice(Name): Meldet die Station beim lokalen Namensdienst von Erlang wieder ab.

Receiver:

start(...): Startet den Receiver und initialisiert weitere Prozesse.

*init(...):*Initialisiert den Socket und geht im Anschluss in die Schleife.

initSlotPositions(NumPos): Initialisert die Liste mit den Slot-Positionen.

loop(...): Stellt die Schleife dar.
{SlotsUsed, CurrentTime} isFrameFinished(...): Berechnung korrigieren
sendFreeSlots(List, _ReceiverDeliveryPID): Sendet die freien Slots an einen Hilfs-Prozesse.
analyse(Packet, ReceiverDeliveryPID, TimeSyncPID): Extrahiert die Daten aus dem Paket und loggt den Inhalt.
List extractIntervall(Binary, From, To): Extrahiert die Daten aus dem Paket von From bis To.
{true/false/corrupt, Number} getSlotNumber(SlotsUsed, Packet): Extrahiert die Slot-Nr. aus dem Paket und prüft Ob der Slot im nächsten Frame schon von einer anderen Station in Gebrauch sein wird.
true/false/corrupt willSlotBeInUse(SlotsUsed, SlotNumber): Liefert true oder false zurück, je nachdem ob der Slot im nächsten Frame von einer anderen Station in Gebrauch sein wird.
List countSlotNumberUsed(SlotsUsed, SlotNumber): Zählt die Anzahl der Stationen hoch, die den Slot SlotNumber im nächsten Frame benutzen werden.
kill(): Schickt kill-Befehl an alle Hilfs-Prozesse und schließt sich.
Receiver-Receiver Services:
delivery(stationAlive, SlotReservationPID, TimeSyncPID): Schickt nächsten Slot und die Timestamp an die jeweiligen Prozesse.
HostAddress getHostAddress(InterfaceName): Ermittelt die Hostaddress des Hosts.

Sender:
start(...): Startet den Sender und weitere Prozesse.
*loop(...):*Hauptschleife des Senders.
PIDList waitForInitialValues(MessageGenPID, PIDList): Wartet auf die PIDs des *TimeSync-Prozesses* und des *SlotReservation-Prozesses*.
HostAddress getHostAddress(InterfaceName): Ermittelt die Hostaddress des Hosts.
CheckedSlot checkSlot(Slot, SlotReservationPID): Fragt beim *SlotReservation-Prozess* an, ob der jeweilige Slot frei ist, damit keine Kollision auftritt.
sendMulticast(Message, Socket, MulticastAddr, ReceivePort, Timestamp): Sendet die Nachricht über den Multicast.

TimeSync:
start(StationClass, UtcOffsetMs, SenderPID): Startet den *Time-Sync-Prozess*.
TimeStamp getNewTime(): Ermittelt den aktuellen Timestamp der Station.
accurate(UtcOffsetMs, SyncOffsetMs, TimesReceived): Zeit wird synchronisiert, da die eigene Staion von der Klasse A ist. Erhält Anfragen von Sender u. MessageGen, berücksichtigt Zeitmitteilungen vom Receiver.
SyncOffsetNew berkley(Flag, TimeInSlot, SyncOffsetMs, TimesReceived): Realisiert den Berkley-Algorithmus.
kill(): Schließt sich selbst.

MessageGen:
start(SenderPID, StationClass): Startet die Messagegen.
loop(): Hauptschleife der Messagegen.
puffer(MessageGenPID, Data): Realisiert den Puffer zwischen Datenquelle und der Messagegen. Puffer enthält immer die aktuellen Nutzdaten.
getDataFromSource(PufferPID): Erhält die Nutzdaten aus der Datenquelle und legt sie in den Puffer.
PIDList waitForInitialValues(PIDList): Wartet auf die PIDs des *TimeSync-Prozesses* und des *SlotReservation-Prozesses*.

