NVS PS 2016: Programmieraufgabe

Thomas Samy Dafir, Dominik Baumgartner, Vivien Wallner

Inhaltsverzeichnis

1 Aufgabe 1

Erstellen zweier Programme, von denen eines als Sender, das andere als Empfänger agiert. Der Sender soll mittels UDP Pakete an den Empfänger senden. Dabei sollen die ersten vier Byte jeder Nachricht eine fortlaufende Nummer enthalten.

2 Dokumentation (Java-Version), Aufgabe 1

Dokumentation: siehe auch kommentierten Sourcecode

2.1 Funktion allgemein

Der Sender erstellt einen Datagram-Socket, dessen Adresse nicht bekannt sein muss, da der Sender nur sendet und keine Antwort erhält (es werden also keine Nachrichten an den Sender gesendet). Der Sendevorgang geht folgendermaßen vonstatten: Die Nachricht liegt als String vor. Sie wird in ein Byte-Array konvertiert, indem der ASCII-Wert jedes Zeichens des String als Byte repräsentiert wird. Aus diesem Array und der Empfänger-Info (IP, Port) wird dann ein UDP-Paket erstellt. Dieses wird dann dem Socket übergeben und versendet. Der Empfänger erstellt einen Socket, um UDP-Pakete empfangen zu können. Die Port-Nummer des Socket muss hier jedoch bekannt sein, da der Sender Pakete an die Kombination aus IP-Adresse (in unserem Fall 127.0.0.1) und Port-Nummer sendet. Da die Nachricht, die ein UDP-Paket enthält eine Abfolge von Bytes ist, muss ein Buffer-Byte-Array erstellt werden, um empfangene Daten zu speichern. Empfangsvorgang: Ein UDP-Paket kommt am Socket an und wird mittels *receive* gespeichert. Die Nachricht kann dann mittels *getData* extrahiert werden. Ergebnis ist ein Array von Byte Werten. Dieses muss nun interpretiert werden, da eine Byte-Folge je nach Kodierung verschiedene Bedeutungen haben kann. Da wir wissen, dass es sich um eine Nachricht handelt, die ursprünglich als String von ASCII-Zeichen vorlag, können wir nun aus diesen Bytes einen String bauen und erhalten damit die gesendete Nachricht im lesbaren Format.

2.2 Empfänger

Generell müssen im Empfänger folgende Schritte ausgeführt werden:

1. Erstelle Datagram-Socket.
2. Erstelle Byte-Array als Buffer.

3. Erstelle Datagram-Paket Variable unter Verwendung des Buffers.

4. Empfange über den Socket Pakete.

2.2.1 Socket erstellen

Ein Datagram Socket wird in Java durch ein Objekt des Type *DatagramSocket* repräsentiert. Dieses Objekt wird unter Angabe einer Port-Nummer erstellt. Im Falle des Empfängers muss diese bekannt sein, da der Sender Pakete an diesen Port sendet.

2.2.2 Byte-Array und Datagram-Paket erstellen

Nachrichten, die als UDP-Pakete versendet werden, werden durch eine Abfolge von Bytes (einem Byte-Array) repräsentiert. Jedes Paket enthält also ein Byte-Array mit der Nachricht in Bytes. Dieses Array wird jetzt verwendet, um eine Paket-Variable zu definieren, der jedes empfangene Paket zugewiesen wird.

2.2.3 Pakete empfangen und zählen

Um eine unbekannte Anzahl an Paketen zu empfangen, wird *socket.receive* in einer Schleife verwendet. *receive* ist eine "blockingMethode, die die Ausführung des aktuellen Thread so lange blockiert, bis ein Paket empfangen wird. Um diese Endlosschleife wieder verlassen zu können, wird ein Timeout für den Socket erstellt. Unter normalen Umständen wird der Empfangsvorgang nach dem Erhalt des letzten Pakets beendet. Werden nicht alle Pakete empfangen, übernimmt der Timeout das Beenden der Empfangsvorgangs.
Dieser veranlasst den Socket, eine *SocketTimeoutException* zu werfen, sobald die festgelegte Zeit verstrichen ist, ohne dass ein Paket empfangen wurde. Diese Exception wird in einem *catch* nach der Schleife aufgefangen, danach an das Ende der Schleife zurückgesprungen und nur noch der Socket geschlossen. Auerdem werden in der Schleife de empfangenen Pakete gezählt.

2.3 Sender

Um UDP-Pakete zu senden, müssen folgende Schritte ausgeführt werden.

1. Erstelle Datagram-Socket.

2. Erstelle Byte-Array als Buffer.

3. Erstelle ein InetAddress Object, das die Empfänger-IP repräsentiert.

4. Erstelle zu sendende Nachricht.

5. Erstelle und sende Pakete.

2.3.1 Socket erstellen

Anders als im Empfänger wird der Socket hier unter Verwendung des leeren Standard-Konstruktors erstellt. Damit wird der Socket keiner bestimmten, sondern dem erst-freien Port zugewiesen. Die Port-Nummer ist damit nicht bekannt, wird jedoch nicht gebraucht, da vom Empfänger keine Antwort erwartet wird.

2.3.2 Byte-Array erstellen

Auch hier wird ein Byte-Array erstellt, das später die zu sendende Nachricht enthalten und zur Erstellung der Datagram-Pakete benötigt wird.

2.3.3 InetAdress Objekt erstellen

Um ein Paket an den Empfänger zu senden, wird natürlich dessen IP-Adresse benötigt. Diese liegt hier als Hostname oder IP-Adresse im String-Format vor. Dieser String wir nun mittels $getByName$ zu einer $InetAddress$ konvertiert, um später für das Erstellen der Pakets verwendet werden zu können.

2.3.4 Nachricht erstellen

Eine Nachricht wird unter Verwendung des aktuellen Schleifen-Index erstellt. Dieser wird einfach als Integer ohne Formatierung verwendet. An diesen String wird zusätzlich noch die vorgegebene Anzahl an Zeichen, die mit der Methode $getPayload$ erstellt wird angehängt.

2.3.5 Pakete erstellen und senden

Nun muss die Nachricht in ein Paket gepackt werden. Dazu wird zuerst der String in ein Byte-Array konvertiert und danach mit diesem Array ein Paket erstellt. Anders als beim Empfänger müssen hier zusätzlich Empfänger-Adresse und -Port angegeben werden, da das Paket hier nicht nur als Empfänger dient. Weiters werden beim Senden keine Adresse und Port angegeben, sämtliche Information ist im Paket gespeichert. Das erstellte Paket kann nun unter der Verwendung von *socket.send* verschickt werden. Dieser Vorgang wird wiederholt, bis der Sendevorgang beendet ist und keine Pakete mehr gesendet werden müssen.

2.4 Zeitmessung und zusätzliche Parameter

Zeitmessung erfolgt mit zwei long-Variablen, denen mit $System.currentTimeMillis$ die Zeitpunkte vor und nach dem Senden/Empfangen zugewiesen werden. Die Differenzzeit ist dann die Zeit, die der Sende-/Empfangsvorgang benötigt. Zusätzlich kann mittels cmd-Argumenten die Länge der im Sender erstellten Nachricht eingestellt werden. Die Nachricht wird dann mit der *getPayload*-Methode erstellt, die einen String gegebener Länge erzeugt. Aus Länge * Paket-Anzahl ergibt sich die gesendete Datenmenge, mittels Division durch die Sendezeit erhalten wir die Übertragungsgeschwindigkeit. Diese Auswertung erfolgt mit der *evaluate*-Methode.

