Private Information Retrieval

Ida Hönigmann

9. April 2020

Zusammenfassung

1 Idee

In diesem Projekt wird ein System entwickelt bei dem Clients Nachrichten von verschiedenen Servern abfragen können. Die Besonderheit liegt in der Sicherheit, die dem Client gewährleistet wird: keiner der Server weiß welche Nachricht der Client lesen möchte.

Eine Möglichkeit diese Sicherheit umzusetzen ist es den Client alle Nachrichten abfragen zu lassen. Dieses Projekt wählt allerdings einen anderen Ansatz, der eine geringere Menge an Daten, die übertragen werden müssen ermöglicht. Der Nachteil ist, dass ein Client jeweils Anfragen an zwei Server stellen muss. Daher müssen zumindest zwei Server existieren, um Nachrichten empfangen zu können.

2 Anwendungsfälle

Dieses System ist für alle Systeme der Form von Servern, die Clients Nachrichten anbieten anwendbar.

Denkbar wäre zum Beispiel eine solche Lösung bei Daten zu Krankheiten. Wenn ein Benutzer nach einer Krankheit sucht, von der er weiß oder befürchtet diese zu haben, möchte er eventuell nicht, dass der Betreiber des Servers Kenntnis davon hat. Das gilt jedoch auch für Personen, die die Krankheit nicht haben und Informationen darüber erhalten wollen. Es könnte angenommen werden, dass sie die Krankheit besitzen, da sie ja danach gesucht haben.

Ein zweiter Anwendungsfall wäre ein Online-Nachrichtendienst. Oft wird nicht dediziert das Verhalten der Nutzer analysiert, sondern nur Logdaten gesammelt um Analysen über die Auslastung der Systeme zu erstellen oder falls ein Problem auftritt dieses nachverfolgen zu können. Jedoch kann es vorkommen, dass eine andere Organisation diese Logdaten anfordert um Informationen über einen bestimmten Benutzer oder eine Benutzergruppe erlangen zu können. Dazu muss die Organisation, die die Daten anfordert in irgendeiner Form über mehr

a	b	a xor b
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 1: Wahrheitstafel der xor-Operation

Macht verfügen. Dies ist zum Beispiel bei Regierungen, auch anderer Länder, der Fall.

In diesen beiden und vielen anderen Fällen kann eine Implementierung eines *private information retrieval* die Benutzer schützen.

3 Funktionsweise

Die in diesem Projekt implementierte Version von *private information retrieval* nutzt die Eigenschaften der Exklusiv-Oder-Operation (xor).

3.1 Exlusiv-Oder-Operation (xor)

Die xor Operation erhält, gleich der "und"- (Englisch "and") und der "oder"- (Englisch "or") Operation zwei Signale als Eingang und liefert ein Signal als Ausgang. Dabei kann man das Verhalten von xor als "entweder oder" beschreiben. Die Wahrheitstafel in Tabelle 1 definiert das Verhalten von xor genauer.

Die Eigenschaft, die dieses Projekt benötigt ist folgende: Jede Folge von Bytes a mit einer anderen Folge von Bytes b nach Anwendung der xor-Operation ergibt eine Folge von Bytes c für die gilt, dass c xor $a = b$ und c xor $b = a$.

3.2 Inhalt der gesendeten Nachrichten

Ein Client baut zwei Verbindungen zu unterschiedlichen Servern auf. Statt nur die gewünschte Nachricht anzufordern, verlangt er das Produkt von xor-Operationen vieler Nachrichten.

Welche Nachrichten der Client anfordert, die Reihenfolge der Nachrichten, sowie die Anzahl sind irrelevant, solange beide Server die fast die gleichen Nachrichten verwenden. Der einzige Unterschied darf darin liegen, dass in einer der beiden Anforderungen die gewünschte Nachricht enthalten ist, und in der anderen Anforderung nicht.

Beide Server schicken dann das geforderte Produkt der Nachrichten zurück.

3.3 Kalkulation der gewünschten Nachricht

Der Client muss nun ebenfalls die xor-Operation auf die beiden erlangten Antworten der Server anwenden. Das Ergebnis ist die gewünschte Nachricht.

4 Möglichkeiten einer Untergrabung

5 Implementation

5.1 Command Line Interface

Das Command Line Interface wurde mit CLI11 umgesetzt.

