UPTEC IT 08 012
Examensarbete 30 hp
Oktober 2008

Utveckling av generell testklient för Nordic Growth Market

Erik Scholander

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Development of Generic Test Client for Nordic Growth Market

Erik Scholander

Development and accuracy tests of advanced exchange trading systems require well adjusted tools for testing. Aptness of these tools are probable to have a major impact on identification of bugs and verification of new functionality. Identifying erroneous parts and bottlenecks of the trading system is of outmost importance for the exchange as a whole since stability and correctness are two of the determinant factors its viability is measured by. This paper covers how this test-tool was created and the motivation behind the design choices made. The tool was created with the goal to be broad enough to cover the majority of cases that could arise, fast enough to be able to test the trading systems speed limitations (i.e. faster than the trading system) and cost efficient to maintain and use.

The solution was a highly efficient library with a bit optimized implementation of the UniTP protocol attached to a script engine. While it do require some degree of programming skill of the user, it produces a very broad, fast and cost efficient tool.

Handledare: Mikael Brännström
Ämnesgranskare: Jim Wilenius, Arne Andersson
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT08 012
Tryckt av: Reprocentralen ITC

Innehållsförteckning

Introduktion och bakgrund

Detta examensarbete beskriver utvecklingen av en testklient till handelssystemet för en av Sveriges mest aktiva handelsplatser; NGM. Nordic Growth Market (NGM) är en auktoriserad börs för nordiska tillväxtbolag, deras primära lista för aktiehandel är NGM Equity samt NDX och finns tillgänglig både på nätet och i dagstidningarna.

En börs styrs av mycket strikta lagar och står i Sverige under Finansinspektionens tillsyn. Det finns tydliga regler för vad börsnoterade företag måste informera om och hur informationen skall vara utformad för börsnoterade företag, likväl finns det en rad regler om hur handel i dess aktier får ske. En börs finns till för att företag skall kunna notera sina aktier för köpare och säljare samt att handeln övervakas och sker efter ställda lagar samt med en viss likviditet. För att ett företag skall få sälja sina aktier på en börs krävs att företaget genomgår en omfattande granskning, en så kallad noteringsprocess. NGM genomför dessa granskningar och kontrollerar kontinuerligt att de börsnoterade företagen följer de lagar och riktlinjer som gäller för börsnoterade företag. De genomför även marknadsövervakning och försöker identifiera insiderhandel eller andra oangelägenheter som kan förekomma på börsen.

Det finns flera *instrument* som kan köpas och säljas på börsen. Ett instrument är ett övergripande namn på de olika typer av värdepapper som emitteras till NGM vilket kan vara; aktier, optioner, warranter eller obligationer med mera. Om en aktör vill handla i ett instrument lägger denne en så kallad *order*. En order är av typen köp- eller sälj ett bestämt antal instrument till angivet pris. Om en köp- och säljorder matchar pris (eller överlappar varandra) sker ett avslut till den först inlagda orderns pris. Varje instrument har således 3 kurser. *Säljkursen* vilket är den lägsta säljordern som finns i systemet, *köpkursen* som är den högsta köpordern och *senast avslut* vilket är priset på det senast genomförda avslutet. Skillnaden mellan högsta köpordern och lägsta säljordern kallas för *spread*. Alla instrumentpriser sker i *punkter* (även kallat *ticks*), det är de olika prisnivåerna som man kan köpa och sälja aktier på, tillsammans bildar alla ordrar på en viss prisnivå en *level*. Det är den sammanslagna volymen av alla ordrar på en specifik prisnivå. Flera prisnivåer skapar tillsammans ett *orderdjup*.