NewSlot getInitialSlot(): Holt den ersten initialen Slot vom *SlotReservation-Prozess*.
{NextSlot, Killed} getNextSlot(SlotReservationPID): Holt den nächsten Slot in dem im nächsten Frame gesendet werden soll.
Sendtime calcSendTime(Slot, Timestamp): Errechnet die Sendezeit, wann der *Sender-Prozess* geweckt werden muss.
waitSendTime(Sendtime): Legt sich solange schlafen bis die Sendezeit vorüber ist.
Message prepareMessage(Slot, StationClass, PufferPID): Bereitet die zu sendende Nachricht vor.

SlotReservation:
start(SenderPID): Startet den *SlotReservation-Prozess*.
loop(FreeSlots, SenderPID): Die Hauptschleife der SlotReservation.
Slot getNewSlot(): Gibt einen neuen Slot für den nächsten Frame zurück.
sendCollisionAnswer(FreeSlots, Slot, SenderPID): Sendet {collision, true} an den Sender, wenn der Slot in dem der Sender sende möchte, nicht in der Liste der freien Slots ist, sonst {collision, false}
kill(): Schließt sich selbst.

4.2. Kommunikation zwischen den Module

Die Nummern die in den jeweiligen Beschreibungen der Sequenzdiagramme angegeben werden, beziehen sich auf den jeweiligen Sequenznummern in dem Sequenzdiagramm.

a) Starten der Anwendung (**vsp4_start**)
Mit dem Startskript (startStations.sh) wird der *Starter-Prozess* initiiert (2:). Diesem Prozess werden alle notwendigen Parameter übergeben, die er im Anschluss an den *Sender-* und *Receiver-Prozess* weiterleitet. Der *Starter-Prozess* startet zuerst den *Sender-Prozess*, da dem *Receiver* die PID des Senders mitgeteilt werden muss.
Sobald der *Sender* gestartet ist, erstellt dieser den *MessageGen-Prozess* (3:) und übergibt seine eigene PID, da diese im weiteren Verlauf miteinander kommunizieren müssen. Der *Sender* benötigt für die Socket-Nutzung einen eigenen Port der aus dem ReceivePort und der jeweiligen StationNumber gebildet wird (4:).
Nach dem Erstellen des *Sender-Prozesses* erzeugt der *Starter* den *Receive-Prozess* und übergibt ihm ebenfalls alle nötigen Parameter, sowie die SenderPID. Der *Starter-Prozess* merkt sich jeweils die PIDs des *Senders* und *Receivers*, da er diese beiden Prozesse, bei Empfang des kill-Befehls über die Konsole ebenfalls ein kill schicken muss. In der Methode „registerAtLocalNameservice(Name)“ registriert sich der jeweilige *Starter* bei dem lokalen Namensdienst von Erlang mit seinem spezifischen Namen (9:), damit dieser den kill-Befehl von der Konsole als Nachricht entgegen nehmen kann.
Der *Receiver-Prozess* erzeugt den *TimeSync-Prozess* (7:) sowie den *SlotReservation-Prozess* (10:). Beide erhalten die SenderPID, damit sie sich direkt mit ihrer eigenen PID beim *Sender-Prozess* melden können (8:, 11:). Der *Sender-Prozess* wiederum schickt jeweils eine Nachricht an den *MessageGen-Prozess* damit auch dieser die PIDs des *TimySync-* und *SlotReservation-Prozesses* kennt (8.1:, 11.1:).
Somit kennen sich nun alle relevanten Prozesse einer Station untereinander.

b) Empfangen von Datenpaketen via Multicast (**vsp4_empfangen**)
1. Erzeugung des SlotReservation-Prozesses gemäß Klassendiagramm
2. Erzeugung des TimeSync-Prozesses gemäß Klassendiagramm
3. Anfrage nach aktueller Zeit zwecks Framebeobachtung (Überprüfung ob man im aktuellen oder nächsten Frame ist)
   - 3.1Erhalt der aktuellen Zeit von TimeSync
4. Collision-Zähler auf 0 setzen vor Eintritt in Loop zwecks Logging.

