Prinzipien verteilter Echtzeitsteuerungen Seminar Robotik

1 Einführung

In Zeiten von autonomen Autos und zunehmend automatisierten Produktionsketten wird es immer wichtiger, dass verteilte Steuerungen auch die harte Echtzeit unterstützen. Im Folgenden wird eine Übersicht über das Thema verteilte Echtzeitsteuerungen gegeben und auf konkrete Lösungen eingegangen. Zunächst werden wichtige Begriffe zu dem Thema definiert, in Abschnitt zwei wird auf Probleme eingegangen, die bei verteilten Echtzeitsteuerungen auftreten können. In Abschnitt drei werden noch weitere Anforderungen an das Design gestellt, im dritten Abschnitt wird zunächst auf Uhr Synchronisation eingegangen und dann auf die drei Echtzeit-Frameworks CORBA, OROCOS und aRDx, wobei mit Verteiltheit, Echtzeit und Performance mit jedem Framework ein anderer Aspekt besprochen wird. Und zuletzt wird ein kleiner Zukunftsausblick gegeben.

1.1 Begriffe

Zeitrichtlinien: Zeitrichtlinien sind ein wichtiger Bestandteil der Echtzeit. Eine Zeitrichtlinie belegt, wann ein Aktionsblock spätestens zu Ende sein muss.

Weiche Echtzeit: Bei weicher Echtzeit spricht man von Wahrscheinlichkeiten, ob eine Zeitrichtlinie eingehalten wird. Man kann also niemals vorhersagen wie lange ein Programm zum Ausführen braucht. Ein Beispiel dafür ist das Abspielen von Webvideos. Es wird in Echtzeit abgespielt. Aber dabei kann man niemals sicher sein, dass das Video flüssig abgespielt wird.

Harte Echtzeit: Harte Echtzeit allerdings, setzt zur Korrektheit des Systems voraus, dass diese Zeitrichtlinien eingehalten werden müssen. Harte Echtzeit wird dann notwendig, wenn das Überschreiten von Deadlines dramatische Folgen haben kann. Im Folgenden wird nur noch von harter Echtzeit gesprochen.

Zeitfenster: Durch das Festlegen von Zeitfenstern, setzt man für die Korrektheit des Systems voraus, dass ein Aktionsblock innerhalb eines Zeitfensters ausgeführt werden kann. Dabei unterteilt man die Echtzeit Applikation in Applikationsblöcke. Durch das Aneinanderreihen von Zeitfenstern, ergibt sich also die Laufzeit des Systems.

Verteilte Echtzeitsysteme: Verteilte Echtzeitsysteme sind Echtzeitsysteme, die dazu in der Lage sind auf mehreren Hosts zu laufen und in Echtzeit miteinander zu kommunizieren.

Verteilte Echtzeitsteuerungen: Der Schritt zur Verteilten Echtzeitsteuerung, fügt dann lediglich noch Software und Treiber zum Steuern von Robotern, Sensoren und anderen Geräten hinzu.

2 Korrektheit eines verteilten Echtzeitsystems

2.1 Hardware

Abgesehen von Hardwaredefekten, die nur mit einer Ausfallerkennung und einem Notfallprogramm behandelt werden können, muss man evtl. kleine Änderungen vornehmen um die Hardware echtzeitfähig zu machen. Zum Beispiel kann ein mehrkerniger Prozessor Probleme bereiten. Es kann auch sein, dass man das Bus-System austauschen muss, um die benötigte Zeit für Übertragungen berechenbar zu machen. Heute kann man allerdings mit spezieller Software fast jede handelsübliche Hardware echtzeitfähig machen. Konkret sind das dann Echtzeit Betriebssysteme wie FreeRTOS, QNX Neutrino oder Windows CE.

2.2 Netzwerk

Die Kommunikation zwischen den einzelnen Systemen ist ein wichtiger Punkt. Herkömmliches W-LAN kann hier kaum benutzt werden, da nicht garantiert werden kann, ob Daten Pakete in einem bestimmten Zeitraum ankommen. Auch Bluetooth fällt wegen ähnlichen Problemen weg. Deshalb wird hier meistens eine spezielle Funkverbindung oder eine LAN-Verbindung verwendet. Zum Beispiel ist nicht jede BUS Topologie für Echtzeitsysteme geeignet. BUS Systeme, die Kollisionen zulassen, können nur eine statistische Übertragungsgeschwindigkeiten garantieren. Deshalb ist für Echtzeitanwendungen spezielle BUS Hard- und Software nötig. Beispiele dafür sind der deutsche PROFIBUS (Process Fieldbus) oder Foundation Fieldbus [1].