		Exempel på orderdjup för Ginger Oil AB			
		Köp		Sälj	
	level	Antal	Pris	Pris	Antal
Orderdjup	1	3500	20,50	20,80	13500
	2	17000	20,40	20,90	6000
	3	13500	20,30	21,00	5500
	4	2500	20,20	21,10	3000
	5	7000	20,10	21,30	12500

Här kan vi se 5 levlar i orderdjupet, och vi ser en spread på 0,30. Om senaste avslutet var på 20,50 så skulle instrumentets kurs vara: *köp*: 20,50, *sälj*: 20,80 och *senaste*: 20,50.

NGM vänder sig uteslutande till större aktörer för handel som till exempel Nordea, Avanza eller SE Banken, vilka antingen är ombud för flera mindre aktörer (tex privatpersoner) eller är stora fondförvaltare. Dessa aktörer ansöker om tillstånd för handel hos Finans Inspektionen. Om tillstånd ges får de bedriva handel hos NGM och antas ha den likviditet som krävs utan kreditkontroll, dvs. om de lägger in en order på att sälja tusen Ericsson aktier så antas de ha

tusen Ericsson aktier. NGM kontrollerar således inte köparnas eller säljarnas tillgångar eller genomför själva *clearingen*. När ett avslut sker meddelas båda aktörerna och de genomför själva transaktionen hos Värdepapperscentralen (VPC). På börsen fattas