5. SlotsUsed initialisieren, realisiert eine Liste mit 25 Positionen (Position = Slotnr. im Frame), Position wird hochgezählt wenn eingetroffenes Packet im nächsten Frame Slotnr = Position nutzt.
6. Received-Zähler auf 0 setzen vor Eintritt in Loop zwecks Logging
7. Socket zwecks UDP-Multicast öffnen mit Multicast-Adresse, eigener Adresse und Port, Port = 15000 + Teamnummer, Teamnummer = 1

   In einer Schleife, die hier einen rekursiven Funktionsaufruf von loop bezeichnet, werden die folgenden Schritte ausgeführt:

8. Solange Station existiert, UDP-Paket empfangen
9. Solange Station exisitert, Slotnr. Aus Paket lesen mit getSlotNumber(SlotsUsed, Packet), liefert Tupel aus atom CollisionDetection = false, wenn Slotnr. Aus Packet 0 in SlotsUsed ist, sonst true. Zweites Element im Tupel die veränderte Liste SlotsUsed mit Position = Slotnr + 1

   Nach dem Empfang eines Pakets muss geprüft werden, ob bereits ein zuvor im selben Frame empfangenes Paket in dem Slot gesendet hat, in dem das aktuell empfangene Paket gesendet hat. Falls ja, wurde eine Kollision entdeckt, die wir nicht verhindern konnten, da zufällig 2 Stationen im selben Slot gesendet haben.

10. Loggen wenn Kollision entdeckt wurde.
11. Received um 1 erhöhen
12. Nutzdaten des Packets analysieren, ggf. in Funktion analyse(Packet) Loggen.
13. Stationsklasse und Sendezeit des empfangenen Pakets an TimeSync schicken
14. Aktuelle Zeit von TimeSync anfordern
    - 14.1 Aktuelle Zeit von TimeSync empfangen