3 Dokumentation (C-Version), Aufgabe 1

Dokumentation: siehe auch kommentierten Sourcecode

3.1 Funktion allgemein

Aufgrund der in C nicht vorhandenen Objektorientierung gestaltet sich die Implementierung komplizierter als in Java. Anstatt praktischer vorhandener Klassen werden hier in Libraries vordefinierte Struct-Konstrukte in Zusammenhang mit Library-definierten Funktionen verwendet. Im Sender müssen zuerst ein Socket und ein Adressen-Konstrukt, welches die Empfängerdaten enthält erstellt werden. Außerdem muss noch ein Character-Array erstellt werden, das die zu sendenden Nachrichten enthalten wird. Danach werden Nachrichten mit fortlaufender Nummer erstellt und im Buffer gespeichert um dann unter Angabe des Adressen-Konstrukts und des Sockets versendet zu werden. Der Empfänger erstellt ebenfalls einen Socket und ein Adressen-Konstrukt, das unter anderem den die Port-Nummer enthält, auf der der Socket erstellt werden soll. Adresse und Socket sind hier 2 separate Objekte. Unter Verwendung von *bind* werden dem Socket die im Adressen-Objekt gespeicherten Eigenschaften wie Port-Nummer zugewiesen. Danach können unter Verwendung von $recvfrom$ in einer Schleife viele Pakete empfangen werden. Um die "blockingFunktion $recvfrom$ wieder verlassen zu können, wird die Schleife terminiert, sobald alle Pakete erhalten wurden. Da auch ein Fall auftreten kann, in dem nicht alle Pakete erhalten werden, wird für den Socket vorsichtshalber ein Timeout gesetzt. Werden für die Dauer des Timeout keine Pakete empfangen, gibt $recvfrom$ einen Wert kleiner 0 zurück, die Schleife wird abgebrochen und der Empfangsvorgang ist beendet.

3.2 Sender

Um UDP-Pakete zu senden, müssen folgende Schritte ausgeführt werden.

- Socket erstellen.
- *sockaddr_in* struct erstellen und Attribute festlegen.
- Packets bestimmter Größe erzeugen.
- Mit *sendto* Pakete senden.

3.2.1 Socket erstellen

In C wird mit $socket(AF\_INET, SOCK\_DGRAM, 0)$ ein neuer IPv4-Datagram-Socket erstellt.

3.2.2 sockaddr_in struct erstellen und Attribute festlegen

Um den Empfänger um identifizieren, wird ein $sockaddr_in$-struct erstellt und IP und Port-Nummer des Empfängers den entsprechenden Member-Variablen zugewiesen.

3.2.3 Packets erzeugen

Um Packets bestimmter Größe zu erzeugen, lesen wir das cmd-Argument aus, und rufen mit diesem Wert die $create_msg$-Methode auf, die einen Speicherbereich gegebener Größe mittels *malloc* alloziert, mit einem konstanten char-Wert füllt und einen Pointer auf den erzeugten String zurückgibt. Dieser erzeugte String wird dann bei jedem Sendevorgang zusammen mit der Sequenznummer in den Puffer geschrieben.

3.2.4 Mit sendto Pakete senden

Das eigentliche Versenden der Pakete erfolgt mit der Funktion *sendto*. Diese erwartet als Argumente unseren erstellten Socket, den Buffer, der die Nachricht enthält, die Länge des Buffer, sowie den erstellten sockaddr_in-struct, der die Empfänger-Informationen enthält und dessen Länge. *sendto* wird in einer Schleife ausgeführt, die erst beendet wird, wenn alle Pakete gesendet wurden.

3.3 Empfänger

Um UDP-Pakete zu empfangen müssen folgende Schritte ausgeführt werden.

- Socket erstellen.
- *sockaddr_in* struct erstellen und Attribute festlegen.
- Socket mit Adressen-Objekt mittels *bind* verbinden.
- Timeout für Socket festlegen.
- Mittels $recvfrom$ Pakete empfangen.

3.3.1 Socket erstellen

Der Socket wird auf gleiche Weise erstellt, wie der Sender-Socket.

3.3.2 sockaddr_in struct erstellen und Attribute festlegen

Auch im Empfänger wird ein *sockaddr_in*-struct gebraucht. Dieser erhält als Attribute die Port-Nummer, eine Einschränkung, von welchen IP-Adressen Pakete akzeptiert werden und die Information dass IPv4 verwendet wird.

3.3.3 Socket mit Adressen-Objekt mittels bind verbinden

Da der Socket und die Socket-Eigenschaften hier 2 getrennte Strukturen sind, müssen diese mittels *bind* verbunden werden. Damit werden dem Socket das im $sockaddr_in$-struct definierte Verhalten Attribute zugewiesen

3.3.4 Timeout festlegen und Pakete mit recvfrom empfangen

Die Terminierung des Empfangsvorgangs wird primär nach dem Empfang des letzten Pakets ausgeführt. Um jedoch auch in Fällen, in denen nicht alle Pakete empfangen wurden, oder sonstige Fehler aufgetreten sind eine Terminierung des Programms zu gewährleisten, wurde ein Timeout gesetzt, der in diesen Fällen die Ausführung terminiert. Der Timeout wird erst nach Empfang des ersten Pakets gesetzt, damit mehr Zeit für das Starten des Senders bleibt. Danach werden mit $recvfrom$ Pakete über den Socket empfangen und in den Buffer geschrieben.

3.4 Auswertung

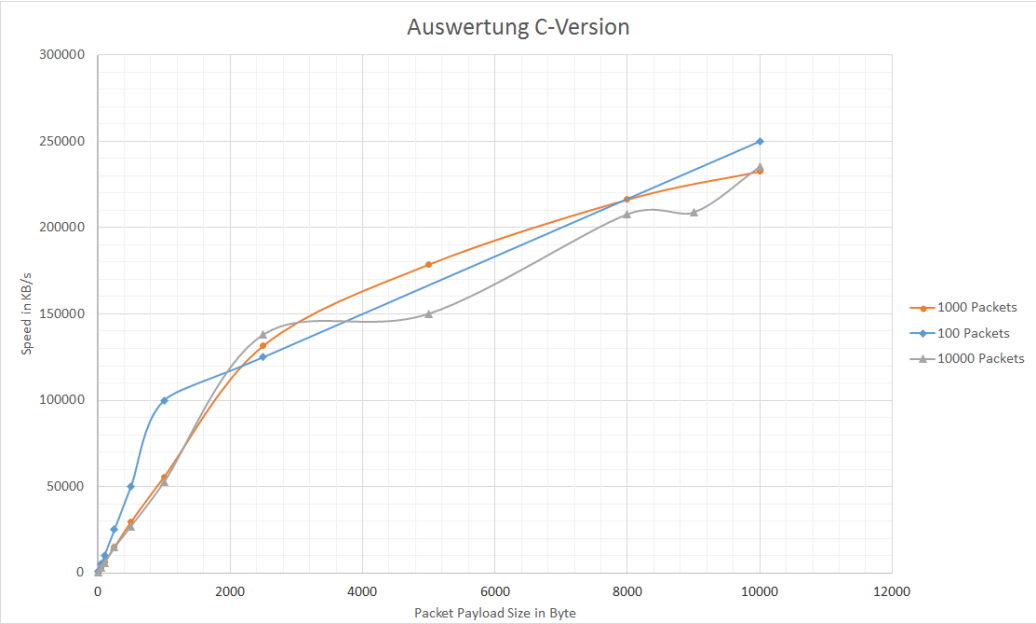
Für die Auswertung werden die Anfangs- und Endzeitpunkte, sowie Anzahl und Größe der Pakete festgehalten. Die Auswertung erfolgt im Sender, wie auch im Empfänger mit der *evaluate*-Funktion. Diese berechnet aus der Paketgröße in Kombination mit der Paketanzahl die Gesamtgröße und mittels Division durch die benötigte Zeit die Übertragungsgeschwindigkeit. Diese Auswertungen sind als Kennlinien im Kapitel Auswertung einzusehen.

