gMix: Eine generische Architektur für Mix-Implementierungen und ihre Umsetzung als Open-Source-Framework

Karl-Peter Fuchs, Dominik Herrmann, Hannes Federrath

Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg
{fuchs, herrmann, federrath}@informatik.uni-hamburg.de

Abstract: Mit dem Open-Source-Projekt *gMix*, einem generischen Framework für Mixe, möchten wir die zukünftige Forschung im Bereich der Datenschutzfreundlichen Techniken fördern, indem wir die Entwicklung und Evaluation von Mix-basierten Systemen erleichtern. Das Projekt gMix wird ein umfassendes Code-Repository mit kompatiblen und leicht erweiterbaren Mix-Implementierungen zur Verfügung stellen. Dies ermöglicht den Vergleich verschiedener Mix-Varianten unter einheitlichen Bedingungen und unterstützt durch leicht zugängliche und verständliche Lösungen auch den Einsatz in der Lehre. Wir stellen eine generische Softwarearchitektur für Mix-Implementierungen vor, demonstrieren ihre Anwendbarkeit anhand einer konkreten Implementierung und legen dar, wie wir die Architektur in ein Software-Framework mit einem Plug-in-Mechanismus zur einfachen Komposition und parallelen Entwicklung von Implementierungen überführen wollen.

1 Einleitung

Mixe ermöglichen die anonyme Kommunikation in Vermittlungsnetzen. Das Grundprinzip ist in Abbildung 1 am Beispiel einer sog. *Mix-Kaskade* dargestellt. Seit dem ursprünglichen Vorschlag von David Chaum im Jahr 1981 [Cha81] wurden zahlreiche Mix-Konzepte und -Strategien vorgestellt. Inzwischen wurden Mixe für verschiedene Anwendungsgebiete, z. B. E-Mail [Cha81, Cot95], elektronische Wahlen [Cha81, PIK94, SK95], Location-Based-Services [FJP96] sowie für die Kommunikation mit Echtzeitanforderungen (z. B. ISDN [PPW91], WWW [BFK01, DMS04], DNS [FFHP11]) vorgeschlagen.

In der Praxis haben sich neben den Systemen Mixmaster [Cot95] und Mixminion [DDM03], die den anonymen E-Mail-Versand ermöglichen, bislang lediglich die Anonymisierungssysteme Tor [DMS04], JAP (JonDonym) [BFK01] und I2P etabliert.[1] Durch ihre konstante Fortentwicklung haben diese Implementierungen eine so hohe Komplexität und Spezialisierung auf ihren Anwendungsfall erreicht, dass die wesentlichen Grundfunk-

[1] Die Implementierungen dieser Systeme sind über die jeweiligen Webseiten unter http://sourceforge.net/projects/mixmaster/, http://mixminion.net/, http://www.torproject.org, http://anon.inf.tu-dresden.de/ und http://www.i2p2.de abrufbar.

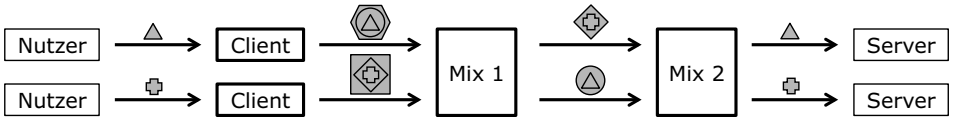

Abbildung 1: Nachrichten werden vom Client schichtweise verschlüsselt. Die Mixe entschlüsseln die Nachricht schrittweise auf dem Weg zum Empfänger. Weitere Details werden in Abschnitt 2 erläutert.

tionen eines Mixes nicht mehr leicht nachvollziehbar sind und ihre Eignung als Ausgangspunkt für die Entwicklung neuer Systeme (z. B. für andere Anwendungsfälle) beschränkt ist. Der JAP-Client besteht beispielsweise derzeit (Dezember 2011) aus über 700 Java-Klassen.[2]

In der Theorie gibt es eine Vielzahl an Vorschlägen, für die keine öffentlich verfügbaren oder gar praktisch einsetzbaren Implementierungen existieren (z. B. [PPW91, DG09, KG10, KEB98, DS03a, Ser07, SDS02, DP04, DS03b] oder [WMS08]). Von diesen Vorschlägen bleibt nur eine verbale oder formale Beschreibung in der jeweiligen akademischen Publikation.

Diese Situation hat drei Konsequenzen: Zum einen wird es Wissenschaftlern unnötig schwer gemacht, bereits publizierte Ergebnisse nachzuvollziehen, da die existierenden Techniken dazu von Grund auf neu implementiert werden müssen. Zweitens ist die Hürde zur Implementierung neuer Mix-Techniken unnötig hoch, da immer wieder von neuem Lösungen für Standardprobleme gefunden und implementiert werden müssen. Und drittens ist es zum aktuellen Zeitpunkt relativ schwierig, die Vorschläge aus unterschiedlichen Veröffentlichungen miteinander zu vergleichen, da diese mit unterschiedlichen Implementierungen, Laufzeitumgebungen und Experimentierumgebungen realisiert wurden.