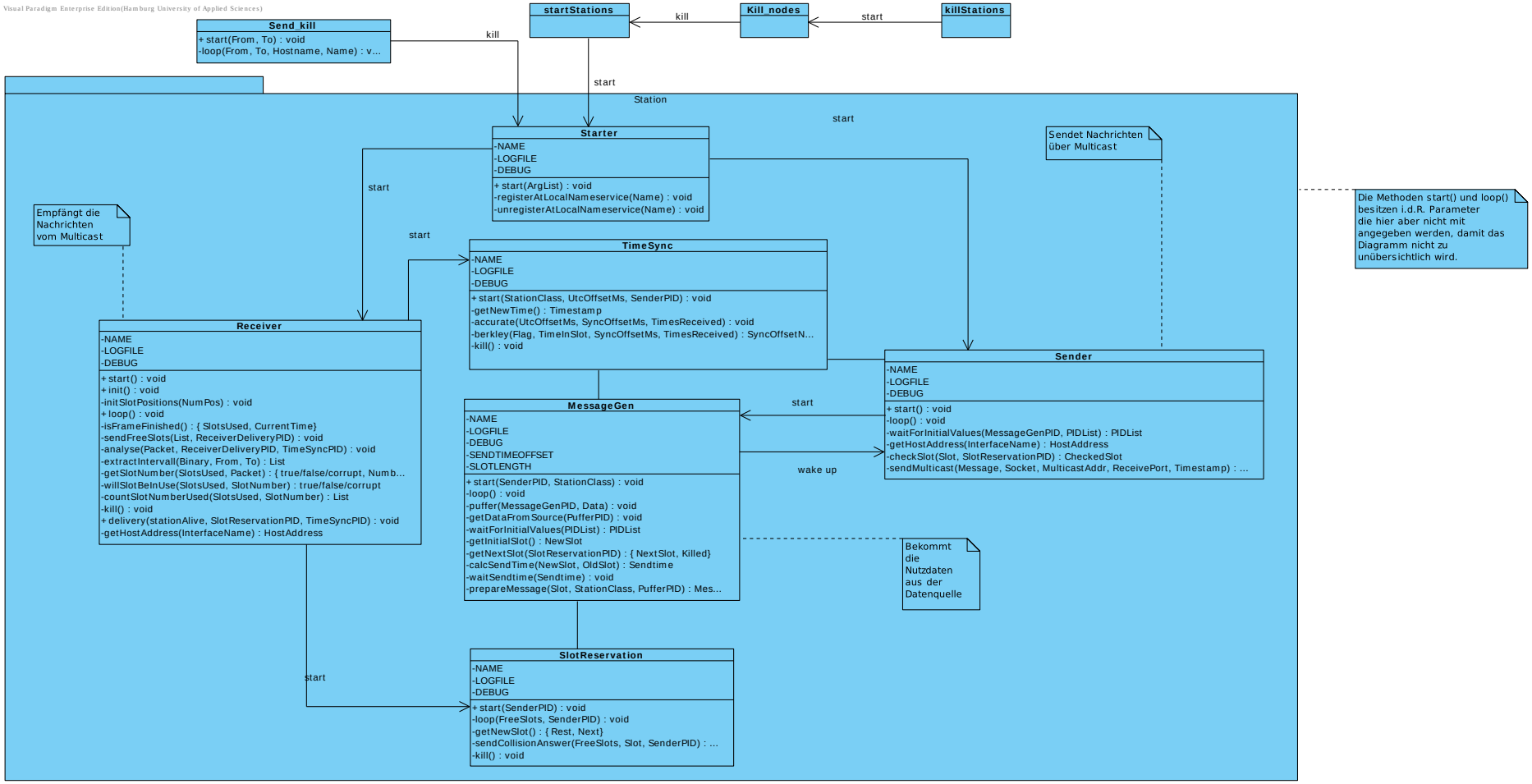
    Prüfen ob 1 Sekunde (1 Frame) vergangen ist, wenn ja, freie Slots an SlotReservation senden und neue aktuelle Zeit holen da diese den Anfang vom nächsten Frame markiert.

15. Aktuelle Zeit von TimeSync anfordern
    - 15.1 Aktuelle Zeit von TimeSync empfangen (Stellt hier den Beginn eines neuen Frames dar)
16. Freie Slots aus SlotsUsed ermitteln (wenn Position = 0) und an SlotReservation senden.
    16.1 Freien Slot an SlotReservation senden.