4 Auswertung, Aufgabe 1

Ausgewertet wird in hier der Verlauf der Übertragungsgeschwindigkeit als Funktion der Paketgröße. Es wurden jeweils drei Kennlinien erstellt, die die Verläufe beim Senden von 100, 1000 und 10000 Paketen repräsentieren. Die Werte für jede Kennlinie wurden bis zur maximalen Paketgröße aufgenommen, bei der noch keine Paketverluste auftreten. Interessant ist, dass die C-Version trotz annähernd gleicher Implementierung ohne überflüssige Operationen um ein vielfaches schneller empfängt als die Java-Version.

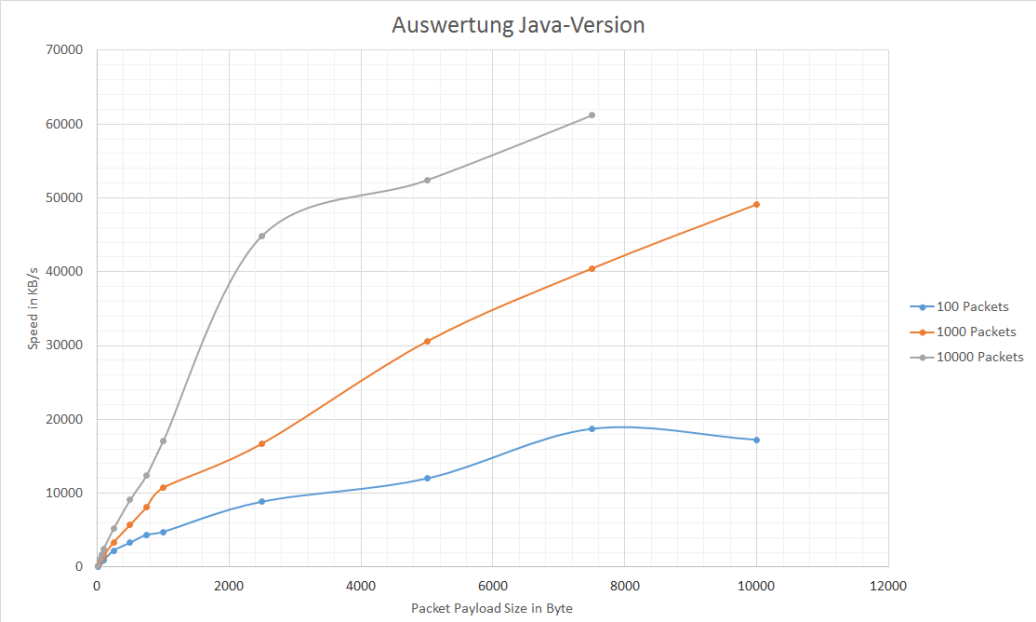
4.1 C-Version

Anfangs steigt die Geschwindigkeit linear an, da mit den zu Beginn kurzen Nachrichten die Übertragungsbandbreite nicht ausgenutzt wird, somit steigt bei steigender Paket-Größe die Geschwindigkeit stark an, da aufgrund nicht vollständig ausgefüllten Kapazität kaum Zeiteinbußen entstehen. Mit stark steigender Paketgröße, also ca ab 4000 Byte nimmt der Zuwachs ab, da wir uns hier der Kapazitätsgrenze der Übertragung nähern und größere Pakete signifikant mehr Zeit benötigen, um empfangen zu werden. Bei einer Paketgröße von ca. 10000 Byte beginnen bei der Übertragung Verluste aufzutreten. Hier wurde nicht weiter geplottet.



4.2 Java-Version

Die Übertragung zeigt hier (bis auf Ausreißer) ein annähernd lineares Verhalten. Hier ist interessanterweise kein Abnehmen des Geschwindigkeitszuwachses zu verzeichnen. Dieser ist langsamer als in der C-Version, aber kontinuierlich und nimmt nicht ab, bevor Verluste auftreten.



4.3 Zusätzliche Bemerkungen

Als interessantes Detail fällt auf, dass ein Senden vom C-Sender zum Java-Empfänger sehr schnell Verluste aufweist, da dieser mit der hohe Sendegeschwindigkeit nicht fertig wird. Erwartungsgemäß gibt es dieses Problem beim Senden vom Java-Sender zum C-Empfänger nicht. Dies bestätigt die Annahme, dass der Java-Empfänger um ein Vielfaches langsamer ist als sein C Pendant.

5 Aufgabe 2

Beide Programme modifizieren, sodass die letzten 4 Byte des letzten Pakets eine CRC32 checksum über alle Pakete enthält, die dann im Empfänger überprüft werden kann. Zusätzlich sollen die Pakete jetzt über LAN gesendet und die Ergebnisse (Geschwindigkeit, verlorene Pakete) analysiert werden.

6 Änderungen, Aufgabe 2

Aufgrund der Schwierigkeit, in C über UDP eine Nachricht zu versenden, die sowohl Characters als auch Integers enthält, wurde die zu sendende Nachricht in beiden Programmversionen auf reine Integer geändert. Dies ermöglicht es, Sequenznummer, Nachricht und CRC checksum unkompliziert zusammenzuhängen. Weiters musste wegen dieser Änderungen die Eingabe und Auswertung geändert werden. Der Paketgrößen-cmd-Parameter erwartet nun die Größe als Anzahl von Integers. Da die C Version nun mit Integers arbeitet, wurde bei der Berechnung der absolut gesendeten/empfangenen Daten der Faktor 4 eingeführt, um dies zu berücksichtigen.

6.1 Java

In Java wird zur einfacheren Handhabung ein ByteBuffer verwendet. Dieser wird mittels $putInt$ mit Integer-Werten gefüllt. Schließlich kann mit $ByteBuffer.array$ ein Byte-Array extrahiert werden, welchen dann in ein UDP-Paket verpackt und versendet wird. Der Empfänger erhält nun ein Byte-Array, das mittels $wrap$ einem ByteBuffer zugewiesen wird. Nun können Integer-Werte einfach mittels $getInt$ extrahiert werden. Die CRC-Berechnung wird in Java mit einer Instanz der Klasse CRC32 realisiert. Für jedes zu sendende Paket (Byte-Array) wird $update(Byte-Array)$ aufgerufen und damit der CRC-Wert aktualisiert. Das letzte Paket wird nur bis zur vorletzten Stelle des Byte-Arrays verwendet, um den CRC zu aktualisieren. Anschließend wird der CRC-Wert an die letzte Stelle geschrieben und das Paket versendet. Die CRC Checksum ist ein long-Wert, dessen 32 niederwertigste Bits verwendet werden. Da der CRC-Wert aber als Integer versendet wird und somit eine 1 an der ersten Stelle eine negative Zahl ergeben würde, muss der 32-bit signed Integer-Wert jetzt als unsigned-Integer interpretiert werden. Dies geschieht in der $getUnsigned$-Methode, die dann den ursprünglichen, positiven long-CRC-Wert zurückgibt.