ett beslut om handel, själva transaktionen sker sedan inom 3 dagar och det åligger aktörerna att detta sker på ett korrekt sätt. NGM spar all avslutshistorik, och om eventuella stridigheter skulle uppstå är det denna historik som skall beaktas. NGM har även möjligheten att genomföra tvångsmakuleringar av avslut där det skett uppenbara fel eller om handel skett på ett otillbörligt sätt. Om det finns misstanke att handel skett olagligen vidarebefordras ärendet till polisiär myndighet.

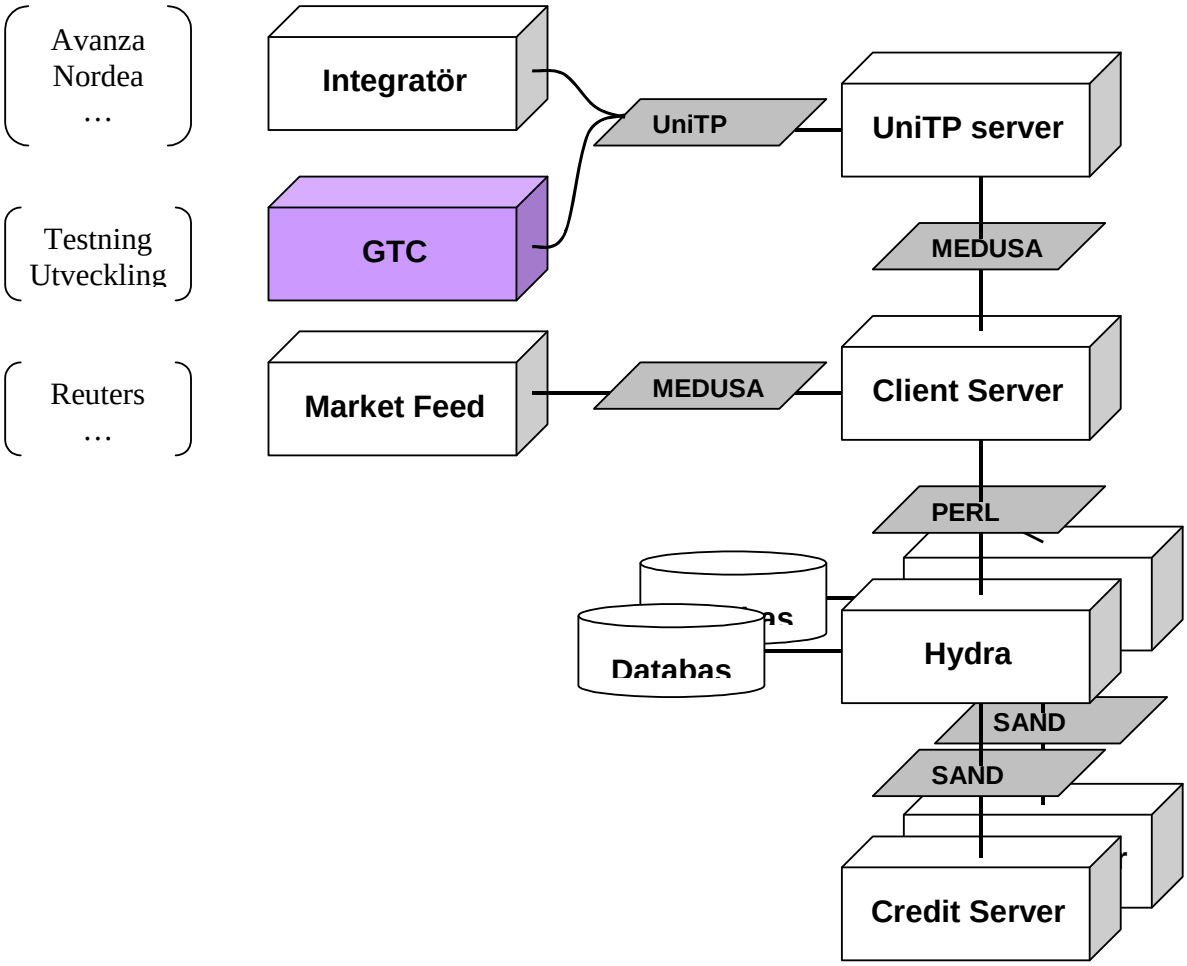
NGM Equity drivs av ett handelssystem som har hand om alla ordrar, modifikationer och användare. Det matchar ordrar och skapar avslut mellan de olika aktörerna samt ser till att handeln sker enligt regelsystemet för aktiehandel. Handelssystemet har ett inbyggt system, *Falken*, vilket kontrollerar all handel. Det larmar vid all handel som kan anses suspekt eller sker enligt vissa mönster. När ett larm uppstår undersöker marknadsövervakningen ärendet och vidtar om nödvändigt åtgärder. I vissa fall kan handeln tvångsmakuleras, aktören uteslutas från handel eller ärendet överlämnas till polis.