Mit dem *gMix*-Projekt möchten wir dazu beitragen, diese aus unserer Sicht unerwünschten Konsequenzen zu beheben. Wir verfolgen dabei fünf Ziele:

1. Verfügbarkeit eines umfassenden Code-Repository mit kompatiblen, leicht erweiterbaren Mix-Implementierungen,
2. Vereinfachen der Entwicklung neuer, praktisch nutzbarer Mixe,
3. Evaluation existierender und neuer Mix-Techniken unter einheitlichen Bedingungen,
4. Unterstützung der Lehre durch leicht zugängliche und verständliche Mix-Lösungen sowie
5. Wissenschaftler dazu zu motivieren, ihre Implementierungen auf Basis von *gMix* zu entwickeln und somit an der Weiterentwicklung von *gMix* beizutragen.

Aus diesem Grund haben wir eine *offene, generische Architektur* entworfen, mit der existierende und zukünftige Mixe abgebildet werden können. Die Architektur bildet die

[2] Gezählt wurden alle Klassen, die unter http://anon.inf.tu-dresden.de/develop/doc/jap/ aufgeführt sind.

gemeinsamen Komponenten ab, die für die Entwicklung praktischer Mix-Implementierungen grundsätzlich erforderlich sind. Weiterhin gibt sie den Rahmen für das Verhalten und die Interaktion der Komponenten vor. Basierend auf dieser Architektur haben wir damit begonnen, ein Framework mit einem Plug-in-Mechanismus zu erstellen, das die Auswahl verschiedener Implementierungen einzelner Komponenten (z. B. verschiedene Ausgabestrategien) erlaubt.[3] Das Framework soll es einem Entwickler ermöglichen, einen konkreten Mix aus den verschiedenen Implementierungen zusammenzustellen, ohne den Quelltext des Frameworks anpassen zu müssen. Wir sehen dies als Voraussetzung dafür, verschiedene Implementierungen einfach und unter einheitlichen Bedingungen (z. B. wie in [FFHP11] mit einem Netzwerkemulator oder in einem Forschungsnetzwerk wie dem PlanetLab (http://www.planet-lab.org/) in [BSMG11]) testen zu können, sowie die Abhängigkeiten bei der parallelen Entwicklung verschiedener Implementierungen durch die Open-Source-Community zu minimieren. Neben dem Framework erstellen wir derzeit Referenzimplementierungen für die bedeutendsten Mix-Techniken.

Das gMix-Projekt ist im Internet unter https://svs.informatik.uni-hamburg.de/gmix/ erreichbar. Die Quellcodes sind unter der GPLv3 veröffentlicht.

Der Beitrag ist wie folgt aufgebaut: Im Abschnitt 2 stellen wir die *gMix*-Architektur vor. Anschließend erläutern wir in Abschnitt 3 anhand einer konkreten Implementierung, einem synchron getakteten Kanal-Mix für stromorientierte Kommunikation, wie die Architektur in der Praxis umgesetzt werden kann. Ausgehend von der Architektur und der konkreten Implementierung wird in Abschnitt 4 der geplante Aufbau des *gMix*-Frameworks skizziert. Abschnitt 5 fasst schließlich die Ergebnisse zusammen.

2 gMix-Architektur

Wir verstehen unter einer Softwarearchitektur analog zu [Bal96], eine „strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen“. Eine detaillierte Beschreibung der verschiedenen Architekturebenen und Sichten (Kontextsicht, Struktursicht, Verhaltenssicht, Abbildungssicht) der *gMix*-Architektur findet sich in [Fuc09]. Im Folgenden beschränken wir uns auf eine allgemeine Beschreibung der Architekturkomponenten und deren grundlegender Interaktion. Weiterhin legen wir die zentralen Designentscheidungen dar.

[3] In einem anderen Forschungsgebiet, dem Data-Mining, hat sich die Veröffentlichung eines Frameworks als großer Erfolg herausgestellt. Das Software-Paket WEKA [HFH+09] (http://www.cs.waikato.ac.nz/ml/weka/index.html), das vor fast 20 Jahren veröffentlicht wurde, bietet Zugriff auf eine Vielzahl von Algorithmen und Evaluationstechniken. Es hat die Nutzung von State-of-the-Art-Data-Mining-Techniken erheblich vereinfacht und dazu geführt, dass diese ganz selbstverständlich heute in vielen Anwendungsfeldern eingesetzt werden. Wegen seiner Einfachheit wird WEKA inzwischen auch an vielen Universitäten im Lehrbetrieb eingesetzt.

2.1 Herausforderungen

Die wesentliche Herausforderung bei der Entwicklung einer generischen Architektur für Mix-basierte Systeme besteht darin, diese einerseits generisch genug zu spezifizieren, so dass eine Vielzahl von Mix-Varianten unterstützt wird, sie andererseits aber auch so spezifisch wie möglich zu entwerfen. Hierzu werden die wesentlichen Aufgaben eines Mixes sowie die für eine praktisch nutzbare Implementierung nötigen zusätzlichen und wiederverwendbaren Artefakte (z. B. Serverfunktionalität oder Logging) identifiziert, voneinander abgegrenzt und in Komponenten gekapselt.