Um Informationen über die Verwendung der beiden Programme *client* und *server* zu erhalten kann die Option $--help$ (oder als Kurzform $-h$) spezifiziert werden.

Sonstige Optionen, die bei *client* spezifiziert werden können sind:

- $--list$ / $-l$: gibt eine Liste aller Nachrichten Titel und deren Index aus
- $--index$ / $-i$: gibt die Nachricht des Index aus
- $--port1$ / $-p$: Port des ersten Server an dem die Nachrichten angefordert werden sollen
- $--server1$ / $-s$: IP Adresse des ersten Server an dem die Nachrichten angefordert werden sollen
- $--port2$ / $-q$: Port des zweiten Server an dem die Nachrichten angefordert werden sollen
- $--server2$ / $-t$: IP Adresse des zweiten Server an dem die Nachrichten angefordert werden sollen
- $--verbose$ / $-v$: gibt zusätzliche Debug Meldungen aus

Die Optionen bei *server* sind folgende:

- $--port$ / $-p$: Port, an dem die Verbindung aufgebaut werden soll
- $--verbose$ / $-v$: gibt Debug Meldungen aus
- $--loop$ / $-l$: lässt den Server in einer Schleife laufen
- $--file$ / $-f$: Datei, in der die Nachrichten gespeichert sind
- $--size$ / $-s$: maximale Länge einer Nachricht

5.2 Aufbau und Speicher der Daten

Da sich dieses Projekt auf die sichere Kommunikation fokussiert werden die Daten einfachheitshalber in einem Textfile gespeichert. Die Datei *data.txt* wird als Datenspeicher verwendet. Der Aufbau der Daten ist abwechselnd der Titel der Nachricht und dann die eigentliche Nachricht. Da in dem Text der Daten prinzipiell alle Zeichen vorkommen können wird als Trennzeichen ein Zeilenumbruch (\n) verwendet.

Das Datenfile wird ausgelesen nachdem der Server gestartet wurde.

5.3 Byte-Operationen

Um xor auf zwei Strings anwenden zu können wurde ein entsprechender Code in der Funktion *xor_string* implementiert.

```
1  /* Perform xor on every char and return a new output char[]
2  * */
3  char* xor_string(const char a[], const char b[], unsigned int len=281) {
4      char* output = new char[len];
5
6      for (int i{}; i < len - 1; i++) {
7          char tmp = a[i] ^ b[i];
8          output[i] = tmp;
9      }
10     output[len - 1] = '\0';
11     return output;
12 }
```

Da die zwei Array-von-Character-Parameter implizit auf Pointer des jeweils ersten Elements geändert werden, muss die Länge der Arrays als Parameter mitgegeben werden.

Die xor-Operation zweier Strings kann gelöst werden, indem jedes Zeichen (jeder Character) des Strings mit dem dazugehörigen Zeichen des anderen Strings xor verknüpft wird. Wenn das für alle Zeichen gemacht wird erhält man, durch zusammenfügen zu einem neuen String, das Ergebnis.

Zwei Character in C++ xor zu verknüpfen ist relativ einfach, da ein Character immer gleich lange ist (meist 8 Bit = 1 Byte). In Zeile 7 wird diese xor Verknüpfung vorgenommen. Nachdem alle Zeichen xor-verknüpft sind (also Zeile 7 *len - 1* Mal durchgeführt wurde) wird zuletzt noch ein \0 hinzugefügt. Dieses markiert für gewöhnlich das Ende eines Strings.

Die Funktion liefert einen Pointer auf ein Characterarray zurück, da alle anderen Rückgabetypen, die normalerweise in solch einer Situation verwendet werden könnten, Probleme verursacht haben. Das Problem, das bei einem xor von zwei gleichen Zeichen auftritt ist, dass das Ergebnis \0 ergibt. Das wird in Abbildung 2 gezeigt. Bei allen Stringoperationen wird allerdings das \0 als Ende angesehen, daher musste direkt in einigen Bereichen direkt auf den Speicher zugegriffen werden.

