Conetex Contract Processing

# Voraussetzungen

Es gibt zwei Voraussetzung für eine Public-Key-Infrastruktur:

* Der private Schlüssel darf nur dem Teilnehmer bekannt sein, der den Schlüssel besitzt. Der private Schlüssel darf also nicht publiziert oder entwendet werden.
* In einem Netzwerk muss der öffentliche Schlüssel sicher der Identität eines Teilnehmers zugeordnet sein. Dies kann durch bekannte Handshake-Verfahren sichergestellt werden.

Sind beide Voraussetzungen erfüllt, können Daten mit Hilfe des privaten Schlüssels signiert werden. Die Signatur kann mit Hilfe des öffentlichen Schlüssels validiert werden. Ausgetauschte Daten sind dadurch vor Manipulation geschützt. Außerdem lässt sich die Herkunft der ausgetauschten Daten eindeutig identifizieren. Auf diesem Verfahren beruht Code-Signing und die digitale Unterschrift auf Dokumenten.

# Abstract

Im Folgenden wird eine Erweiterung des Verfahrens beschrieben.

Die Erweiterung bezieht sich auf drei Aspekte:

* Signierung des Zustands von Anwendungen, die durch ein Netzwerk verteilter Teilnehmerknoten ausgeführt werden
* Signierung des Codes derartig verteilter Anwendungen
* Reproduktion des Anwendungszustands

Die Teilnehmer am Protokoll legen sich auf Code und Zustand fest, indem sie beides gemeinsam signieren. Zustand kommt den Daten eines ausgefüllten Formulars gleich. Code kommt Handlungsanweisungen gleich. Verträge kombinieren Formular und Handlungsanweisungen beziehungsweise Rechte auf Handlungen (Siehe dazu „Quest“). Die Kombination aus Code und Zustand kommt also einem digitalen Vertrag, einem Kontrakt zwischen Teilnehmern gleich.

Durch das Validieren der gegenseitigen Signierung ist es Teilnehmerknoten möglich, Konsens bezüglich Code und Anwendungszustand festzustellen. Ein Protokoll zum Behandeln von Dissens kann als expliziter Bestandteil des Kontrakts implementiert werden. Wird der Dissens nicht ausgeräumt, brechen die Teilnehmerknoten die Verarbeitung ab.

# Referenzimplementierung

In der Referenzimplementierung des „Conetex Contract Processing“ wird der Status der Anwendung zunächst im XML-Format gespeichert. Denkbar sind auch andere Formate.

Teilnehmende Knoten werden im XML in einer Struktur des Typs „Participant“ beschrieben. Der komplexe Typ „Participant“ bestehend aus den Attributen Nickname, Mailadresse und öffentlichem Schlüssel.

Die verschiedenen Rollen der teilnehmenden Knoten werden im XML in Strukturen vom komplexen Typ „Duty“ definiert. Der komplexe Typ „Duty“ enthält das Attribut „responsibleParticipant“ vom Typ „Participant“. Von Duty abgeleitete Typen enthalten den Code, der das Verhalten der Rolle implementiert. Dieser Code wird von demjenigen teilnehmenden Knoten ausgeführt, der responsibleParticipant dieser Duty ist. Dem teilnehmenden Knoten wird also durch Duty seine Rolle im Kontrakt zugewiesen.

Empfängt ein Knoten ein XML, so durchsucht er das XML nach ihm zugewiesenen „Duties”. Er überprüft jeweils, ob er zu Recht als „responsibleParticipant” aufgeführt wird. Falls ja, hat der Knoten festgestellt, dass er derjenige teilnehmende Knoten ist, dem das in Duty implementierte Verhalten zukommt. Er führt den Code der Duty aus. Duties werden nacheinander gemäß ihrer Reihenfolge im XML ausgeführt. Im Regelfall wird der Knoten durch das Ausführen des Codes den Zustand des Contracts ändern. Die Änderungen des Contracts werden gespeichert, signiert und zur weiteren Verarbeitung an andere die Teilnehmerknoten geschickt. Dies allerdings vorbehaltlich der vorherigen Validierung des Contracts inklusive der Reproduktion seiner bisherigen Zustände.