Dementsprechend haben wir eine Vielzahl der bislang vorgeschlagenen Mix-Konzepte (insbesondere verschiedene Ausgabe- und Dummy-Traffic-Strategien sowie Umkodierungsschemata) und drei öffentlich verfügbare Implementierungen (Tor, JAP, Mixminion) analysiert. Der Architekturentwurf wurde zusätzlich von der prototypischen Implementierung verschiedener Mix-Komponenten, darunter auch der in Abschnitt 3 beschriebene Mix sowie der von uns in [FFHP11] vorgestellte DNS-Mix, begleitet und iterativ weiterentwickelt.

Die *gMix*-Architektur ermöglicht die parallele bidirektionale anonyme Kommunikation (Vollduplex), für nachrichtenorientierte Anwendungen (z. B. E-Mail) sowie für Datenströme mit Echtzeitanforderungen (Möglichkeit zur Schaltung *langlebiger* anonymer Kanäle, z. B. für WWW-Traffic), unabhängig von der konkreten Realisierung des zugrundeliegenden Kommunikationskanals, des zur Anonymisierung verwendeten Umkodierungsschemas oder der Ausgabestrategie.

2.2 Komponenten

Die wesentlichen Softwarekomponenten sind in Abbildung 2 dargestellt und werden im Folgenden kurz skizziert.

Die Komponenten *MessageProcessor*, *OutputStrategy* und *InputOutputHandler* sind für die Kernfunktion des Mixes (Unverkettbarkeit ein- und ausgehender Nachrichten) zuständig. Die restlichen Komponenten nehmen Hilfsaufgaben wahr.

Die Komponente **MessageProcessor** setzt das Umkodierungsschema des Mixes um. Dieses soll sicherstellen, dass ein- und ausgehende Nachrichten nicht anhand ihrer Bitrepräsentation verkettet werden können. Aufgaben der Komponente sind folglich das Ver- bzw. Entschlüsseln von Mix-Nachrichten sowie die Durchführung einer Integritätsprüfung[4] und Wiederholungserkennung[5]. Für den Entwurf der Komponente wurden die Verfahren nach [Cha81, PPW91, Cot95, DDM03, BFK01, Köp06] und [DMS04] analysiert. Die aktuelle Implementierung ist in Abschnitt 3 beschrieben. Derzeit wird das

[4] Können Nachrichten vom Mix unbemerkt verändert werden, kann die Anonymität der Nutzer aufgehoben werden [Dai00].

[5] Wird ein deterministisches Umkodierungsschema eingesetzt, darf jede Nachricht vom Mix nur ein einziges Mal verarbeitet werden. In diesem Fall muss der Mix wiedereingespielte Nachrichten erkennen und verwerfen, wie beispielsweise in [Köp06] beschrieben.

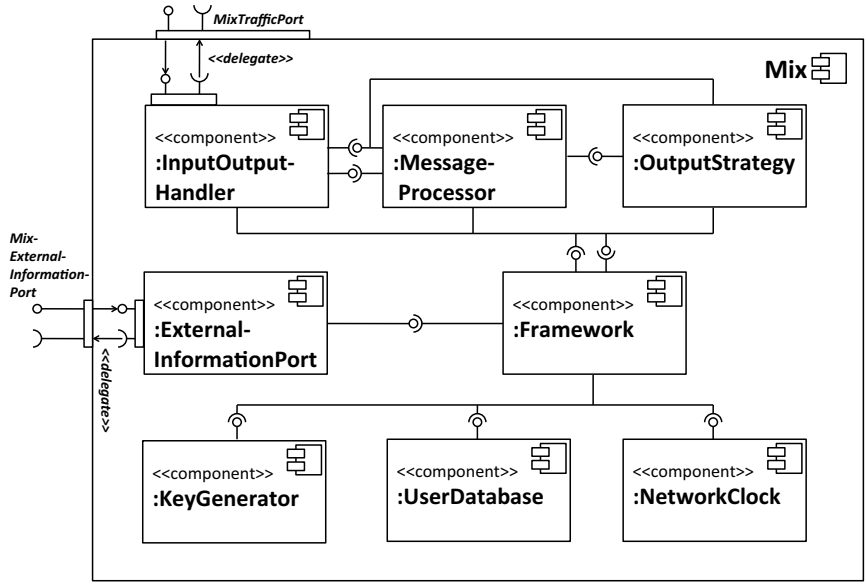

Abbildung 2: Komponentendiagramm

Umkodierungsschema nach [DG09] integriert.

Umkodierte Nachrichten werden vom *MessageProcessor* an die Komponente **Output-Strategy** weitergeleitet, die für die Umsetzung der Ausgabestrategie zuständig ist. Ihre Aufgabe ist es, durch das gezielte Verzögern und die umsortierte Ausgabe eingehender Nachrichten eine Anonymitätsgruppe zu bilden. Ist vorgesehen, dass unter bestimmten Bedingungen bedeutungslose Nachrichten erzeugt oder verworfen werden (Dummy-Traffic-Schema), muss dies ebenfalls in dieser Komponente erfolgen. Realisierungsmöglichkeiten für die Ausgabestrategie und das Dummy-Traffic-Schema sind u. a. in [Cha81, PPW91, Cot95, KEB98, SDS02, DS03b, DP04, Ser07, BL02, DS03a, VHT08, VT08] und [WMS08] beschrieben.