2.3 Software

Das Framework sollte sich Robust gegenüber Fehlern in der Software verhalten. Wenn von dem System Menschenleben abhängen, kann es sogar vonnöten sein die Korrektheit der Software zu beweisen. Evtl. geworfene Exceptions oder Fehler sollten speziell behandelt werden. Sie könnten die Zeitvorschriften verletzen.

Dass Zeitliche Richtlinien eingehalten werden, ist wohl der wichtigste Punkt. Hierbei muss die Software die Hardware des gesamten Systems berücksichtigen. Ein Programm, das auf einer langsameren CPU läuft braucht länger

um eine bestimmte Anzahl von Befehlen auszuführen. Die Anderen Systeme müssen dann evtl. warten bis die langsamste CPU fertig ist. Dazu muss ein einzelnes System alle anderen kennen und auch die Netzwerktopologie, damit diese verteilt in Echtzeit laufen können [3].

Die CPU kann noch weitere Probleme hervorrufen. Beispielsweise hat ein anderes Modell bei gleicher Taktfrequenz eine bessere Sprungvorhersage und somit wird der gleiche Code mit einer geringeren Anzahl an Takten ausgeführt. Auch das muss von der Software berücksichtigt werden.

2.4 Ressourcen

Alle Ressourcen, die von dem Framework verwendet werde, sollten festgelegte Deadlines einhalten können. Wenn ein Task von einer nicht zeitlich vorhersehbaren Ressource abhängt, kann es passieren, dass Deadlines überschritten werden. Beispielsweise können Webserver als Informationsquelle nicht verwendet werden, da die Reaktionszeit zum Beispiel von der Anzahl der Nutzer und der Internet Qualität abhängt [2].

3 Designziele

Wenn man sich verschiedenen verteilten Echtzeitsteuerungen genauer anschaut werden schnell noch weitere Anforderungen an das System wichtig. Dabei ist das Hauptziel die Programmierung von Applikationen mit dem Framework möglichst einfach zu gestalten.

3.1 Anforderungen

Das Framework sollte modular aufgebaut sein. Ein System sollte jederzeit durch weitere Komponenten erweitert werden können.

Es sollten High-Level Programmiersprachen unterstützt werden und das System sollte Objekt orientiert aufgebaut sein um den Entwicklungsprozess zu vereinfachen.

Für das Software Framework sollte schon existierende Middleware verwendet werden. Die eigene Entwicklung von Middleware ist sehr komplex und Zeitaufwendig, da verschiedene Betriebssysteme und Treiber unterstützt werden müssen. Schon vorhandene Middleware wie CORBA löst viele der Probleme schon hervorragend.

Wenn sich die Hardware auch nur geringfügig ändert, muss wegen oben genannten Gründen auch die Software angepasst werden. Hier gilt es also beim Design dies Änderungen möglichst gering zu halten, oder sie sogar zu Automatisieren. Beispielsweise kann der selben Code auf einer langsameren CPU zu Fehlern führen, da Zeit Vorschriften nicht eingehalten werden.

Neue Treiber und Geräte sollten einfach in das System eingebunden werden können. Auch die Kompatibilität mit verschiedenen Betriebssystemen und Programmiersprachen ist wichtig.

Eine hohe Performance ist wichtig, damit auch komplexe Algorithmen in Echtzeit ausgeführt werden können. Beispielsweise wenn ein Neuronales Netz in einem Auto zur Laufzeit Personen erkennen soll [3].

4 Lösungsvorschläge

4.1 Uhr Synchronisation

Bei der Uhr Synchronisation muss man beachte, dass es niemals perfekt synchronisierte Uhren gibt. Das liegt allein schon daran, dass die CPU nicht im gleichen Takt Rhythmus laufen. Somit wird es immer eine kleine Abweichung geben die man beachten muss [1].