## Contract Processing Language

Um allen teilnehmenden Knoten die Reproduktion des Anwendungszustands zu ermöglichen, verfügen alle Knoten über eine Implementierung der gleichen Laufzeitumgebung. Das heißt, gleicher Code liefert auf allen Knoten das gleiche Ergebnis. Dies kann derzeit nur durch die Implementierung einer eigenen Sprache sichergestellt werden. Die Contract Processing Language der Referenzimplementierung ist FLDSMDFR[[1]](#footnote-2). Die Contract Processing Language hat folgende Eigenschaften:

- statische Typisierung

- Primitive Datentypen

- numerische Datentypen sind ausschließlich ganzzahlig (Integer, Long, BigInteger)

- Der einzige Zeichen-Typ ist ASCII. Die maximale Länge von Strings wird bei der Typ-Definition angegeben. Unicode-Strings müssen als Base64 codiert werden.

- Der Boolsche Datentyp wird wird im XML durch die Strings "true" und "false" codiert.

- Instanzen primitiver Datentypen werden explizit durch eine entsprechende Anweisung erzeugt

- Für die numerischen Datentypen stehen die elementare Arithmetik (+, -, \*, /, mod) zur Verfügung

- Innerhalb der arithmetischen Operation können die verschiedenen numerischen Datentypen benutzt werden. Numerischer Überlauf führt zum Abbruch der gesamten Verarbeitung. Die Laufzeitumgebung liefert eine Fehlermeldung.

- Die arithmetischen Operationen können verschachtelt werden.

- Für den Zeichen-Typ stehen die Operationen search, substring, concat zur Verfügung

- Für alle primitiven Datentypen stehen die Vergleichs-Operationen =, <, > zur Verfügung

- Für die Vergleichs-Operationen und den boolschen Datentyp stehen die boolschen Operationen "Not", "And", "Or", "XOr" zur Verfügung

- Die boolschen Operationen können verschachtelt werden.

- Für den Kontrollfluss stehen if-else, sowie die while-Schleife zur Verfügung

- Complexe Datentypen sind Kombinationen aus Feldern primitiver oder complexer Typen

- Instanzen complexer Datentypen werden explizit durch eine entsprechende Anweisung erzeugt

- Zuweisungen unter Feldern primitiven Typs sind nur unter Übereinstimmendem Basis-Datentyp (Numerisch, String, Boolean)

- Complexe Datentypen können aus einem Supertyp abgeleitet werden (einfache Vereerbung)

- Zuweisungen unter Feldern complexen Typs sind nur möglich wenn der Typ des Zielfeld dem Typ des Quell-Feldes entspricht oder wenn der Typ des Quell-Feldes vom Typ des Zielfeldes erbt.

- Es können Funktionen innerhalb von Complexen Datentypen definiert werden.

- Funktionen können Rückgabewerte haben (Commando "return")

- Gemäß Datentyp können Funktionsaufrufe innerhalb von arithmetischen, Vergleichs- oder boolschen Operationen benutzt werden.

- Rekursion ist nicht möglich

- Nebenläufigkeit, Multithreading ist nicht möglich

## Policy

Jede weitere Funktionalität darf nur dann in den Sprachumfang aufgenommen werden, wenn sichergestellt werden kann, dass jeder Teilnehmerknoten bei der Reproduktion des Anwendungszustands zum gleichen Ergebnis kommt.

Entsprechend ist kein Zugriff auf die individuellen Systemressourcen (Filesystem, Uhrzeit, Gleitkommaeinheiten) vorgesehen.