Durch die Komponente *OutputStrategy* verzögerte Nachrichten werden an den **Input-OutputHandler** weitergeleitet. Dieser abstrahiert von den Details des zugrundeliegenden Kommunikationskanals (z. B. einer TCP-Verbindung) und leitet die Nachrichten an die vorgesehenen Empfänger (z. B. andere Mixe oder Server) weiter. Zusätzlich stellt der *InputOutputHandler* mittels einer Schnittstelle über den Kommunikationskanal empfangene Nachrichten für den *MessageProcessor* bereit. Die Übergabe von Nachrichten an den *InputOutputHandler* erfolgt asynchron, das Beziehen von Nachrichten mittels eines Benachrichtigungsprinzips (*wait-notify*). Entsprechend können *InputOutputHandler*, *MessageProcessor* und *OutputStrategy*, abgesehen von der Nachrichtenübergabe, unabhängig voneinander arbeiten. Im Ergebnis können parallel zur Umkodierung weitere Nachrichten empfangen oder versendet werden. Das Umkodieren selbst kann ebenfalls parallelisiert werden.

Im Folgenden wird jeweils kurz auf die Hilfskomponenten *UserDatabase*, *ExternalInformationPort*, *KeyGenerator*, *NetworkClock* und *Framework* eingegangen.

Die **UserDatabase** ermöglicht das Speichern von Daten zu einzelnen Nutzern (z. B. verbundene Clients) des Mixes. Dies umfasst beispielsweise Daten zu geschalteten anonymen Kanälen, wie Kanalkennzeichen [PPW91] und Sitzungsschlüssel. Im einfachsten Fall kann die Realisierung mit einer Hashtabelle erfolgen.

Über den **ExternalInformationPort** können allgemeine Informationen über den Mix veröffentlicht werden, z. B. dessen Public Key, seine unterstützten Protokolle und eine Versionsnummer. Im Falle von JAP würde diese Komponente mit dem *InfoService* [BFK01] kommunizieren.

Der **KeyGenerator** kann zum Erstellen von kryptographischen Sitzungs- oder Langzeitschlüsseln verwendet werden. Da für einige Mix-Verfahren Zeitstempel benötigt werden, kann über die Komponente **NetworkClock** eine synchronisierte Uhr (z. B. über das *Network Time Protocol* nach RFC 5905) realisiert werden.

Neben grundlegenden Funktionen, wie dem Erstellen von Logdateien und dem Laden von Konfigurationsdateien sind in der Komponente **Framework** weitere Funktionen gekapselt. Eine detaillierte Betrachtung erfolgt in Abschnitt 4.

2.3 Grenzen der Generalisierung

Einschränkungen bei der Generalisierung ergeben sich hinsichtlich des Detaillierungsgrades bei der Spezifikation für bestimmte Datenstrukturen und Nachrichtenformate. Das grundlegende Problem ist, dass sich konkrete Implementierungsmöglichkeiten im Detail stark unterscheiden können. Dies macht eine Vereinheitlichung schwer oder gar unmöglich. Betroffen sind das *Nachrichtenformat* sowie die Inhalte der *UserDatabase* und des *ExternalInformationPort*. Ein SG-Mix [KEB98] benötigt beispielsweise exklusiv ein in das Nachrichtenformat integriertes Zeitfenster, um den Ausgabezeitpunkt von Nachrichten einzugrenzen. Sind der Aufbau eines anonymen Kanals und die *eigentliche* Nachrichtenübermittlung wie in [PPW91, DMS04] oder [KG10] getrennt, müssen zusätzlich Informationen wie Kanalkennzeichen und Sitzungsschlüssel (in der *UserDatabase*) gespeichert werden. Kommt ein Umkodierungsschema wie in [Cha81, DDM03] oder [DG09] zum Einsatz, kann hingegen jeder Schlüssel sofort nach dem Umkodieren der zugehörigen Nachricht (hybrides Kryptosystem) verworfen werden.

Eine weitere Konkretisierung der Architektur lässt sich mit dem Ziel, die Architektur so generell wie möglich zu gestalten, nicht vereinbaren. Um dieses Problem zu lösen, verwenden wir abstrakte Datentypen, die nach dem Key-Value-Prinzip arbeiten. Konkrete Werte und Schlüssel werden über eine *Enumeration* spezifiziert. Entsprechend kann für konkrete Implementierungen einer Komponente angegeben werden, welche Werte sie benötigt, d. h. mit welchen *Enumerations* sie kompatibel ist. Da die *Enumeration* nur eine abstrakte Beschreibung benötigter Inhalte ist, kann die konkrete Realisierung unterschiedlich erfolgen. Die Spezifizierung von Abhängigkeiten und das Auflösen von Konflikten sind eine der wesentlichen Aufgaben des Frameworks (vgl. Abschnitt 4).

3 Fallstudie: Implementierung eines synchron getakteten Mixes

Wir stellen in diesem Abschnitt vor, wie der abstrakte Architekturentwurf in einer konkreten Implementierung, einem synchron getakteten Kanal-Mix (d. h. einen Schub-Mix mit konstantem Dummy Traffic aller Teilnehmer) umgesetzt werden kann. Mit dem Mix können duplexfähige anonyme Kanäle zur Übermittlung von Binärdaten geschaltet werden. Von den zu übermittelnden Daten selbst und den in diesen Zusammenhang verwendetet Protokollen wird abstrahiert, um die Implementierung möglichst generisch zu halten.