Wenn ein Knotenpunkt die Uhrzeit an alle anderen Knoten schickt, kommt diese je nach Netzwerktopologie nach einer zufälligen Verzögerung an. Hier wird also ein Verfahren benötigt um die Uhren unabhängig von der Verzögerung zu ermitteln. Die meisten Verfahren lösen dies, indem sie die Verzögerung zwischen den beiden Knoten ermitteln. Dazu wird ein Signal hin und her geschickt und es werden die Zeiten messen. Dadurch kann recht genau der Zeitunterschied zwischen den Uhren ermittelt werden [1].

Es wurden zu diesem Zweck Verfahren für verschiedene Anwendungsgebiete entwickelt. Beispiele dafür sind Christian Verfahren von Fetzer (1994), van Oorschot (1993), Schedl (1996) oder Mills (1991) [1].

4.2 CORBA

CORBA [4] (Common Object Request Broker Architecture) wird seit Oktober 1991 von OMG entwickelt. Die neueste Version wurde November 2012 veröffentlicht. Unter dem Namen CORBA 3.3 [4].

CORBA ist eine Spezifikation, die festlegt wie man ein verteiltes System aufbaut. Allerdings sind freie Implementationen in verschiedenen Programmiersprachen verfügbar. Die mit diesen Implementationen erzeugten Applikationen laufen unabhängig von der Hardware Betriebssystemen [4].

CORBA ist unter anderem auch für die Verteiltheit von OROCOS Anwendungen zuständig [5].

Designziele von CORBA sind die Unabhängigkeit von Plattform und Sprache und ein objektorientiertes Framework zu schaffen, das abstrakt von Prozessor, Speicher und Kommunikation ist. Auch will man dem Programmierer mit objektorientierten Programmiersprachen wie C++, Java oder Smalltalk die Möglichkeit geben ausschließlich auf High-Level. CORBA kann mit den gängigen Datentypen plattformunabhängig umgehen. Das beinhaltet zum Beispiel auch Little- und Big-Endian [4].

Jedoch ist CORBA ein sehr umfangreiches Framework und die Programmierer müssen viel Arbeit in die Einarbeitung stecken. Von der Seite der Performance ist der Overhead bei CORBA größer verglichen zu anderen Technologien [4].

CORBA besteht aus zwei Teilen. CORBA IDL ist dafür zuständig das verteilte System zu erzeugen und CORBA ORB ist eine konkrete Implementation für die Kommunikation über Netzwerk. Aus dem erzeugten System und ORB ergibt sich die ausführbare verteilte Applikation [4].

Das bedeutet der Programmierer kann Applikationen schreiben ohne sich darüber Gedanken machen zu müssen, dass ein Objekt sich auf einem anderen Server befindet. Er kann also die Methode von einem Server Objekt aufrufen als wäre es ein lokales Objekt [4].

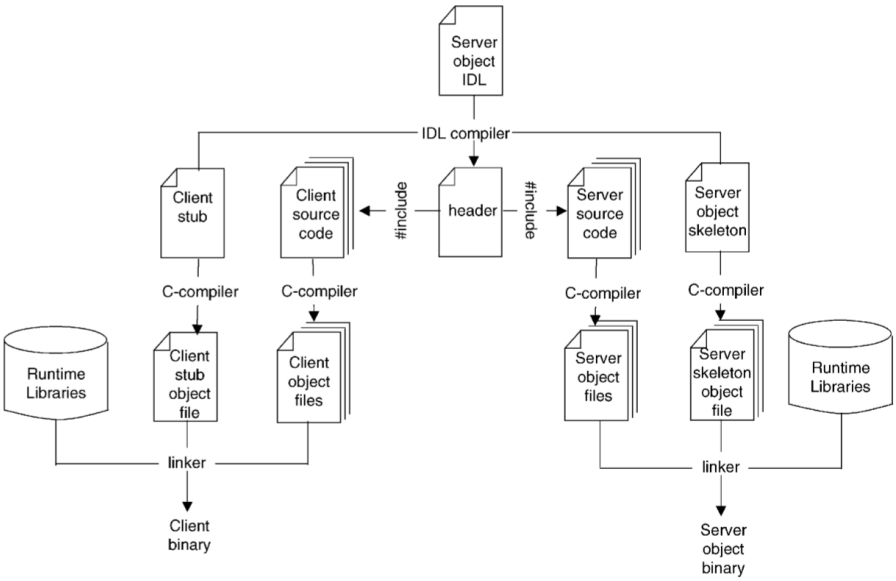

Fig. 1. Hier ist der Entstehungsprozess des Maschinencodes einer Anwendung mithilfe des CORBA Frameworks zu sehen [4].