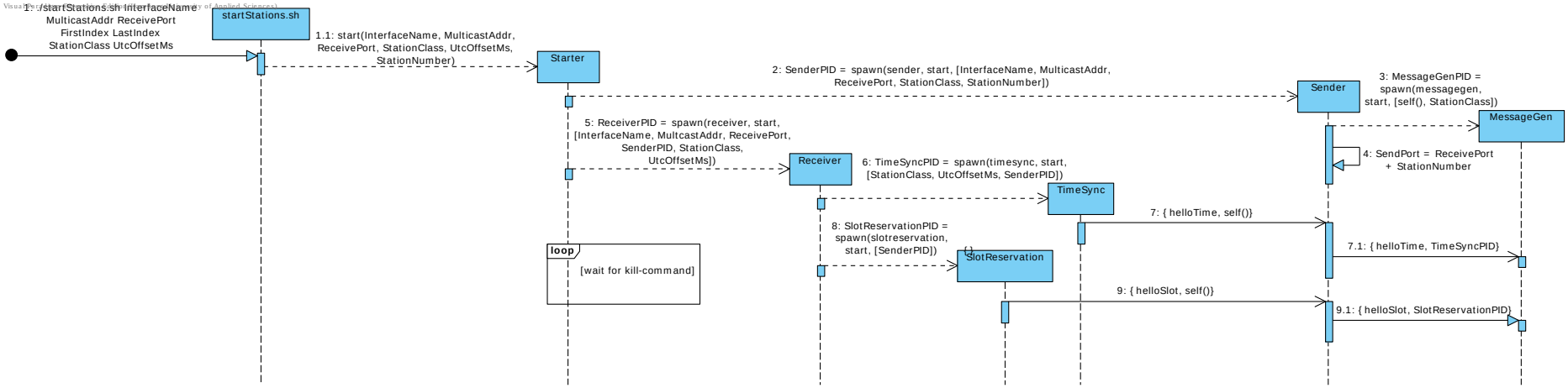
c) Senden von Datenpaketen via Multicast (**vsp4_senden**)
Zu Beginn berechnet der *MessageGen-Prozess* die Sendezeit, zu der er den *Sender-Prozess* wecken soll (1:), damit dieser die entsprechende Nachricht über den Multicast senden kann. Danach legt sich die MessageGen für diese Zeit schlafen (2:). Wurde nun die Sendezeit verpasst, dann wird dem *Sender-Prozess* eine entsprechende Nachricht zugeschickt (7:), dass es keine Nachricht zu versenden gibt. Wurde sie nicht verpasst, muss