Dieser Mix besitzt einen hohen Komplexitätsgrad in allen Komponenten (Echtzeitanforderung, verschiedene Nachrichtentypen, Anonymisierung von Datenströmen statt nur Einzelnachrichten). Es gibt unseres Wissens nach noch keine entsprechende Open-Source-Implementierung.

3.1 Umsetzungsdetails

Als Programmiersprache wird wegen seiner Plattformunabhängigkeit und weiten Verbreitung im wissenschaftlichen Bereich und Lehrbetrieb Java eingesetzt. Auf Clientseite kann die Implementierung wie ein *normaler* Socket mittels *InputStream* und *OutputStream* angesprochen werden. Entsprechend ist die Tunnelung anderer Protokolle, z. B. mittels eines HTTP- oder SOCKS-Proxies, sehr einfach realisierbar.

Unsere Implementierung setzt in der Komponente **MessageProcessor** ein zu JAP und den ISDN-Mixen [PPW91] sehr ähnliches Umkodierungsschema um: Nachrichten werden hybrid mit RSA (OAEP-Modus, 2048 bit Schlüssellänge) und AES (OFB-Modus, 128 bit Schlüssellänge) verschlüsselt. Es gibt drei Nachrichtentypen. *Kanalaufbaunachrichten* zum Schalten anonymer Kanäle (Verteilung der Sitzungsschlüssel für AES und Integritätsprüfung mit HMAC-SHA256). Rein symmetrisch verschlüsselte *Kanalnachrichten* zum Übermitteln von Nutzdaten und *Kanalabbaunachrichten* zum Auflösen anonymer Kanäle. Zusätzlich wurde eine Wiederholungserkennung nach dem Vorbild von [Köp06] implementiert. Zum parallelen Umkodieren können mehrere Threads gestartet werden.

Die Komponente **OutputStrategy** wurde als synchroner Schub-Mix implementiert. Entsprechend wird von jedem Teilnehmer eine *Kanalnachricht* (ggf. eine Dummy-Nachricht) erwartet, bevor die Nachrichten gemeinsam und umsortiert ausgegeben werden. Zusätzlich kann eine maximale Wartezeit angegeben werden.

Der **InputOutputHandler** setzt das in Abschnitt 2 beschriebene asynchrone Benachrichtigungsprinzip in einer Steuerungsklasse um. Die Steuerungsklasse kann mittels verschiedener **ConnectionHandler** an konkrete Protokolle (z. B. TCP oder UDP) und Kommunikationsbeziehungen (z. B. *Client-Mix*, oder *Mix-Mix*) angepasst werden. Die aktuelle Implementierung setzt aus Performanzgründen zwischen Mixen und Clients asynchrone Ein- und Ausgabe *(non-blocking I/O)* ein. Zwischen zwei Mixen besteht jeweils nur eine verschlüsselte Verbindung, durch welche die Nachrichten aller Clients getunnelt werden *(multiplexing)*. Es kommt jeweils TCP zum Einsatz.

3.2 Entwicklungs- und Evaluationsumgebung

Verteilte Systeme wie Mixe sind mit herkömmlichen Debugging-Werkzeugen nicht vollständig analysierbar, da Nachrichten im Betrieb mehrere Systeme durchlaufen. Wir haben eine einfache *Testumgebung* entwickelt, die die Fehlersuche erheblich erleichtern kann (Paket *testEnvironment*). Neben der Möglichkeit, die Mixe unter realen Bedingungen, also auf mehreren Systemen verteilt, manuell zu starten, besteht auch die Option, mehrere Mixe lokal auf einem Entwicklungssystem automatisiert zu starten, ohne sich um Kommunikationseinstellungen und die Schlüsselverteilung kümmern zu müssen.

Nachrichten können mit einer eindeutigen ID ausgezeichnet werden, um sie beim Debugging über Systemgrenzen hinweg zu verfolgen. So kann überprüft werden, ob die Nachrichtenübermittlung zwischen Clients und Mixen fehlerfrei funktioniert.

Weiterhin existiert ein einfacher *Last-Generator*, der automatisiert nach einer vorgegebenen Zufallsverteilung Mix-Clients instanziiert und deren Verhalten simuliert. Dies ermöglicht es dem Entwickler, das Verhalten der Mixe mit dynamischen Nutzerzahlen und wechselndem Verkehrsaufkommen zu analysieren. Die aktuelle Implementierung ist auf das Debugging ausgerichtet und hat die Implementierung des synchron getakteten Mixes erheblich erleichtert. Wir arbeiten an einer Erweiterung, die realistische Verkehrsmodelle unterstützt und automatisiert die Verzögerung von Nachrichten in den einzelnen Mix-Komponenten erfassen und visualisieren kann. Wir versprechen uns davon die Identifizierung von Flaschenhälsen und den vereinfachten empirischen Vergleich von verschiedenen Implementierungen.