6.2 C

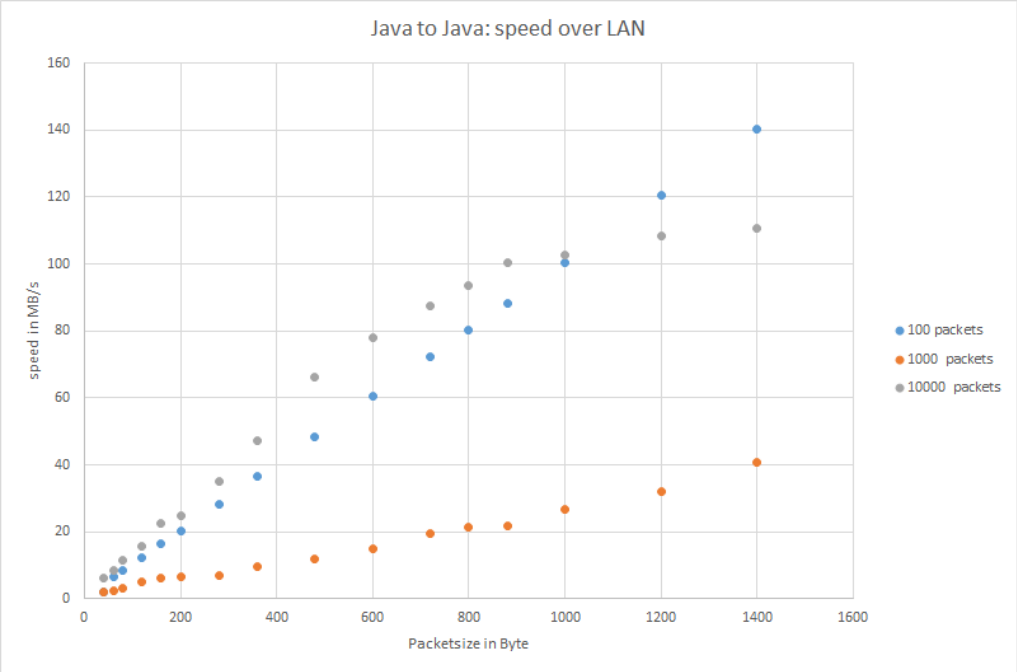
In C wird für die Berechnung des CRC32 Wertes die $crc32$-Funktion aus der zlib-Libary verwendet. In C wird jetzt ein int-Array versendet. Dafür wird zuerst ein int Array gegebener Länge erstellt und mit Zahlen gefüllt. In jedem Sendeschritt wird nun nur die erste Stelle durch die Sequenznummer ersetzt. Das fertige, zu versendende Array wird dann verwendet, um die CRC Checksum zu aktualisieren. Beim letzten Paket wird wiederum (analog zur Java-Version) das Array ohne die letzte Stelle verwendet, um die

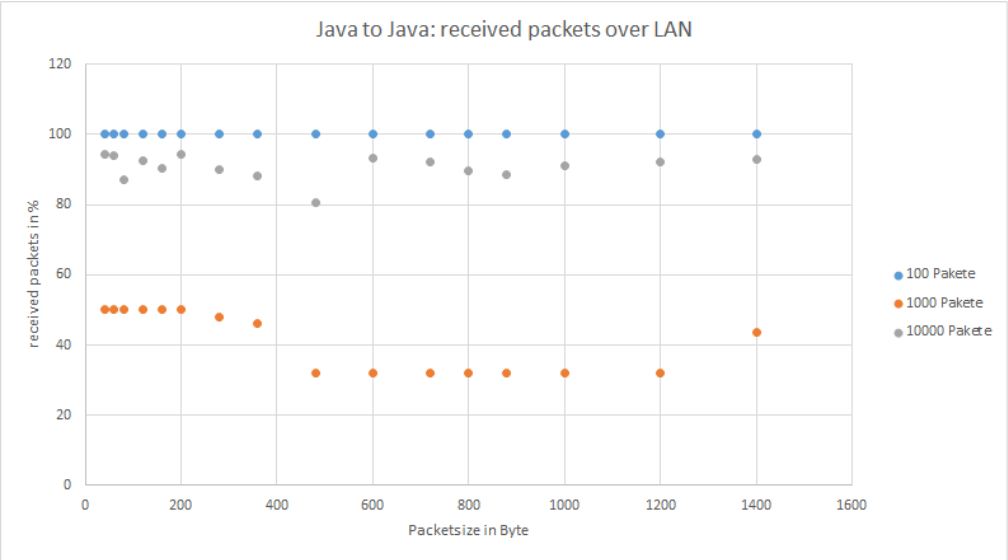
Checksum zu aktualisieren. Diese wird dann an die letzte Stelle geschrieben und das Array als Paket versendet. Der Empfänger schreibt wiederum jedes ankommende Paket in ein Buffer-Array und verwendet den Paketinhalt (Int-Array), um die Checksum zu aktualisieren. Das letzte Paket wird natürlich ohne die letzte Stelle verwendet, die ja die von Sender berechnete Checksum enthält. In C muss die Checksum nicht extra als unsigned interpretiert werden, da C (anders als Java) unsigned Datentypen unterstützt. Im C Programm wurde auch ein Berechnungsfehler behoben, der dazu führte, dass kurze Übertragungszeiten nicht gemessen werden konnten. Um dies zu erreichen, wird die Übertragungszeit jetzt in Mikrosekunden gemessen und nicht wie bisher in Millisekunden umgerechnet (dies führte zu einigen 0-Werten und machte damit eine Berechnung der Geschwindigkeit unmöglich).

7 Auswertung, Aufgabe 2

7.1 Java-Java

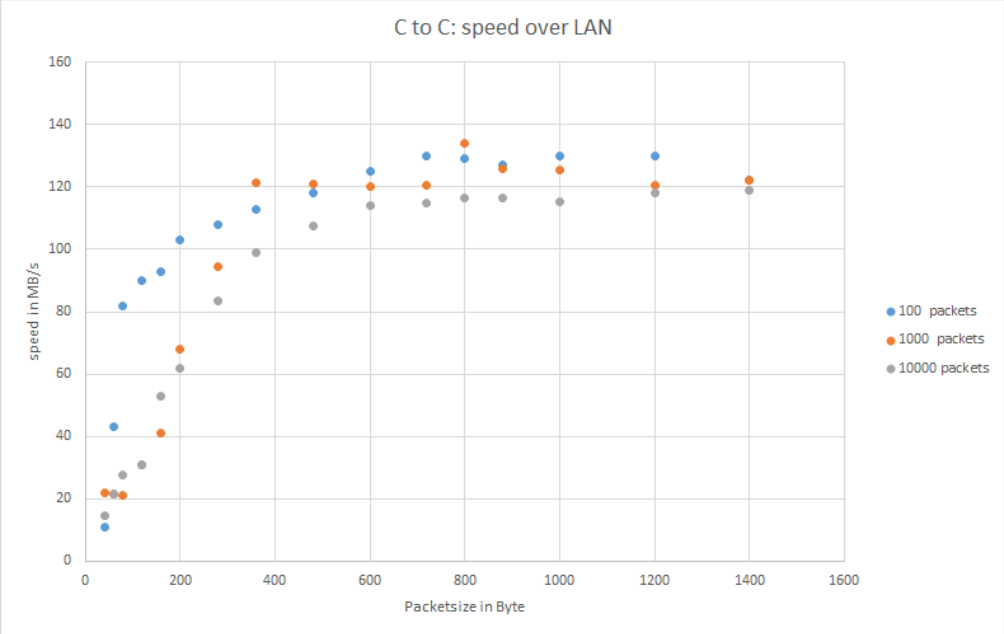
Das Senden von Packets über LAN vom Java-Sender zum Java-Empfänger gestaltet sich ähnlich dem Senden über die lokale Schnittstelle. Eine Anomalie fällt jedoch auf: Auch nach mehrmaligem Wiederholen der Tests befindet sich die Kennlinie für 1000 Pakete unter der für 100 Pakete. Dieses Verhalten lässt sich kaum erklären, da das Senden von 1000 Paketen eigentlich zwischen den Geschwindigkeiten für 100 und 10000 Pakete liegen müsste. Interessant ist auch die Betrachtung der Verlorenen Pakete. Während beim Übertragen über localhost fast nie ein Paket verloren ging, gehen hier nun sehr viele Pakete verloren. Besonders fällt das bei den Werten für 1000 Pakete auf.

