Conetex Contract Processing

# Voraussetzungen

Es gibt zwei Voraussetzungen 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.

# Bestandteile eines Contracts

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

Der Zustand des Contracts wird durch Daten-Felder / Member-Variablen direkt unterhalb des Wurzelelements abgebildet. Datenfelder werden als „**Value**“ bezeichnet. Values sind typisiert. Die Contract Processing Language stellt primitive Datentypen zur Verfügung und ermöglicht die Implementierung komplexer Datentypen.

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

Die verschiedenen Rollen der teilnehmenden Knoten werden im XML in Strukturen vom komplexen Typ „**Duty**“ definiert. Von Duty abgeleitete Typen enthalten den Code, der das Verhalten der Rolle implementiert. Dieser Code wird von demjenigen teilnehmenden Knoten ausgeführt, welcher der responsible Participant dieser Duty ist. Der komplexe Typ „Duty“ enthält das Attribut „responsibleParticipant“ vom Typ „Participant“. Entspricht der öffentliche Schlüssel des responsible Participant dem eigenen öffentlichen Schlüssel, so weiß der teilnehmende Knoten, dass er laut Contract die Duty ausführen soll. Dem teilnehmenden Knoten wird also durch Duty seine Rolle im Contract zugewiesen.

# 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 boolesche Datentyp 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 ergebnislosen 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 booleschen Datentyp stehen die booleschen Operationen (Not, And, Or, XOr) zur Verfügung
* die booleschen Operationen können verschachtelt werden
* für den Kontrollfluss stehen „if-else“, sowie die „while“-Schleife zur Verfügung
* komplexe Datentypen sind Kombinationen aus Feldern primitiver oder komplexer Typen
* Instanzen komplexer Datentypen werden explizit durch eine entsprechende Anweisung erzeugt
* Zuweisungen unter Feldern primitiven Typs sind nur möglich, wenn der Basis-Datentyp (Numerisch, String, Boolean) übereinstimmt
* komplexe Datentypen können aus einem Supertyp abgeleitet werden (einfache Vererbung)
* Zuweisungen unter Feldern komplexen 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 komplexen Datentypen definiert werden
* Funktionen können Rückgabewerte haben (Kommando "return")
* gemäß Datentyp können Funktionsaufrufe innerhalb von arithmetischen, Vergleichs- oder booleschen Operationen benutzt werden
* Rekursion ist nicht möglich
* Nebenläufigkeit, Multithreading ist nicht möglich

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.

# Contract Processing Runtime

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 die anderen Teilnehmerknoten geschickt. Dies allerdings vorbehaltlich der vorherigen Validierung des Contracts inklusive der Reproduktion seiner bisherigen Zustände.

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.

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

Höhere Funktionalität mit Zugriff auf Systemressourcen kann nicht zum Sprachumfang der Contract Processing Language gehören. Derartige Funktionalität wird von der Laufzeitumgebung zur Verfügung gestellt. Der Code der Duty fordert diese Funktionalität mit dem Kommando "Quest" an und erhält von der Laufzeitumgebung einen Rückgabewert primitiven Typs. Die Laufzeitumgebung beantwortet 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 Default-Agent beantwortet Quests, 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 der Contract-Validierung zur Verfügung.

# Protokolle auf Basis von Contracts

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 für solche höhere Protokolle sind demokratische Abstimmungen, Kontoführung, Bereitstellung von Diensten wie Persistenz, Einigung auf eine gemeinsame Uhrzeit, das Signieren eines Daten-Blocks durch einen zufälligen Knoten (Proof of Work wie bei Bitcoin)…

Matthias Franke Januar 2018

# Abgrenzung

Sicher sind nur Contracts ohne Quest.

Konsens muss nicht der Wahrheit entsprechen.

# Themenspeicher

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 Entscheidungsträger, Richter fungieren.

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)