4 Erweiterung zum gMix-Framework

Die in Abschnitt 2 dargestellte Architektur kann durch die Spezifizierung notwendiger Komponenten, deren Verhalten und Interaktion die Entwicklungszeit für neue praktische Mix-Systeme erheblich verkürzen. Es wäre jedoch darüber hinaus wünschenswert, über ein umfangreiches Code-Repository mit kompatiblen Implementierungen, die *einfach* zu einem konkreten Mix zusammengestellt werden können, zu verfügen. Idealerweise sollte es unterschiedlichen Entwicklern möglich sein, unabhängig voneinander an verschiedenen Implementierungen zu arbeiten und diese ggf. zu veröffentlichen.

Dieses Ziel soll mit Hilfe eines dedizierten Rahmenwerks erreicht werden. Das Framework trennt zwischen veränderbarem Quelltext und gleich bleibenden Strukturen. Im Framework können folglich jeweils mehrere konkrete Implementierungen für die einzelnen Komponenten erstellt und registriert werden, so lange sie den Architekturvorgaben genügen. Die Instanziierung und Steuerung (d. h. die Sicherstellung der Einhaltung des vorgesehenen Kontrollflusses) der gewählten Komponentenimplementierungen erfolgt durch das Framework. Veränderungen am Quelltext des Frameworks durch die Entwickler konkreter Implementierungen sind nicht vorgesehen.

Die zentralen Herausforderungen bei der Entwicklung dieses Frameworks werden im Folgenden zusammen mit unseren geplanten Lösungsansätzen kurz dargestellt.

- Es wird ein *Plug-in-Mechanismus* benötigt, der das Einbinden konkreter Komponentenimplementierungen durch das Framework ermöglicht.
- Es wird ein Mechanismus benötigt, mit dem *Abhängigkeiten* zwischen Komponenten und Nachrichtenformaten abgebildet und erkannt werden können.
- Die einfache Komposition verschiedener Implementierungsvarianten zu einem konkreten Mix soll *ohne Anpassung des Quelltextes* möglich sein.
- Mit Hilfe einer *Versionsverwaltung für verschiedene Implementierungsvarianten* soll die Nachvollziehbarkeit von durchgeführten Evaluationen verbessert werden.
- Die Bereitstellung umfangreicher *Dokumentation und Tutorials* ist erforderlich, um die Hürde zur Nutzung des Frameworks zu senken.

Der *Plug-in-Mechanismus* ist in Abbildung 3 dargestellt. Das Framework enthält Schnittstellendefinitionen für sämtliche (in Abschnitt 2 beschriebene) Komponenten. Zusätzlich verfügt jede Komponente über eine sogenannte *Controller-Klasse*, welche die durch die jeweilige Schnittstelle spezifizierte Funktionalität *nach außen* (d. h. für die anderen Komponenten) zur Verfügung stellt. Intern werden Aufrufe an eine konkrete Implementierung (*Komponentenimplementierung*) weitergeleitet.

Wie aus der Architektur hervorgeht, arbeiten die einzelnen Komponenten nicht völlig autonom, sondern sie interagieren. Jede Implementierung muss daher über Referenzen auf die anderen Komponenten verfügen. Diese Anforderung setzen wir durch einen geeigneten objektorientierten Entwurf und ein dreiphasiges Initialisierungskonzept um. Jede Komponentenimplementierung muss von einer abstrakten Implementierungsklasse *(Implementation)* abgeleitet werden, welche Referenzen auf alle Komponenten enthält. Eine Komponente wird nach dem Laden ihrer Klasse durch den Aufruf ihres Konstruktors instanziiert (Phase 1). Sobald alle Klassen geladen und alle Referenzen gültig sind, wird der Beginn von Phase 2 durch Aufruf einer Initialisierungsmethode bekannt gegeben. In Phase 2 werden Vorbereitungsfunktionen für die Anonymisierung durchgeführt, z. B. können Schlüssel generiert und ausgetauscht werden, Zufallszahlen erzeugt werden oder ggf. die verteilte Netzwerkuhr initialisiert werden. Wenn alle Komponenten ihre Initialisierungsmaßnahmen abgeschlossen haben beginnt Phase 3, der eigentliche Mix-Betrieb, in dem Mix-Nachrichten entgegengenommen und verarbeitet werden.

Die Zusammenstellung der konkreten Komponentenimplementierungen soll nicht im Quelltext „fest verdrahtet“ werden, um Plug-ins entwickeln zu können, ohne den Quelltext des Frameworks anpassen zu müssen. Um diese Anforderung umzusetzen, greifen wir auf einen Klassenlader zurück, der dynamisch zur Laufzeit die konkreten Implementierungen in die *Java Virtual Machine* lädt. Der Klassenlader wertet Kompatibilitätsinformationen, die für jedes Plug-in erfasst werden, aus und verhindert dadurch, dass inkompatible Konfigurationen ausgeführt werden. Die Kompatibilitätsinformationen werden vom Autor des Plug-ins spezifiziert und liegen in einer Property-Datei vor. Darin können auch weitere Laufzeitparameter (z. B. die Schubgröße bei einem Batch-Mix) definiert werden.