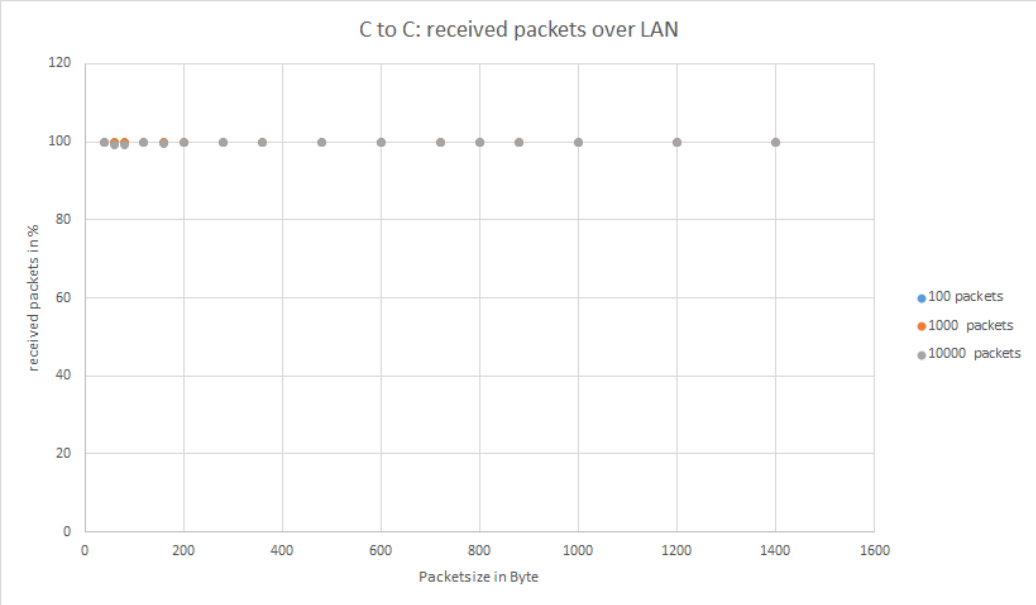




7.2 C-C

Beim Senden von C nach C ist das Verhalten wiederum dem aus der vorherigen Aufgabe (localhost) sehr ähnlich. Es fällt auf, dass hier kein einziges Paket verloren geht. Der Geschwindigkeitszuwachs verhält sich gleich, wie beim senden über localhost.

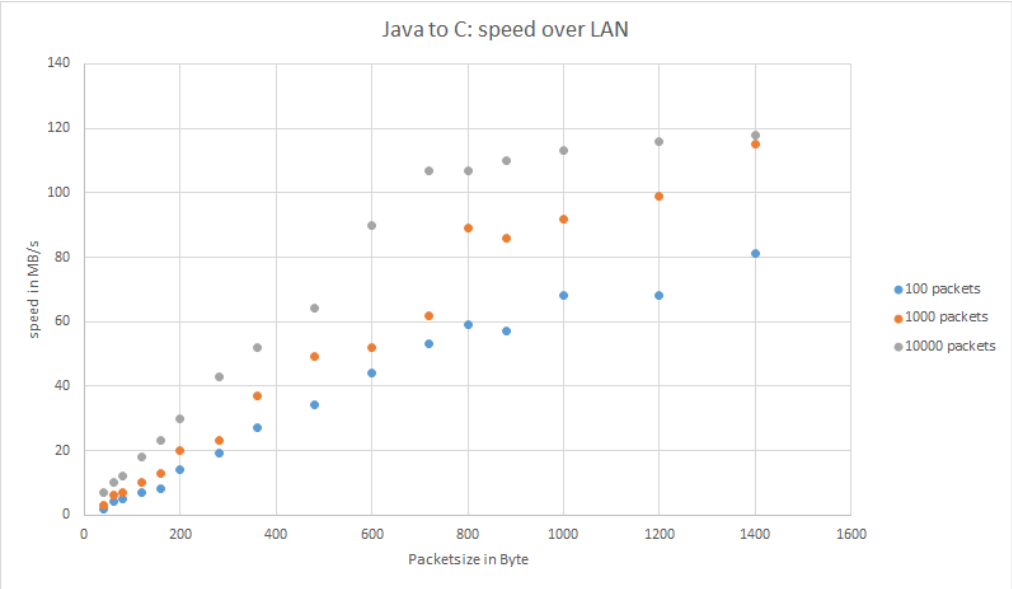


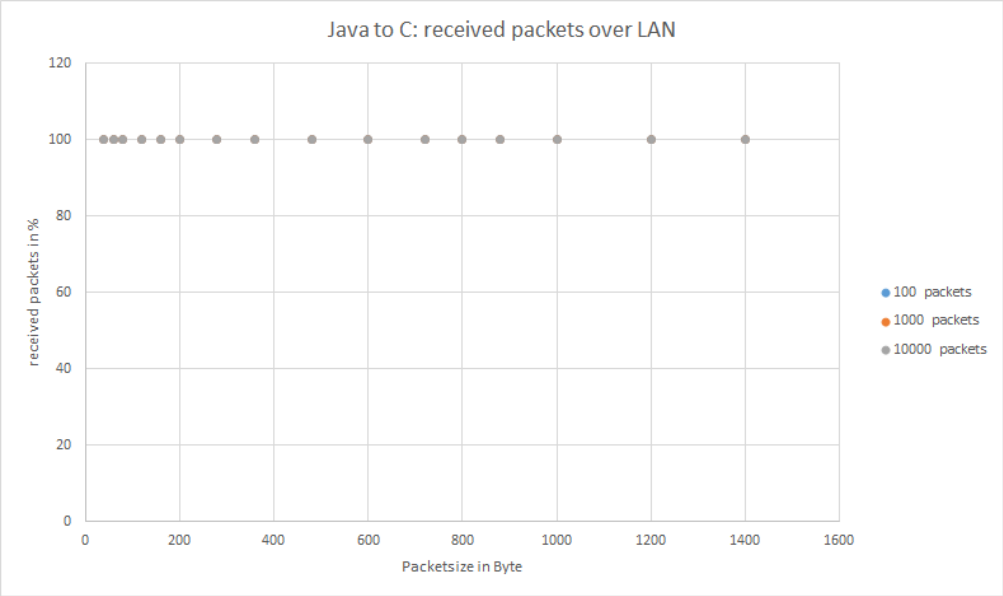


7.3 Java-C

Beim Senden von Java nach C ist wiederum ein annähernd linearer Verlauf zu beobachten. Hier befinden sich die Werte für 1000 Pakete jedoch zwischen denen für 100 und 10000. Bezüglich verlorener Pakete ist zu sagen,