ein neuer Slot für den nächsten Frame beim *SlotReservation-Prozess* angefragt werden (3:). Im Anschluss wird die Nachricht (Message) präpariert (4:) und an den *Sender-Prozess* verschickt (5:).
Dort angekommen, muss nun geprüft werden, ob der Slot frei ist (5.1:), indem der *SlotReservation-Prozess* dafür angefragt wird. Sofern dies der Fall ist wird der aktuelle Timestamp für das Nachrichtenpaket angefordert (5.2:). Nun kann die

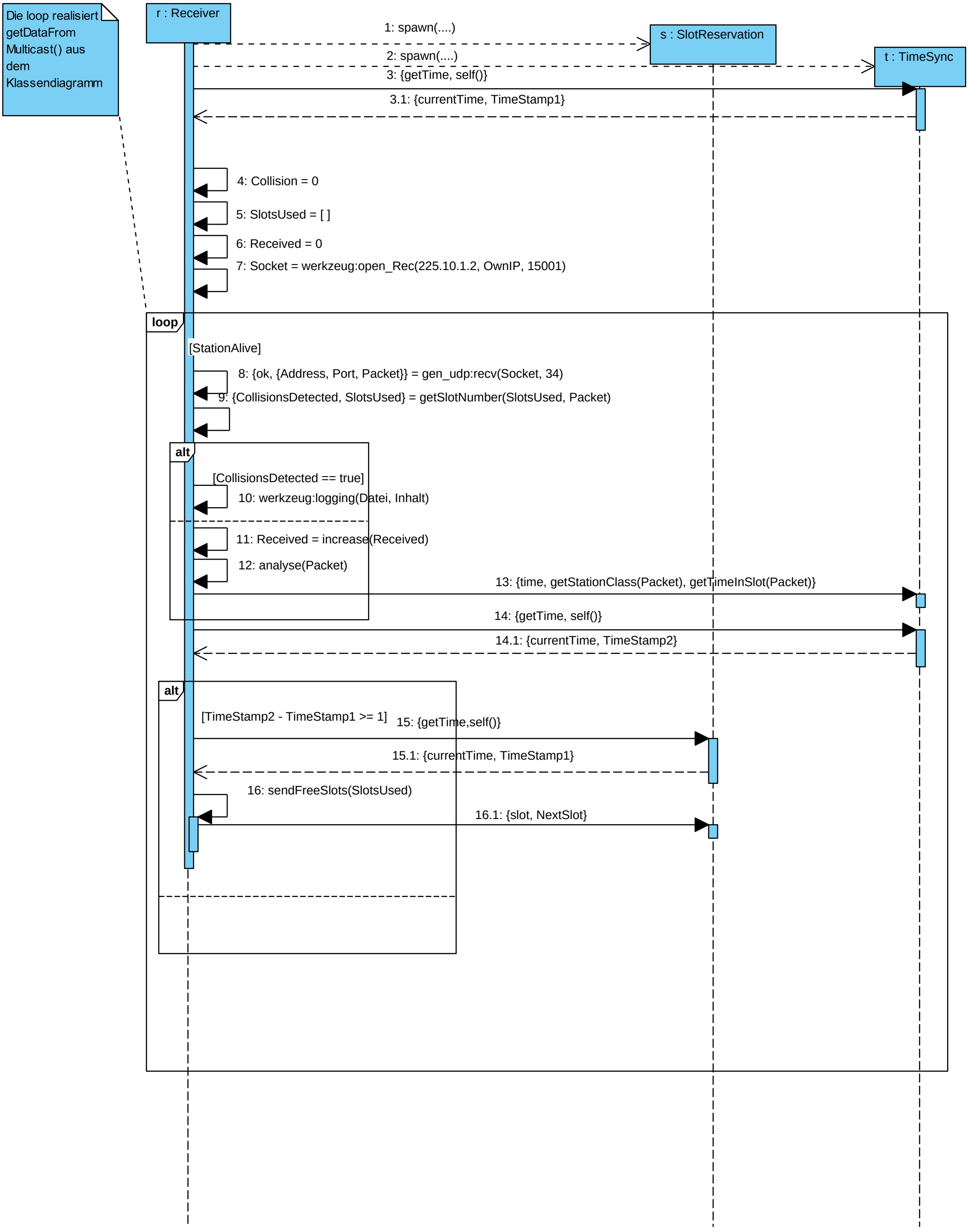
Messagen über den Multicast versandt werden (5.4:).