Um die Konfiguration eines Mixes zu vereinfachen, wollen wir eine grafische Oberfläche entwickeln, die eine interaktive Auswahl der zu integrierenden Komponenten erlaubt. Die-

se soll die Kompatibilitäts- und Konfigurationsparameter interpretieren, um den Nutzer bei der Auswahl der zueinanderpassenden Implementierungen zu unterstützen. Auf der Projektseite ist der Quelltext des ersten Prototypen des Frameworks veröffentlicht. Neben einer rudimentären Implementierung des Plug-in-Mechanismus sind bereits 14 der in [Cha81, Cot95, KEB98, SDS02, DS03b] und [WMS08] beschriebenen Ausgabestrategien implementiert. Implementierungen für die restlichen Komponenten sind in Arbeit.

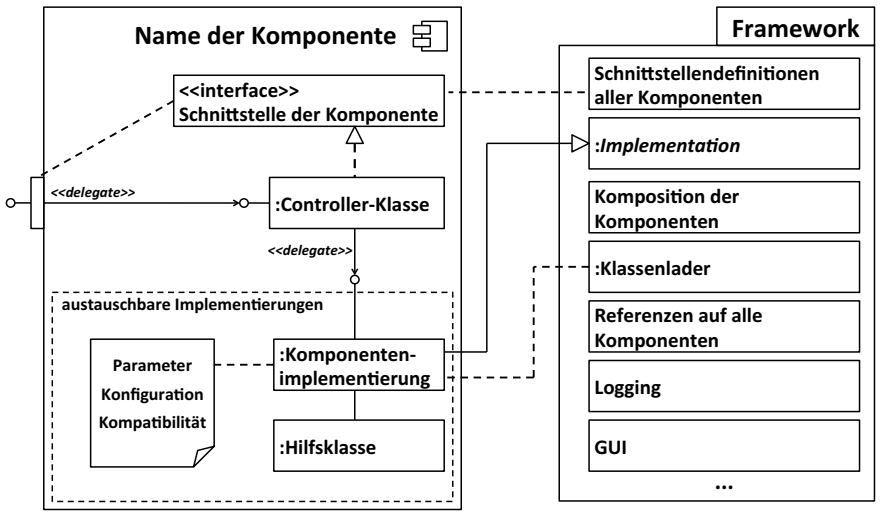

Abbildung 3: Plug-in-Mechanismus

5 Zusammenfassung

In diesem Beitrag haben wir eine generische Mix-Architektur vorgestellt. In der Architektur werden die zentralen Funktionen, die von einer praktischen Mix-Implementierung erbracht werden müssen, in einzelnen Komponenten gekapselt. Jede Komponente erfüllt eine klar abgegrenzte Aufgabe und hat definierte Schnittstellen. Die Architektur reduziert zum einen die Komplexität für Entwickler, zum anderen ist sie die Grundlage für untereinander kompatible Implementierungsvarianten. Am Beispiel des synchron getakteten Kanalmixes wurde die grundsätzliche Anwendbarkeit beispielhaft demonstriert.

Architektur und Implementierung sind öffentlich zugänglich und erweiterbar. Weiterhin wurde das geplante *gMix*-Framework, an dem wir derzeit arbeiten, vorgestellt. Es wird eine weitgehend unabhängige Entwicklung dedizierter Mix-Plug-ins und die Evaluation unter einheitlichen Umgebungsbedingungen ermöglichen. Wir hoffen, dass durch das *gMix*-Projekt nicht nur die Entwicklung Datenschutzfreundlicher Techniken vorangetrieben, sondern vor allem auch deren Überführung in die Praxis begünstigt wird.

Literatur

[Bal96] Helmut Balzert. *Lehrbuch der Software-Technik.: Software-Entwicklung.* Lehrbuch der Software-Technik. Spektrum, Akad. Verl., 1996.

[BFK01] Oliver Berthold, Hannes Federrath und Stefan Köpsell. Web MIXes: a system for anonymous and unobservable Internet access. In *International workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, Seiten 115–129, New York, USA, 2001. Springer-Verlag.

[BL02] Oliver Berthold und Heinrich Langos. Dummy Traffic against Long Term Intersection Attacks. In Roger Dingledine und Paul F. Syverson, Hrsg., *Privacy Enhancing Technologies*, Jgg. 2482 of *Lecture Notes in Computer Science*, Seiten 110–128. Springer, 2002.

[BSMG11] Kevin Bauer, Micah Sherr, Damon McCoy und Dirk Grunwald. ExperimenTor: A Testbed for Safe Realistic Tor Experimentation. In *Proceedings of Workshop on Cyber Security Experimentation and Test (CSET)*, 2011.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[Cot95] Lance Cottrell. Mixmaster and remailer attacks. In *http://www.obscura.com/loki/remailer-essay.html*, 1995.

[Dai00] Wei Dai. Two attacks against freedom. In *http://www.weidai.com/freedom-attacks.txt*, 2000.

[DCJW04] Yves Deswarte, Frédéric Cuppens, Sushil Jajodia und Lingyu Wang, Hrsg. *Information Security Management, Education and Privacy, IFIP 18th World Computer Congress, TC11 19th International Information Security Workshops, 22-27 August 2004, Toulouse, France*. Kluwer, 2004.

[DDM03] George Danezis, Roger Dingledine und Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *IEEE Symposium on Security and Privacy*, Seiten 2–15. IEEE Computer Society, 2003.

[DG09] George Danezis und Ian Goldberg. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE Symposium on Security and Privacy*, Seiten 269–282. IEEE Computer Society, 2009.