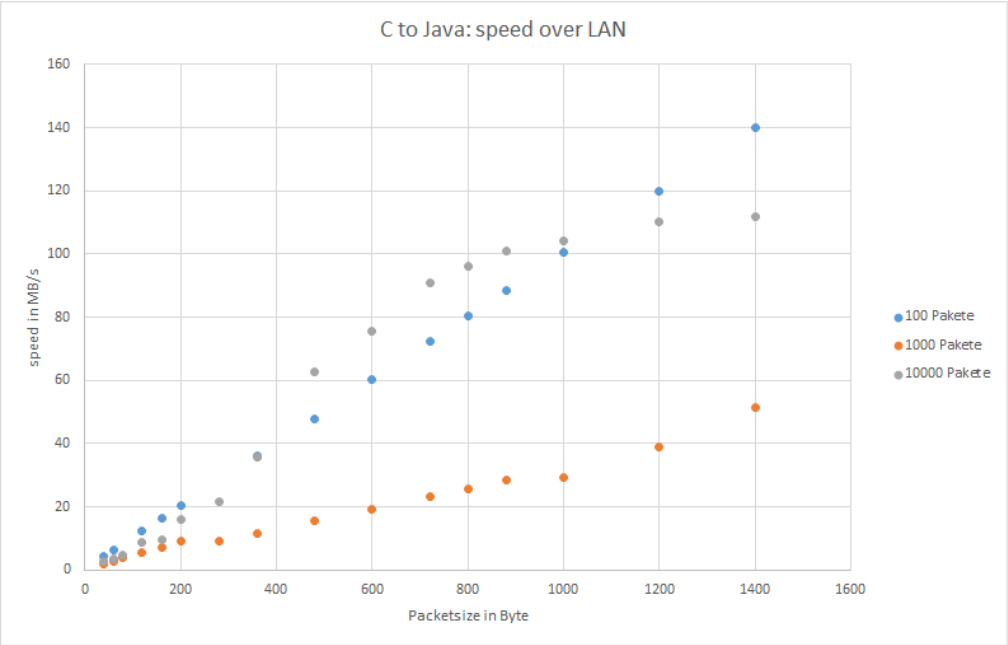
dass der Java-Sender höchstwahrscheinlich um einiges langsamer ist als der C-Empfänger. Die meisten Paketverluste entstehen, weil der Empfänger die Pakete nicht schnell genug verarbeiten kann. Dass hier kein einziges Paket verloren geht liegt wahrscheinlich daran, dass ein schneller Empfänger mit einem langsameren Sender zusammenarbeitet und der Empfänger damit genug Zeit hat, um die Pakete zu verarbeiten.

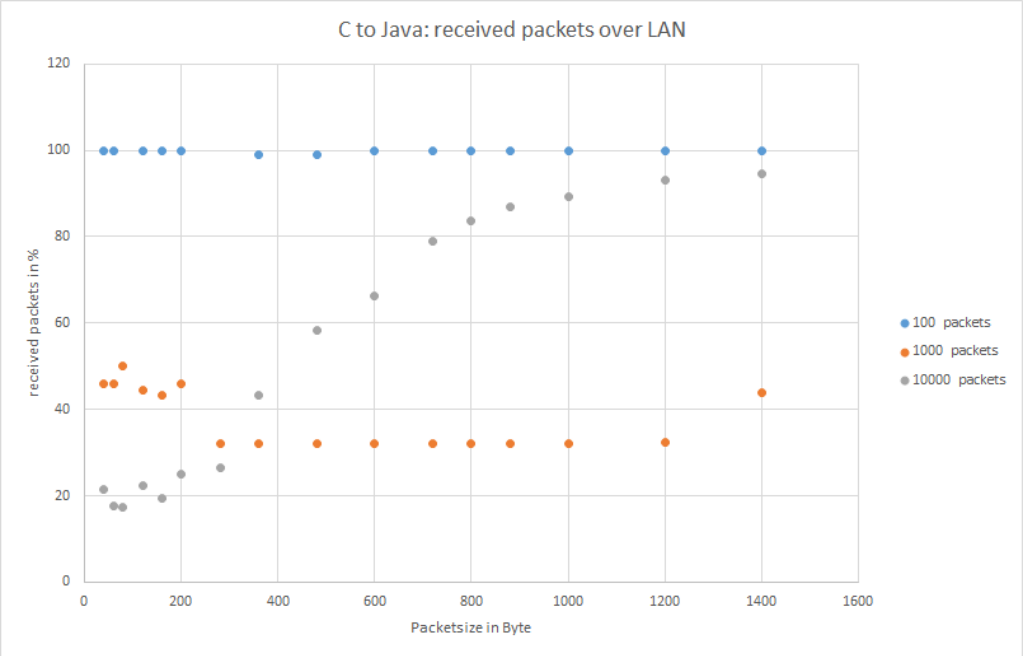




7.4 C-Java

Beim Senden von C nach Java lässt sich ein sehr linearer Geschwindigkeitsverlauf beobachten. Da die Übertragungsgeschwindigkeit ja größtenteils vom Empfänger beschränkt wird, ähnelt der Verlauf hier sehr dem beim Senden von Java nach Java. Die Paketverluste sind in diesem Fall sehr groß. Das tritt vermutlich deshalb auf, da hier der genau umgekehrte Effekt auftritt wie beim senden von Java nach C. Wir vermuten, dass der C Sender so schnell ist, dass der Java Empfänger einfach nicht mit kommt, und dadurch viele Pakete verpasst. Außerdem treten auch hier die größten Verluste beim versenden von 1000 Paketen auf.




C to Java: received packets over LAN
120
100
80
60
40
20
0
received packets in %
0
200
400
600
800
1000
1200
1400
1600
Packetsize in Byte
100 packets
1000 packets
10000 packets

8 Aufgabe 3

Ziel ist die Implementierung eines Schiebefensters-Protokolls. Das führt dazu, dass jetzt keine Paketverluste mehr auftreten. Unsere Messungen können sich jetzt auf die Übertragungsgeschwindigkeit beschränken.

9 Funktionsweise Schiebefenster

ein Schiebefenster wird hier auf der Sender Seite verwendet und hat den Sinn sicherzustellen, dass keine Pakete mehr verloren gehen. Das könnte auch mit dem Stop-and-Wait Verfahren realisiert werden. Dabei würde die Übertragungsleitung jedoch nur unzureichend ausgenutzt, da immer nur ein Paket gesendet wird, auf dessen Bestätigung dann noch zusätzlich gewartet werden muss. Das Schiebefensterverfahren zielt darauf ab, die Leitungskapazität optimal auszunutzen. Das funktioniert folgendermaßen:
Der Sender sendet zuerst so viele Pakete, wie durchs Schiebefenster vorgegeben und wartet dann darauf, dass das erste Paket bestätigt wird. Ist dies der Fall, rückt das Schiebefenster um eine Position weiter und das nächste Paket wird gesendet. Das Schiebefenster ist hier also nichts anderes als eine Liste gesendeter, aber noh nicht bestätigter Pakete. Wird jedoch nach einem bestimmten Timeout keine Bestätigung erhalten, oder wird das falsche Paket bestätigt, wird das Schiebefenster gelehrt und der Sender sendet alle darin enthaltenen Pakete erneut.

10 Änderungen, Aufgabe 3

Änderungen umfassen hier das Ändern des Sendemodus. Der Sender sendet nun nicht mehr durchgehend von ersten bis zum letzten Paket, sondern anfangs nur bis das Schiebefenster voll ist. Ist das Schiebefenster voll, muss darauf gewartet werden, dass das erste Paket im Fenster bestätigt wird, um das nächste Paket zu senden.

10.1 Java

Empfänger: Auf der Empfängerseite wurden nur kleine Änderungen vorgenommen. Der Empfänger muss aus dem ersten erhaltenen Paket die Sender-Informationen extrahieren. Zusätzlich muss aus jedem erhaltenen Paket die Sequenznummer ausgelesen werden, diese in ein Paket verpacken und an den Empfänger mit der extrahierten Adresse retournieren. Zusätzlich verwaltet der Empfänger einen Zähler, der immer die letzte bestätigte Sequenznummer

festhält.