## Ablauf des Contract Processing

Enthält der Code eine Zuweisung auf ein Feld, dass über die Wurzel-Struktur "Contract" adressierbar ist, so kommt dies der Änderung des Anwendungszustands gleich. Das geänderte XML wird daher neu signiert. Die Signatur wird innerhalb des XMLS in einer Struktur zusammen mit Nickname, Mailadresse und öffentlichem Schlüssel gespeichert. Der öffentliche Schlüssel wird in Base64 kodiert.

// 1 eine Anfrage machen ...

Der als XML vorliegende signierte Status kann nun unter teilnehmenden Knoten ausgetauscht werden. Für den Austausch sind diverse Protokolle (http, FTP, Socket, Mail, File-System ...) vorgesehen.

Höhere Funktionalität wird von der Laufzeitumgebung lediglich abstrakt zur Verfügung gestellt. Der Code der Duty fordert diese Funktionalität mit dem Commando "Quest" an und erhält von der Laufzeitumgebung einen Rückgabewert primitiven Typs. Die Laufzeitumgebung beanwortet Quests mithilfe von Plugins, die für bestimmte Quest registriert werden können. Wir bezeichnen derartige Plugins als Agent (Software-Agent). Im einfachsten Fall ist der Agent eine GUI, die einem menschlichen Betreiber des Knotens ermöglicht, Daten manuell einzugeben. Der Agent GUI ist der Default-Agent, der Quests beantwortet, auf die kein Agent registriert wurde. Seitens der Laufzeitumgebung ist damit sichergestellt, dass jeder Quest beantwortet werden kann.

Um eine Reproduktion des Anwendungszustands durch andere Knoten zu ermöglichen, werden die Rückgabewerte zu jedem Quest in einer Stack-Struktur innerhalb des XMLs gespeichert und stehen dadurch dem Validierer zur Verfügung.

Jeder Contract muss das Verhalten bei Dissens explizit selbst implementieren, ansonsten verbleibt der Contract in einem Zustand, der von keinem teilnehmenden Knoten weiterverarbeitet wird. Eine denkbare Implementierung für Dissens wäre, die Entscheidung über den tatsächlichen Zustand von einem dritten teilnehmenden Knoten vornehmen zu lassen. Dieser Knoten würde dann sozusagen als Boss oder Richter fungieren.

Ein Contract bildet den Zustand eines konkreten zu verarbeitenden Falls ab. Dies kann der operativen Umsetzung einer Vereinbarung auf höherer Ebene entsprechen. Ein Beispiel für die Vereinbarung auf höherer Ebene ist ein SLA, der festlegt, welcher Sachbearbeiter welche Klasse von Anfragen bearbeitet. Der Contract, der den Bearbeitungsstand einer konkreten Anfrage abbildet würde auf diesen SLA-Contract referenzieren, so dass jeder teilnehmende Knoten überprüfen kann, ob ihm Duties zu Recht zugewiesen wurden.

Durch das Referenzieren anderer Contracts ist es möglich, Protokolle umzusetzen, die als Basis des Contract dienen.

Beispiele können demokratische Abstimmungen sein, Lotto für das Signieren eines Blocks (Bitcoin), Kontoführung…

# Abgrenzung

Sicher sind nur Contracts ohne Quest.

Konsens muss nicht der Wahrheit entsprechen.

Themenspeicher

Dadurch, dass der Code zur Laufzeit kompiliert wird sind Änderung des Codes zur Laufzeit möglich.

Nur so ist es möglich, dass alle Knoten über eine Implementierung der gleichen Laufzeitumgebung verfügen und Konsens bezüglich des Anwendungszustands feststellen können.

1. FLDSMDFR (Franke Literal Dialogic Settlement Mutating Dynamic Foo Replicator)   
   in Anspielung auf den Film "Cloudy with a Chance of Meatballs 2"   
   https://en.wikipedia.org/wiki/Cloudy\_with\_a\_Chance\_of\_Meatballs\_2 [↑](#footnote-ref-2)