[DMS04] Roger Dingledine, Nick Mathewson und Paul Syverson. Tor: The Second-Generation Onion Router. In *In Proceedings of the 13th USENIX Security Symposium*, Seiten 303–320, 2004.

[DP04] Claudia Díaz und Bart Preneel. Taxonomy of Mixes and Dummy Traffic. In Deswarte et al. [DCJW04], Seiten 215–230.

[DS03a] George Danezis und Len Sassaman. Heartbeat traffic to counter (n-1) attacks: red-green-black mixes. In Sushil Jajodia, Pierangela Samarati und Paul F. Syverson, Hrsg., *WPES*, Seiten 89–93. ACM, 2003.

[DS03b] Claudia Díaz und Andrei Serjantov. Generalising Mixes. In Roger Dingledine, Hrsg., *Privacy Enhancing Technologies*, Jgg. 2760 of *Lecture Notes in Computer Science*, Seiten 18–31. Springer, 2003.

[FFHP11] Hannes Federrath, Karl-Peter Fuchs, Dominik Herrmann und Christopher Piosecny. Privacy-Preserving DNS: Analysis of Broadcast, Range Queries and Mix-Based Protection Methods. In Vijay Atluri und Claudia Díaz, Hrsg., *ESORICS*, Jgg. 6879 of *Lecture Notes in Computer Science*, Seiten 665–683. Springer, 2011.

[FJP96] Hannes Federrath, Anja Jerichow und Andreas Pfitzmann. MIXes in Mobile Communication Systems: Location Management with Privacy. In Ross J. Anderson, Hrsg., *Information Hiding*, Jgg. 1174 of *Lecture Notes in Computer Science*, Seiten 121–135. Springer, 1996.

[Fuc09] Karl-Peter Fuchs. Entwicklung einer Softwarearchitektur für anonymisierende Mixe und Implementierung einer konkreten Mix-Variante. Magisterarbeit, Universität Regensburg, 2009.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann und Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[KEB98] Dogan Kesdogan, Jan Egner und Roland Büschkes. Stop-and-Go-MIXes Providing Probabilistic Anonymity in an Open System. In David Aucsmith, Hrsg., *Information Hiding*, Jgg. 1525 of *Lecture Notes in Computer Science*, Seiten 83–98. Springer, 1998.

[KG10] Aniket Kate und Ian Goldberg. Using Sphinx to Improve Onion Routing Circuit Construction. In Radu Sion, Hrsg., *Financial Cryptography*, Jgg. 6052 of *Lecture Notes in Computer Science*, Seiten 359–366. Springer, 2010.

[Köp06] Stefan Köpsell. Vergleich der Verfahren zur Verhinderung von Replay-Angriffen der Anonymisierungsdienste AN.ON und Tor. In Jana Dittmann, Hrsg., *Sicherheit*, Jgg. 77 of *LNI*, Seiten 183–187. GI, 2006.

[PIK94] Choonsik Park, Kazutomo Itoh und Kaoru Kurosawa. Efficient anonymous channel and all/nothing election scheme. In *Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, EUROCRYPT '93, Seiten 248–259, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[PPW91] Andreas Pfitzmann, Birgit Pfitzmann und Michael Waidner. ISDN-MIXes: Untraceable Communication with Small Bandwidth Overhead. In Wolfgang Effelsberg, Hans Werner Meuer und Günter Müller, Hrsg., *Kommunikation in Verteilten Systemen*, Jgg. 267 of *Informatik-Fachberichte*, Seiten 451–463. Springer, 1991.

[SDS02] Andrei Serjantov, Roger Dingledine und Paul F. Syverson. From a Trickle to a Flood: Active Attacks on Several Mix Types. In Fabien A. P. Petitcolas, Hrsg., *Information Hiding*, Jgg. 2578 of *Lecture Notes in Computer Science*, Seiten 36–52. Springer, 2002.

[Ser07] Andrei Serjantov. A Fresh Look at the Generalised Mix Framework. In Nikita Borisov und Philippe Golle, Hrsg., *Privacy Enhancing Technologies*, Jgg. 4776 of *Lecture Notes in Computer Science*, Seiten 17–29. Springer, 2007.

[SK95] Kazue Sako und Joe Kilian. Receipt-Free Mix-Type Voting Scheme - A Practical Solution to the Implementation of a Voting Booth. In *EUROCRYPT*, Seiten 393–403, 1995.

[VHT08] Parvathinathan Venkitasubramaniam, Ting He und Lang Tong. Anonymous Networking Amidst Eavesdroppers. *IEEE Transactions on Information Theory*, 54(6):2770–2784, 2008.

[VT08] Parvathinathan Venkitasubramaniam und Lang Tong. Anonymous Networking with Minimum Latency in Multihop Networks. In *IEEE Symposium on Security and Privacy*, Seiten 18–32. IEEE Computer Society, 2008.

[WMS08] Wei Wang, Mehul Motani und Vikram Srinivasan. Dependent link padding algorithms for low latency anonymity systems. In Peng Ning, Paul F. Syverson und Somesh Jha, Hrsg., *ACM Conference on Computer and Communications Security*, Seiten 323–332. ACM, 2008.