Handelssystemet är uppbyggt i flera lager och är spritt över flera servrar, dels för att utvecklingen av marknaden har drivit fram nya standarder och protokoll, men framför allt för att kunna sprida ut belastningen så handelssystemet klarar av att hantera den enorma mängd ordrar som inkommer och modifieras varje dag. Handelssystemet utvecklas kontinuerligt, och det är för närvarande (2007) två stycken utvecklare som driver den. Redan under nästa år (2008) kommer NGM strukturera om systemet radikalt för att kunna införa ny funktionalitet och kunna balansera orderhanteringen ännu bättre mellan flera servrar.

För att kunna kontrollera dagens och framtidens handelssystem fungerar korrekt för slutkunden behövs ett kraftfullt verktyg för att kontrollera funktionaliteten och identifiera buggar. En bugg i handelssystemet skulle kunna ha förödande effekter på marknaden, och om det skulle bli fel i handeln eller om handelssystemet stannar kostar det enorma summor för både aktörerna och NGM. Om tekniska fel upprepas kan det ha förödande konsekvenser för NGM som företag. Önskemålet om utvecklingen av ett nytt testverktyg för att identifiera buggar och verifiera ny funktionalitet är upprinnelsen till detta exjobb.

Hur handelssystemet är uppbyggt

Idag är stommen i handelssystemet är en applikation som kallas *Hydra*. Den styr när marknaden öppnas och stängs, tar emot och lägger in nya ordrar och kontrollerar om en användare verkligen har rättigheter att modifiera en order. Hydra har dock inte hand om marknadens alla instrument, utan ett begränsat antal så den inte överbelastas. Det finns således flera Hydror som var och en har hand om ett fåtal instrument var. Varje Hydra har en databas med instrumentdata, ordrar och användare och modifierar sin databas utifrån de instruktioner som inkommer från *Client Server*. Den kan lägga till nya, ta bort och modifiera objekt, men den söker dock inte efter avslut; det sker i en modul kopplad till Hydran som heter *Credit Server*. Den söker efter matchande eller överlappande ordrar och gör avslut mellan dem. Hydra och Credit Server kommunicerar med ett protokoll som heter *SAND*.



Över Hydran sitter en *Client Server*, den har två huvudsakliga uppgifter, dels översätter den från Hydrans språk med *PERL-objekt* till ett protokoll som heter *MEDUSA*, men framför allt fungerar den som en switch mellan olika Hydror. Eftersom en enskild Hydra har hand om ett begränsat antal instrument, till exempel endast Ericsson, och andra Hydror kontrollerar handeln med andra instrument behövs det en applikation för att switcha så användarens Ericsson order kommer till rätt Hydra. Kopplat till Client Server finns flera komponenter, bland dem; *Market Feed* och *UniTP server*.

Market Feed lyssnar på Client Server och kommunicerar information om alla avslut samt inkomna, borttagna eller modifierade ordrar kontinuerligt till olika nyhetskunder, till exempel Reuters som trycker det i morgondagens tidning, eller till NGM's hemsida som kontinuerligt visar marknadsdata.

UniTP-servern är en protokollöversättare från MEDUSA till protokollet *Univits Transfer Protocol*, förkortat *UniTP*. UniTP är ett standardiserat protokoll som gör det lätt för slutkunder att implementera det i sin handelsklient. Det finns en mängd olika integratörer som implementerar UniTP i sina applikationer och möjliggör kommunikation med NGMs handelssystem. Några av de mer välkända handelssystemen är *ORC* och *Front*, vilka används av merparten av dem som handlar på NGM börsen.

Det är även till UniTP servern som den generella testklienten (GTC) skall kopplas. Denna rapport beskriver hur utvecklingen av denna klient genomfördes.

Hur utveckling och testning fungerar hos NGM

Vid test av handelssystemet används en utvecklingsserver vilken är liknande den som används vid handel. Det är på den som ny programvara testas innan det tas i bruk på servrarna som sköter handelssystemet. På utvecklingsservern finns det ordrar och instrument som kan modifieras, läggas till eller tas bort och den kan startas om eller kraschas utan konsekvenser. Att använda sig av en klient som ORC eller Front skulle vara komplicerat och otillräckligt i vissa avseenden när man utvecklar eller testar utvecklingsservern. Dessa klienter utvecklas av tredjepart, så de utvecklas i takt med att protokollen utvecklas och presenteras som färdig produkt och kan därför inte användas för att testa funktionalitet som håller på att utvecklas. För att på ett snabbt och effektivt sätt kunna kontrollera att marknaden beter sig som förväntat samt att enkelt kunna reproducera eventuella fall som visar sig vara problematiska eller som kunderna har rapporterat som abnormaliteter behövs ett kraftfullt verktyg för att kunna skapa dessa situationer snabbt och enkelt. Vidare behövs verktyg för att kunna generera stora mängder trafik för att simulera överbelastning, och för att verifiera att börsen beter sig korrekt när överbelastning sker. Innan genomförandet av detta examensarbete saknades ett sådant generellt verktyg och att göra mer avancerade testfall kunde kräva mycket tid och analys av svaren tog mycket tid. För att enkelt kunna kommunicera med börsen och kontrollera svaren krävs en klient speciellt anpassad för detta ändamål. En klient som enkelt och snabbt skall kunna följa med i utvecklingen av protokollet samt börssystemet och enkelt kunna generera den önskvärda trafiken samt att kontrollera att börsen utför korrekta handlingar.