Eine Bedingung um die *xor_string* Funktion anwenden zu können ist, dass beide Strings gleich lange sein müssen. Selbst wenn die beiden Strings als Character-Array mit der gleichen Größe angelegt werden gibt es Probleme. Um effizient zu sein, wird beim Initialisieren eines Character-Arrays nur der Teil beschrieben, der auch verwendet werden muss. So kann zum Beispiel folgender Code `char* char_ptr = new char[10]; strcpy(char_ptr, "Hello!");` das Bitmuster in Abbildung 3 ergeben.

Wenn nun zwei Strings mit einer unterschiedlichen Anzahl an Zeichen an die *xor_string* Funktion übergeben werden, entsteht nach dem letzten Zeichen des

Bitmuster	Bedeutung (ASCII)
01100001	a
01100001	a
00000000	\0

Tabelle 2: Zwei gleiche Zeichen (a) miteinander xor verknüpft ergibt das Bitmuster 00000000, welches \0 bedeutet. Dieses Symbol steht normalerweise nur am Ende eines Strings.

Bitmuster	Bedeutung (ASCII)
01001000	H
01100101	e
01101100	l
01101100	l
01101111	o
00100001	!
00000000	\0
00001001	TAB
00000000	\0
01000001	A

Tabelle 3: Das Bitmuster, das nach Ausführung des Codes `char* char_ptr = new char[10]; strcpy(char_ptr, "Hello!");` beispielsweise im Speicher stehen kann.

kürzeren Texts eine zufällige Zeichenfolge und die längere Nachricht kann vom Client nicht mehr gelesen werden.

Um das zu vermeiden füllt die Funktion *cleanup_char_arr* die Zeichen nach dem \0 mit Leerzeichen auf. Um den resultierenden String weiter wie jeden andren String behandeln zu können wird zuletzt das \0 auf der letzten Position gesetzt.

```
/* Sets all chars after (and including) the \0 to spaces
* to enable all xor operations to be performed smoothly;
* (and sets the \0 back into the last char)
* */
void cleanup_char_arr(char a[], unsigned int len=281) {
    std::fill(a + strlen(a), a + len - 1, ' ');
    a[len - 1] = '\0';
}
```

5.4 Zufällige Nachrichten anfordern

Um die Sicherheit des Client zu garantieren müssen zufällige Nachrichten angefordert werden. Weiters muss die Anzahl der Nachrichten zufällig sein, da sonst ein Server feststellen könnte, das eine der von ihm verlangten Nachrichten, die "richtige" ist. Zusätzlich müssen die Nachrichten in einer zufälligen Reihenfolge angefordert werden. Wenn zum Beispiel die "richtige" Nachricht immer die erste wäre, hätte der Server ein leichtes Spiel.

Um all diese Anforderungen umzusetzten wird in diesem Projekt die *random*-Bibliothek verwendet.

```
/* setup number distributions */

random_device seeder;
mt19937 rndm_engine{seeder()};

uniform_int_distribution<int> dist_idx(0, stoi(cnt_messages) - 1);
uniform_int_distribution<int> dist_cnt(5, 10);

int cnt{dist_cnt(rndm_engine)}; /* number of messages to request */
for (int i{}; i < cnt; i++) {
    int num{message_idx};
    while (find(req_idx.begin(), req_idx.end(), num) != req_idx.end()) {
        num = dist_idx(rndm_engine);
    }
    req_idx.push_back(num);  /* message idx to request */
}
```

Dieser Code erfüllt die ersten beiden Anforderungen (zufällige Anzahl an

zufälligen Nachrichten). Zuerst wird, in Zeile 9, die Anzahl der Nachrichten festgelegt. Diese ann einen Wert zwischen 5 und 10 annehmen. In der darauffolgenden Schleife werden so lange zufällige Zahlen generiert, bis diese noch nicht in *req_idx*, einem Vector der die Nachrichten Indizes speichert, enthalten sind. Dabei kann jede Zahl zwischen 0 und der Anzahl der Nachrichten weniger Eins hinzugefügt werden.

Das zufällige Umordnen des Vectors wird mit folgender Zeile Code erreicht:

```
1 /* randomize the oder of the requested messages */
2
3 shuffle(begin(idx_), end(idx_), rndm_engine);
```

Dabei steht *idx_* für eine Kopie von *req_idx*. Damit wird der originale Vektor nicht verändert.

5.5 Kommunikation

Die Kommunikation zwischen allen beteiligten Einheiten wird über asio geregelt. Asio ermöglicht eine einfache Kommunikation über Streams.