vsp4_klassendiagramm

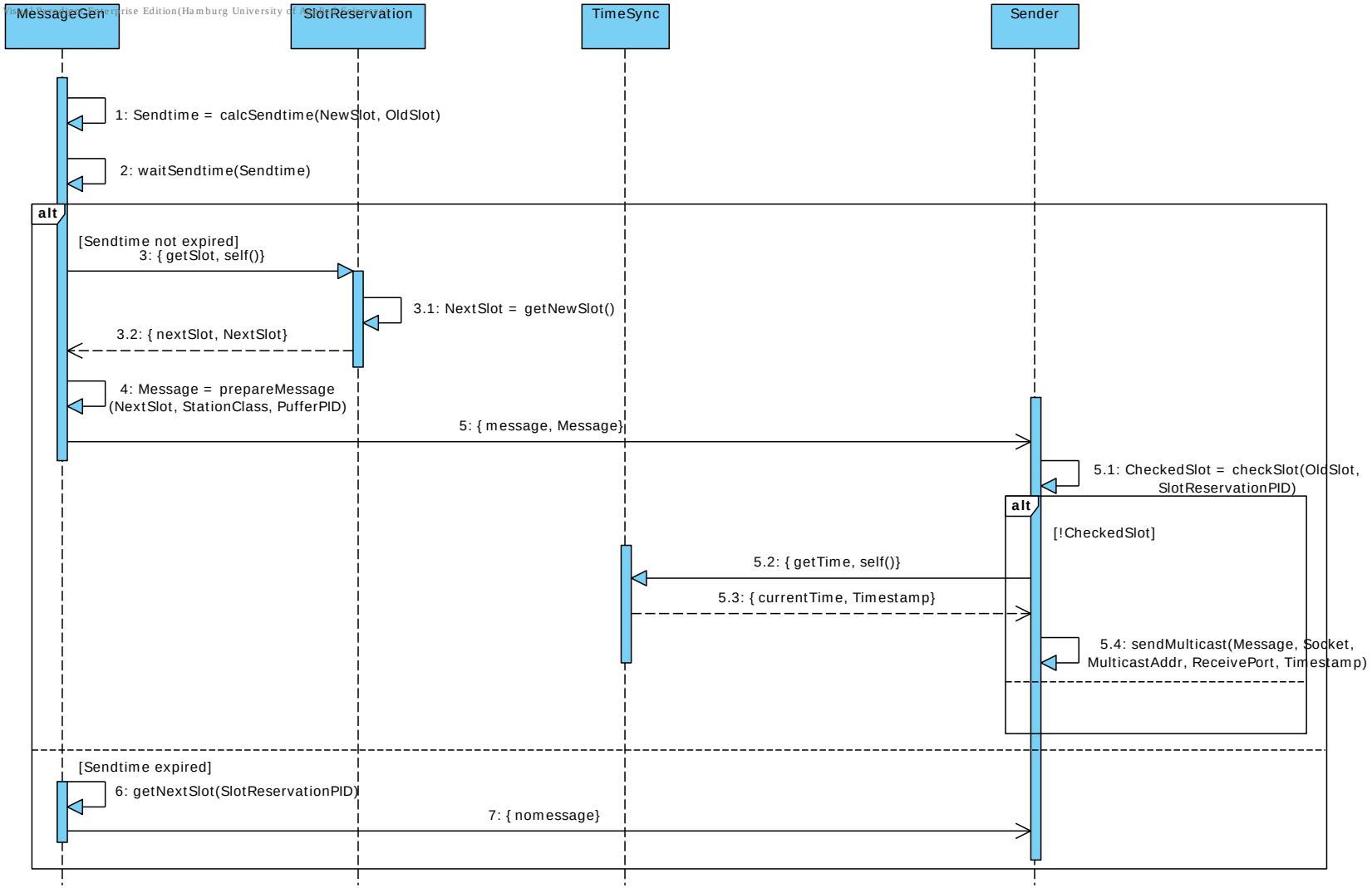


vsp4_start


1: ./startStations.sh InterfaceName
MulticastAddr ReceivePort
FirstIndex LastIndex
StationClass UtcOffsetMs
startStations.sh
1.1: start(InterfaceName, MulticastAddr,
ReceivePort, StationClass, UtcOffsetMs,
StationNumber)
Starter
2: SenderPID = spawn(sender, start, [InterfaceName, MulticastAddr,
ReceivePort, StationClass, StationNumber])
Sender
3: MessageGenPID =
spawn(messagegen,
start, [self(), StationClass])
MessageGen
4: SendPort = ReceivePort
+ StationNumber
5: ReceiverPID = spawn(receiver, start,
[InterfaceName, MultcastAddr, ReceivePort,
SenderPID, StationClass,
UtcOffsetMs])
Receiver
6: TimeSyncPID = spawn(timesync, start,
[StationClass, UtcOffsetMs, SenderPID])
TimeSync
7: { helloTime, self()}
7.1: { helloTime, TimeSyncPID}
8: SlotReservationPID =
spawn(slotreservation,
start, [SenderPID])
SlotReservation
loop
[wait for kill-command]
9: { helloSlot, self()}
9.1: { helloSlot, SlotReservationPID}



vsp4_senden


MessageGen
SlotReservation
TimeSync
Sender
1: Sendtime = calcSendtime(NewSlot, OldSlot)
2: waitSendtime(Sendtime)
alt
[Sendtime not expired]
3: { getSlot, self()}
3.1: NextSlot = getNewSlot()
3.2: { nextSlot, NextSlot}
4: Message = prepareMessage
(NextSlot, StationClass, PufferPID)
5: { message, Message}
5.1: CheckedSlot = checkSlot(OldSlot,
SlotReservationPID)
alt
[!CheckedSlot]
5.2: { getTime, self()}
5.3: { currentTime, Timestamp}
5.4: sendMulticast(Message, Socket,
MulticastAddr, ReceivePort, Timestamp)
[Sendtime expired]
6: getNextSlot(SlotReservationPID)
7: { nomessage}