4.2.1 Verteilte Steuerung

Auch muss beim Einsatz von CORBA bedacht werden, dass CORBA nur ein verteiltes System ist. Das bedeutet CORBA liefert keine Treiber, Bibliotheken oder Sonstiges zum Steuern von Robotern. Beim Einsatz in der Robotik muss CORBA also noch um diese erweitert werden [4].

4.2.2 Echtzeit

CORBA alleine ist allerdings noch nicht echtzeitfähig. Das Bedeutet die Spezifikation muss zu Real-Time CORBA (RT CORBA) erweitert werden. Nach mehreren Anfragen hat OMG dann eine feste Spezifikation von RT CORBA veröffentlicht. Implementationen sind in Java und C++ verfügbar. RT CORBA läuft auf allen Betriebssystemen die dem POSIX RT Betriebssystem Standard entsprechen [4].

4.2.3 CORBA Object Services

Object Services sollen die Interaktion zwischen Client und Server vereinfachen. Ein Service läuft auf einem Server und ist im gesamten System verfügbar. Im Folgenden sind einige der Services beschrieben: Der **Naming Service** kann Systemweit Objekten eindeutige Namen vergeben. Mit dem **Life Cycle Service** können CORBA Objekte erzeugt und zerstört werden. Mit dem **Event Service** können events anhand eines eindeutigen Namens ausgelöst werden. Außerdem gibt es noch Services für Nebenläufigkeit, Zeitsynchronisation, Sicherheit und viele weitere [4].

4.2.4 CORBA IDL (Interface Definition Language)

CORBA IDL besteht aus einer Sprache um Serverobjekte zu definieren und einem Compiler um Maschinencode sowohl für den Server als auch für den Client zu kompilieren, dem CORBA IDL Compiler. Die Sprache ist rein deklarativ und liefert keine Syntax um die Funktion einer Methode zu beschreiben. Dafür gibt es Verknüpfungen zu vorhandenen Programmiersprachen wie C, C++, Java. Der Entstehungsprozess einer auf CORBA basierenden Applikation, der in Abbildung 1 zu sehen ist, wird im Folgenden genauer beschrieben [4].

Das bedeutet auf der Server Seite wird mit CORBA IDL das Objekt deklariert. Daraus wird dann mit dem CORBA IDL Compiler zum Beispiel ein C++ Header und der Code für das Skelett des Objektes generiert. In C++ definiert man dann den Methodeninhalt. Alles zusammen wird dann mit einem C++ Compiler zu Maschinencode kompiliert. Zusammen mit dem ORB und den Object Services ergibt das die Serverapplikation [4].

Auch auf der Client Seite wird der Header benötigt um die Methoden des Objektes aufzurufen. Auf der Client Seite wird der Code zu dem Header aus der Objektdeklaration mit dem IDL Compiler generiert. Aber statt dem eigentlichen Methodeninhalt generiert der IDL Compiler Code um mit dem ORB zu interagieren, der wiederum mit dem Server Objekt kommuniziert. Zusätzlich fügt man nun noch den C++ Code der Applikation hinzu. Alles zusammen wird mit einem C++ Compieler zu der ausführbaren Clientanwendung compiliert [4].

Zusätzlich zu den Objekt Headern werden dem Client Code CORBA Exceptions bereitgestellt. Somit kann die Clientanwendung zum Beispiel auf einen Verbindungsabbruch reagieren [4].

4.2.5 CORBA ORB (Object Request Broker)

CORBA ORB sorgt für die Kommunikation zwischen Client und Server. Dies geschieht meistens mit dem Internet Standard Protokoll IIOP (Internet Inter-ORB Protocol). Bei auftretenden Fehlern wirft ORB CORBA Exceptions. ORB wird der Client Implementation bereitgestellt [4].

Eine Interaktion der Client Implementation mit ORB könnte folgendermaßen aussehen:

Zunächst verlangt der Code nach einer Referenz auf das Objekt. Anhand eines eindeutigen Namens macht ORB dann Server und Objekt ausfindig. Meistens bedeutet das IP Adresse und Port des Servers herauszufinden. Ist das Objekt gefunden wird eine Referenz zurückgegeben [4].