Sender: Auf der Sender-Seite muss etwas mehr verändert werden. Der Sender verwaltet eine Variable *windowSize*, die die Größe des Schiebefensters angibt und auch als cmd-Parameter eingegeben werden kann. Das letzte erhaltene ACK wird in der Variable *LAR* festgehalten.
Nach jedem gesendeten Paket wird jetzt überprüft, ob das Schiebefenster schon voll ist, also ob das letzte gesendete Paket die Sequenznummer $LAR+$ *windowSize* hatte. Ist dies der Fall wird nun auf eine Bestätigung gewartet. Das ACK-Paket sollte optimalerweise die Sequenznummer des ersten Pakets im Schiebefenster enthalten. Wird die richtige Sequenznummer erhalten, sendet der Sender das Nächste Paket → das Fenster rückt um eine Position weiter.
Erhält der Sender jetzt ein ACK mit einer falschen Sequenznummer, verwirft er das Fenster. In unserem Fall wird einfach der Schleifenzähler auf *LAR* (danach von Schleife erhöht) gesetzt und alle Pakete im Fenster noch einmal gesendet. Es kann außerdem vorkommen, dass überhaupt kein ACK erhalten und damit ein gewisses Empfangs-Timeout überschritten wird. In diesem Fall gilt das gleiche: Fenster verwerfen, Schleifenzähler auf auf *LAR* setzen und Pakete erneut senden. Hier musste zusätzlich ein kurzes *sleep* eingebaut werden. Angenommen es wird ein falsches oder kein ACK erhalten. Dann würde der Sender den Schleifenzähler zurücksetzen und direkt beginnen zu senden. Jetzt würde der Sender ein ACK erwarten, dass die Sequenznummer des ersten gesendeten Pakets enthält. Hier kann es aber nun vorkommen, dass noch ein ACK vom vorherigen Sendevorgang unterwegs ist und empfangen wird. Das würde der Sender jetzt als ein falsches ACK identifizieren und den Sendevorgang gleich wieder wiederholen. Dies führt zu einer unendlichen Schleife, in der der Sender immer wieder den gleichen Abschnitt sendet und nicht mehr weiter kommt. Um das zu verhindern wurde ein kurzes *sleep* eingebaut, das dafür sorgt, dass der Sender nachdem ein falsche Paket oder ein Timeout erkannt wurde kurz wartet bis die restlichen ACKs die "Leitung"verlassen haben.

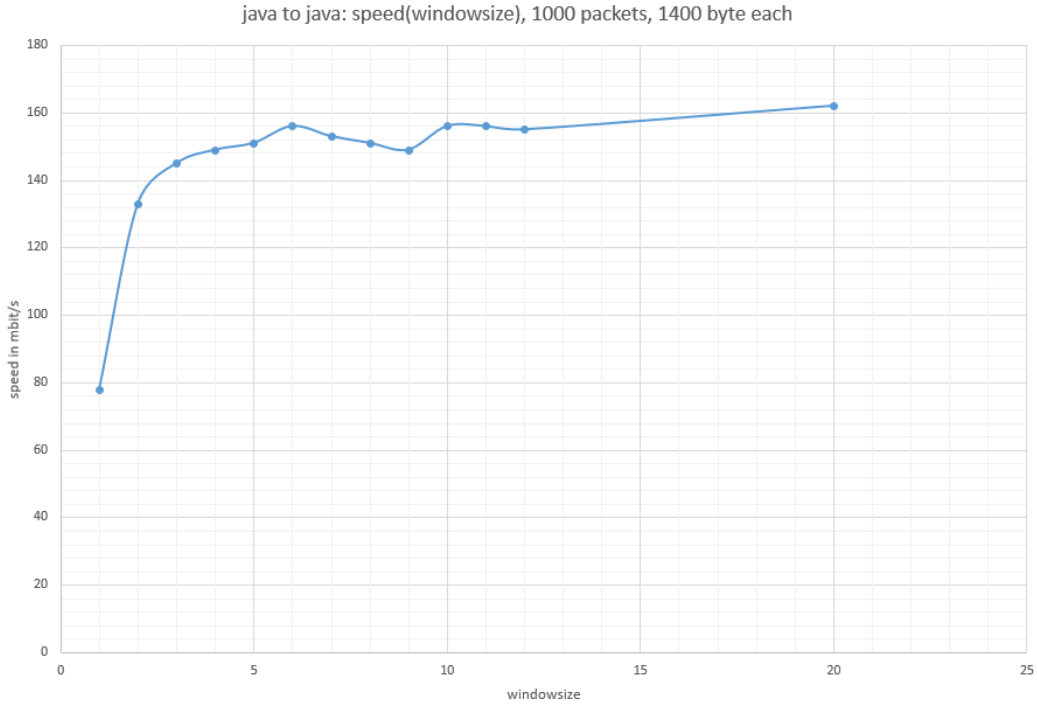
10.2 C

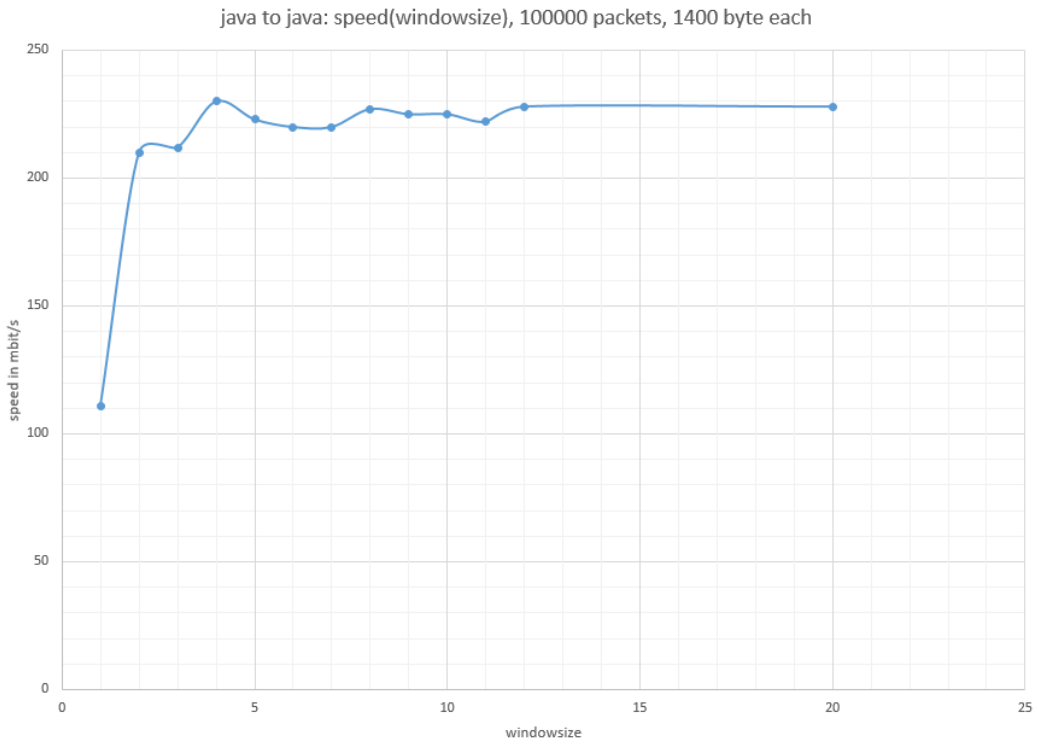
In der C-Version wurden praktisch die gleichen Änderungen vorgenommen, wie in der Java-Version, nur ist es etwas komplizierter diese Änderungen vorzunehmen.
Empfänger: Im Empfänger wurde bereits früher ein struct vom Typ $sockaddr_in$ erstellt, der nach jedem Empfang eines Pakets die Sender-Informationen enthält. Nun wird aus jedem empfangenen Paket, das hier als int-Array re-

präsentiert ist die Sequenznummer extrahiert, also $array[0]$ gelesen, in ein neues Array der Länge 1 und damit in ein neues Paket verpackt. Anschließend wird dieses neue Paket an den Sender zurückgeschickt.
Sender: Der Sender agiert exakt gleich wie der Java-Sender. Es wird gesendet bis das Schiebefenster voll ist, dann wird auf ein ACK gewartet. Ist dieses ACK die Bestätigung des ersten Pakets im Fenster, wird fortgesetzt → ein weiteres Paket gesendet. Wird kein oder das falsche ACK erhalten, wird auch hier der Fensterinhalt verworfen und der Schleifenzähler auf LAR gesetzt (danach von Schleife erhöht) → alle Pakete im Fenster werden erneut gesendet.