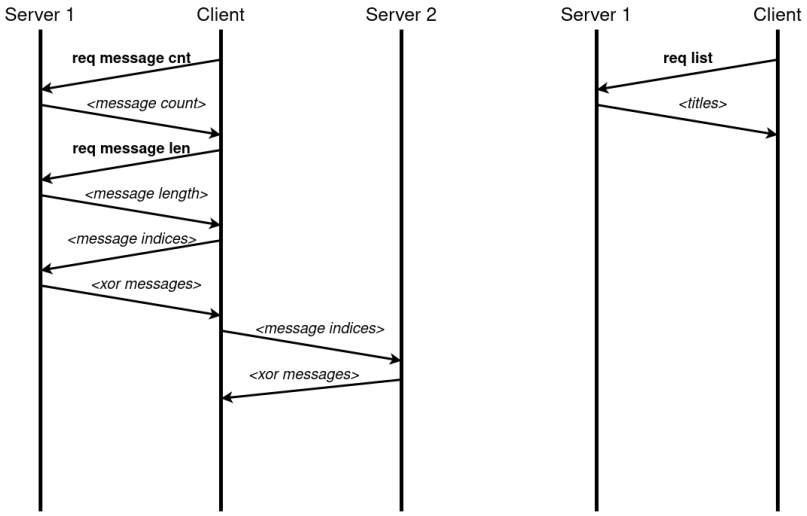
Die zwei typischen Kommunikationsabläufe werden in Abbildung 1 gezeigt.

Der einzige Codeteil, bei dem der asio-stream nicht wie gewohnt, um Strings einzulesen und zu versenden, verwendet wird ist bei der Übermittlung des eigentlichen Dateninhalts. Da bei den Daten, wie in Abschnitt 5.3 beschrieben, das Zeichen \0 vorkommen kann, können hier nicht gewöhnliche Stringfunktionen verwendet werden. Stattdessen wird auf Senderseite folgender Code verwendet:

```
1 /* send data */
2
3 spdlog::debug("sending {}", data);
4 strm.write(output, message_len + 1);
```

Auf der Empfängerseite sieht das Gegenstück folgendermaßen aus:

```
1  /* receive data */
2
3  string data;
4  char* buffer = new char[message_len + 1];
5
6  /* write messages into char buffer */
7
8  while (strm.read(buffer, sizeof(buffer))) {
9      data.append(buffer, sizeof(buffer));
10 }
11
12 /* move buffer into string */
13
```



(a) Eine Nachricht wird abgefragt. Dazu muss der Client zuerst die Anzahl der Nachrichten, sowie deren Länge wissen. Dann fragt er beide Server nach bestimmten Nachrichten, die dann xor-verknüpft zurückgesendet werden.

(b) Client fragt die Nachrichtentitel ab.

Abbildung 1: Die zwei gewöhnlichen Kommunikationsabläufe.

```
14  data.append(buffer, strm.gcount());
15
16  delete[] buffer;
17  spdlog::debug("received data");
```

Diese beiden Codeabschnitte ermöglichen es, statt wie normalerweise Daten, deren Ende durch ein \n oder \0 gekennzeichnet sind, eine fixierte Länge an Bytes zu schreiben und auszulesen. Die Anzahl an geschriebenen und ausgelesenen Bytes ist durch die Nachrichtenlänge (*message_len*) definiert. Diese wird um Eins erhöht, da die Nachricht durch ein \0 beendet ist.

Um die erhaltenen Daten zu speichern, werden diese in einen Vector angehangen. Da die Daten später als Characterarray vorliegen müssen, werden sie hier konvertiert. Das stellt auch sicher, dass keine Probleme mit \0 auftreten.

```
1  /* copy messages viy memcpy to avoid \0 stopping the copy process */
2
3  char* answer = new char[message_len + 1];
4  memcpy(answer, data.c_str(), message_len + 1);
5  responses.push_back(answer);
```

Um die Daten ohne Verlust von allem, das hinter dem ersten \0 steht in ein Characterarray kopieren zu können, muss die Funktion *memcpy* verwendet werden. Diese kopiert die Bytewerte von Beginn des *data.c_str()* Pointers auf den Speicherbereich des *answer* Pointers für *message_len* + 1 Bytes.