Wird jetzt eine Methode des Objekts ausgeführt, öffnet ORB eine Socketverbindung mit dem Server. Über diese wird dann mitgeteilt welche Methode ausgeführt werden soll. Parameter und Rückgabewerte werden dabei von ORB verpackt und über die Verbindung versendet [4].

4.3 OROCOS

Die Entwicklung von OROCOS [5] (Open Robot Control Software) wurde 2001 gestartet. Ziel war es ein freies Softwareframework für plattformunabhängige erweiterbare Echtzeitsteuerungen zu schaffen. OROCOS läuft auf mehreren Betriebssystemen: Linux, RTAI, RTLinux, Ecos [5].

OROCOS besteht aus drei Teilen. Der Kinematics and Dynamics Library, eine Hilfsbibliothek für die Steuerung von Robotern, der Bayesian Filtering Library und der OROCOS Toolchain, die den Kern des Projektes bildet. Im Weiteren wird nur auf die OROCOS Toolchain näher eingegangen [5].

Das Design ist auf Portabilität und Konfigurierbarkeit ausgelegt. Die einzelnen Komponenten sollen abstrakt gehalten werden, um für Flexibilität zu sorgen [5].

4.3.1 OROCOS Toolchain

Ein Entwicklungszyklus mit der OROCOS Toolchain sieht folgendermaßen aus. Zunächst legt man über die OROCOS Spezifikation Code und Struktur der Applikation fest. Dann erzeugt die OROCOS Toolchain daraus den angepassten Code. Zusammen mit dem OROCOS Code wird mit einem Compiler der Maschinencode für die verschiedenen Architekturen erzeugt [5].

OROCOS ist zunächst einmal nur eine Echtzeitsteuerung. In dem Projekt ist aber schon vorhergesehen, dass man OROCOS zu einer verteilten Echtzeitsteuerung erweitern kann. Dafür wird die Middleware CORBA verwendet [5].

4.3.2 OCC (OROCOS Control Core)

Zwischen der Hardware und der Benutzer Applikation sitzt der OCC. Er ist dafür zuständig anhand der Benutzer Applikation und der Sensordaten in Echtzeit die Roboter zu steuern [5].

Die Architektur des OCC, die in Abbildung 2 zu sehen ist, ist aus neun Komponenten aufgebaut. Jede Komponente wird Eventbasiert ausgeführt. Keine Komponente darf Schleifen beinhalten. Wird also eine Komponente aufgerufen, wird jede Codezeile nur einmal ausgeführt. Dadurch ist jede Komponente deterministisch und man kann eine maximale Laufzeit für den Code berechnen, was das ganze Konstrukt wiederum Echtzeitfähig macht.

Der Code einiger Komponenten muss erst noch vom Entwickler währen der Entwicklung geschrieben werden [5].

4.3.3 OCC Komponenten

Der **Scanner** und **Actuator** sind für das Auslesen von Sensoren und das Ausführen von Bewegungen zuständig. Der **Generator** generiert anhand der Code befehle und der Sensordaten neue Bewegungspunkte. Der **Observer** berechnet anhand von Scanner und Generator Schätzungen. Der

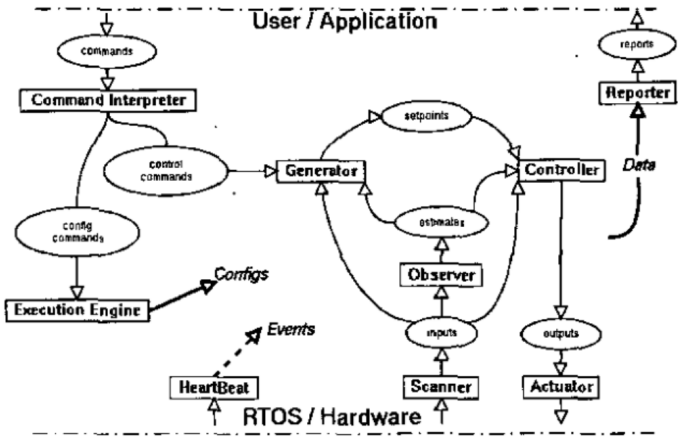

Fig. 2. Hier ist die Architektur des harten Echtzeitkerns von OROCOS zu sehen [5].