11 Auswertung, Aufgabe 3

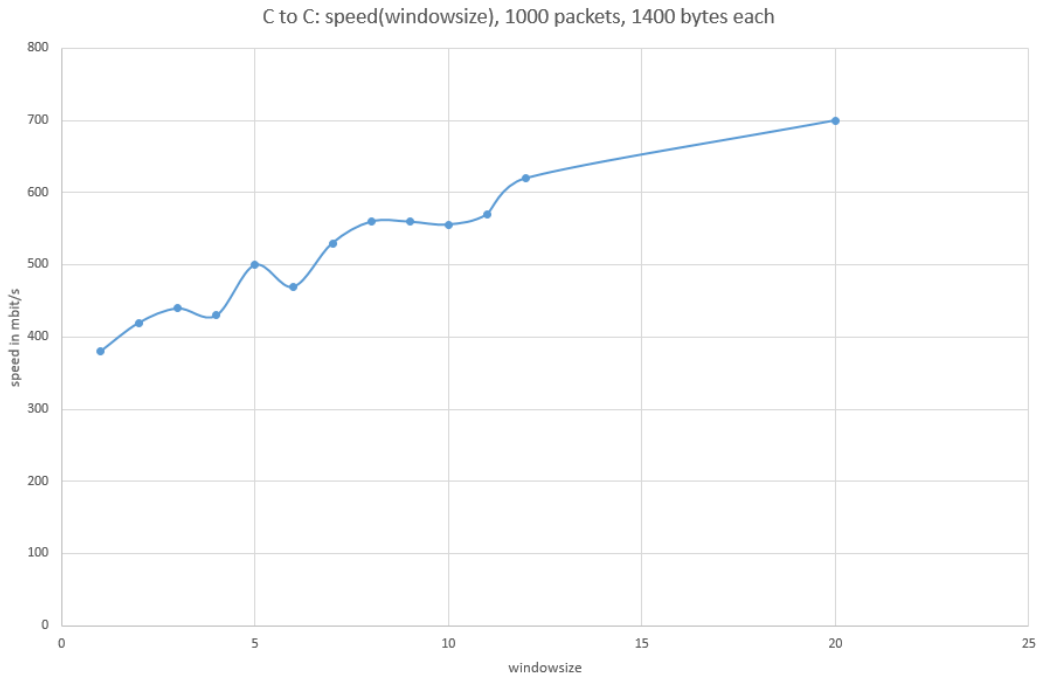
11.1 Java-Java

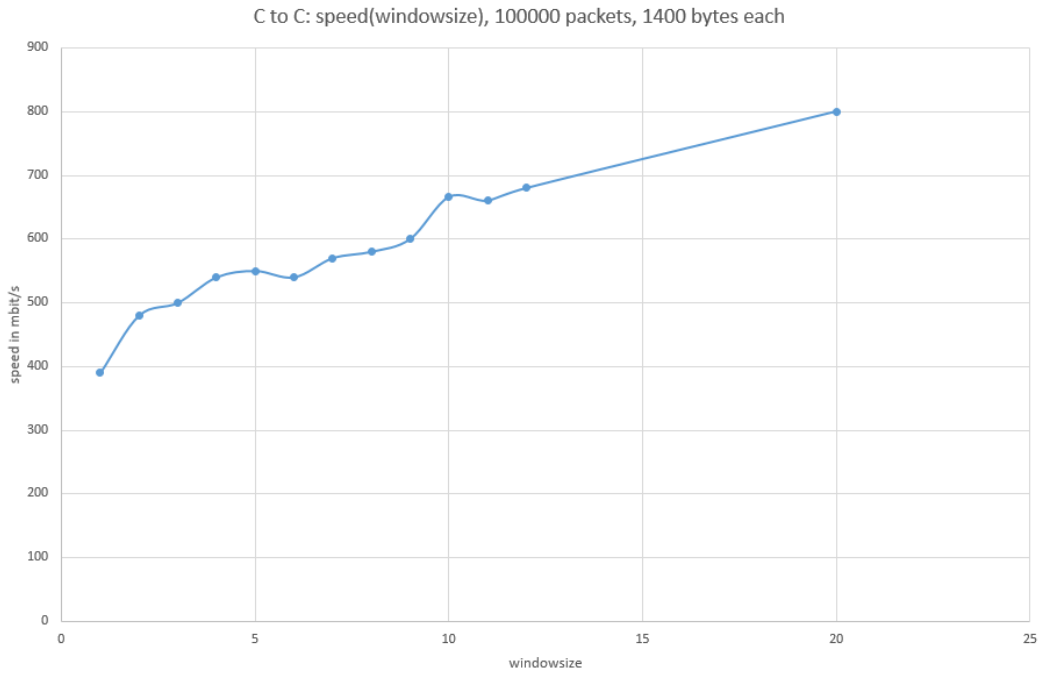




Hier wurden jeweils 1000 und 100000 Pakete mit einer Paketgröße von 1400 Byte über den localhost gesendet und die Übertragungsgeschwindigkeit in Abhängigkeit von der Größe des Sliding-Window gemessen Hier fällt auf, dass beim Übergang von Fenstergröße 1 zu Fenstergröße 2 ein enormer Sprung auftritt. Das ist dadurch zu erklären, dass bei Größe 1 immer auf ein ACK für das gerade gesendete Paket gewartet wird (entspricht dem Stop-and-Wait Verfahren). Sobald mehr als 1 Paket ins Fenster passt, werden mehr Pakete gesendet, bevor auf eine Antwort gewartet wird. Es sind also mehrere Pakete gleichzeitig unterwegs, was die Leitungskapazität viel besser ausnutzt.
Ansonsten zeigt die Java-Version ein logarithmisches Verhalten in Abhängigkeit von der Fenstergröße. Ungefähr bis zur Fenstergröße 5 lässt sich ein Anstieg messen, danach bleiben die nahezu konstant. Dies könnte durch die kurze roundtrip-time beim Senden über localhost erklärt werden: die Kapazität ist schon beim Senden von wenigen Paketen erreicht, weitere Erhöhung der Fenstergröße bringt nichts. Andererseits könnte es sich auch um eine Beschränkung durch Java handeln, da dieses logarithmische Verhalten bei der C-Version nicht auftritt.

11.2 C-C





Die C-Version wurde auf gleiche Art und Weise ausgewertet, wie die Java-Version. Hier fällt auf, dass der Geschwindigkeitszuwachs sehr linear ausfällt

und keine Sättigungseffekte auftreten → die Geschwindigkeit steigt immer weiter linear an und ist generell viel höher als in der Java-Version (evtl. unrealistisch?).