Controller berechnet letztendlich die tatsächlichen Bewegungspunkte, die er dem Actuator übergibt. Diese berechnet er mithilfe der Daten von Generator, Observer und Scanner. Der **Command Interpreter** interpretiert die User Befehle und übergibt die interpretierten befehle dem Generator. Befehle für die Konfiguration werden der Execution Engine übergeben. Da der Command Interpreter mit befehlen von außerhalb arbeitet, läuft er nicht in Echtzeit. Die **Execution Engine** erstellt und konfiguriert die Komponenten. Die **Heart Beat** Komponente erzeugt anhand einer Echtzeituhr Events. Mit jedem Takt wird jede Komponente einmal ausgeführt. Die **Reporter** Komponente ist dafür zuständig Informationen an den Rest des Systems weiterzugeben [5].

4.4 aRDx

Bei aRDx [6] (agile Robot Development – nextgeneration) handelt es sich um eine Weiterentwicklung des Frameworks aRD. aRDx ist eine Echtzeit Robotersteuerung. Unter dem Echtzeit Betriebssystem QNX laufen damit sogar verteilte Applikationen. aRDx wurde schon ausführlich an dem humanoiden Roboter Agile Justin getestet [6].

aRDx wurde mit dem Gedanken entwickelt, dass die Performanceschwächen aktueller Robotik Frameworks nicht an der Hardware liegen können, sondern an der Software. Deshalb liegt das Hauptaugenmerk bei der Entwicklung von aRDx auf der Performance. Somit wurde ein erheblicher Teil der Arbeit an aRDx durch Performantestes und Vergleichen mit anderen Frameworks ausgefüllt [5].

4.4.1 Design

Mit dem Focus auf die Performance wurden folgende Anforderungen an das Design von aRDx gestellt:

Das System ist aus Komponenten aufgebaut.

Die Kommunikation zwischen den Komponenten geschieht in Form von Paketen über Channel mit einer eindeutigen Channel ID. Channel sind abstrakt gehalten. Der Programmierer muss also nicht wissen was unter der Haube von Channeln passiert [6].

Channel können lokal oder aber auch zwischen verschiedenen Hosts sein. Sind sie lokal wird die Technologie zero-copy(das Kopieren von Daten wird nicht von der CPU

erledigt) genutzt um sich CPU Takte zu sparen und die Performance erheblich zu erhöhen. Ansonsten wird copy-once(Daten werden nur den hosts geschickt, die sie haben wollen) verwendet [6].

Ressourcen sind statisch, werden also nur einmal beim Start des Systems erstellt und können während der Laufzeit nicht mehr verändert werden. Wenn das System dynamisch Ressourcen erstellen dürfte, müsste zum Beispiel Arbeitsspeicher dynamisch allokiert werden. Da aber nur begrenzter Arbeitsspeicher zu Verfügung steht würde das die Echtzeitvorschrift verletzen [6].

Daten werden nicht serialisiert. Dadurch spart man sich einen zusätzlichen Rechenschritt. Aber wenn nötig, könnte Serialisierung noch im Nachhinein eingebaut werden.

Es wird ein effizientes System für die Synchronisation von Daten verwendet, damit nicht noch weitere Threads laufen müssen.

Es dürfen keine weiteren Threads laufen. Dadurch wird die Priorisierung der Threads für den Entwickler vereinfacht.

Es wird eine leichtgewichtige API benutzt um die Entwicklung zu vereinfachen [6].

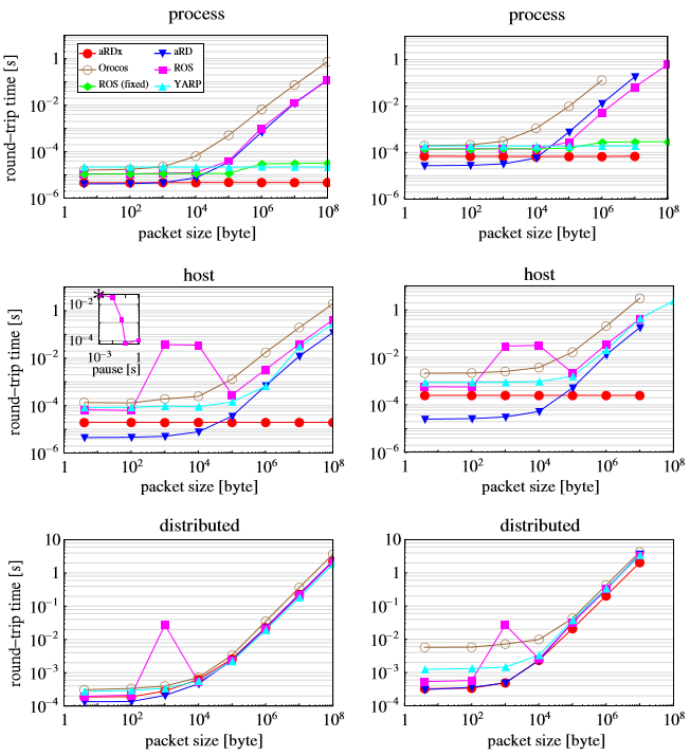

Fig. 3. In der Linken Spalte sind die Ergebnisse des Performace Tests für einen Client zu sehen, in der rechten für 20 [6].

4.4.2 *Performance-Tests*

Im Laufe der Arbeit an aRDx wurden verschiedene Performance-Tests ausgeführt. In einigen konnte gezeigt werden, dass aRDx wesentlich performanter ist als die anderen Frameworks. Bei allen anderen Frameworks konnten Schwächen gefunden werden, die aRDx nicht hat. Verglichen wurde aRDx mit den Roboter-Frameworks ROS, YARP, aRD und OROCOS [6].

Im Folgenden wird ein Test zur Ermittlung der Kommunikationsgeschwindigkeit beschrieben. Mehr Details und weitere Tests werden in [6] beschrieben:

Ein Host schickt Datenpaket einer festen Größe an mehrere Clients. Diese antworten sofort, wenn sie das Datenpaket erhalten haben und schicke dasselbe Datenpaket wieder zurück. Dieser Vorgang wird mehrere Male hintereinander durchgeführt. Dabei wird die Zeit für einen Zyklus gemessen [6]. Der Test wird in drei verschiedenen Umgebungen ausgeführt. Einmal laufen alle Komponenten in einem einzigen Prozesse(process), einmal läuft jede Komponente in einem eigenen Prozess(host) und einmal läuft jede Komponente auf einem eigenen Prozess(distributed). Der Versuch wurde mit verschiedenen Clientzahlen und Paketgrößen auf den verschiedenen Frameworks durchgeführt [6]. Die Testergebnisse in Abbildung 3 zu sehen, zeigen zum Beispiel, dass bei process und host einige Frameworks massive Schwächen zeigen, sobald die Paket Größe steigt. Bei distributed hingegen verhalten sich fast alle Frameworks ähnlich [6].

5 Zukunftsausblick

Ein interessanter Aspekt, der auch in der Zukunft in Bezug auf verteilte Echtzeitsteuerung noch wichtiger werden könnte, sind Drohnen. Drohnen können sich nicht auf eine Echtzeit Netzwerkverbindung verlassen. Zum Beispiel könnten Hindernisse zwischen den Drohnen zu Verbindungsabbrüche führen.

6 Zusammenfassung

Im Zusammenhang mit verteilten Echtzeitsteuerungen, liefert RT CORBA eine sehr gute Basis, wenn man viel selber machen will. OROCOS hat einen sehr großen Funktionsumfang und befindet sich schon seit vielen Jahren bei etlichen Systemen im Einsatz. Das eher jüngere Framework aRDx dagegen ist die richtige Wahl, wenn man viel Wert auf Performance legt.

References

[1] Johan Nilsson, *Real-Time Control Systems with Delays*. Lund, 1998.
[2] John A. Stankovic, Tian He, Tarek Abdelzaher, Mike Marley, Gang Tao, Sang Son and Chenyang Lu , *Feedback Control Scheduling in Distributed Real-Time Systems*. University of Virginia, 1995.
[3] Aloysius Ka-Lau Mok, *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Enviroment*. Massachutes Institute of Technology, 1977.
[4] Victor Fay-Wolfe, Lisa C. DiPippo, Gregory Cooper, Russell Johnston, Peter Kortmann, and Bhavani Thuraisingham, *Real-Time CORBA*. San Diego, 2000.
[5] Herman Bruyninckx, Peter Soetens, Bob Koninckx , *The Real-Time Motion Control Core of the Orocos Project* . Belgien, 2003.
[6] Tobias Hammer, Berthold Bäuml , *The Communication Layer of the aRDx Software Framework: Highly Performant and Realtime Deterministic*. Deutschland, 2014.