Denna testklient är främst avsedd som ett verktyg för felsökning och utveckling av börsen, men skulle kunna komma att vidareutvecklas för problem som ligger utanför utvecklingsdelen, till exempel börsövervakning eller kontroll av insiderhandel.

Målsättning

För att lösa ovan nämnda problem krävs en generell klient som enkelt kan anpassas och automatiseras av användaren på kort tid, samt ger en god och enkel överblick över de meddelanden som börsen svarar med. Målsättningen är därför att konstruera en mycket generell klient som kan kommunicera med börsen och kontrollera svaren. En rad krav och målsättningar på denna klient kan sammanfattas i tre kategorier; funktionalitet, användbarhet och utvecklingsbarhet. Resonemanget bakom de olika målsättningarna beskrivs i detalj i metodkapitlet, de presenteras kort här för att ge en överblick.

Användbarhet

Om verktyget skall brukas för testning behöver den vara enkel att använda och det skall snabbt gå att sätta sig in i hur den fungerar. Instruktioner skall vara enkla och rättfram som tydligt förklarar hur den kan användas med gott om bifogade exempel. Även om klienten skall vara enkel att använda skall den inte vara begränsad i sitt användningsområde, detaljerade ändringar i ett instrument skall kunna ske lika lätt som att generera en miljon köpordrar. Språket som klienten förstår skall vara enkelt och väldokumenterat och följa vanliga standarder. Idealiskt sett skall användaren med begränsad programmeringsvana kunna anpassa klienten till sitt specifika behov utan tidigare erfarenheter med den på en dag. Klienten skall även fungera identiskt eller mycket snarlikt oavsett vilket protokoll som servern pratar.

Funktionalitet
I appendix finns en rad testfall bifogade som klienten skall kunna utföra, utöver det skall arkitekturen vara så pass öppen att det går att manipulera paket ned på lägsta bytenivå, samtidigt som det går att utföra systematiska kontroller och sortera paket på en högre applikations-nivå. Klienten skall vara mycket anpassningsbar och kunna användas i ett stort antal olika situationer och testfall. För att kunna testa serverns kapacitet måste klienten vara extremt snabb, snabbare än servern själv. Om tester begränsas av klientens kapacitet snarare än serverns, bär de föga vikt.

Utvecklingsbarhet
Det är även ett krav att klienten enkelt går att utveckla i takt med börsen, klienten skall vara ett beständigt verktyg även när större förändringar sker i börssystemet och vid protokollförändringar eller byten. Idealiskt sett skall klienten kunna exekveras identiskt med hur de fungerade innan, även om större förändringar sker i systemet eller protokollet

Varför utveckla denna klient?
Enligt Lars Wiktorin (Systemutveckling på 2000-talet) är förvaltningen av ett system den huvudsakliga kostnaden för datorsystem, det är inte ovanligt att den kostnaden är 60% eller högre av den totala kostnaden av systemet. Att effektivisera felsökning och testning är två delar vilket drastiskt kan minska utvecklingskostnaden för ett system. Att effektivisera utvecklingen av ett system ger dessutom en snabbare utvecklingscykel och man får därigenom en konkurrenskraftigare produkt. Ulf Eriksson (Test och kvalitetssäkring av IT-system) menar dock att effektiva testverktyg sällan minskar kostnaderna i utvecklingsfasen utan snarare att betoningen ligger på högre kvalitet och fler tester (s 264). Båda betonar dock vikten av en effektiv testning som en grundsten för ett effektivt systemutvecklande. Mikael Brännström, en av de två utvecklarna från NGM, uppger även han att debugging och testning upptar en huvudsaklig del av hans tid. I vissa fall är ett testverktyg inte bara bra att ha utan direkt nödvändigt, vissa tester är omöjliga att utföra utan ett verktyg för ändamålet. Att utveckla och underhålla ett testverktyg kräver givetvis resurser även det och det måste ställas relativt till de förtjänster det ger, ett testverktyg bör inte utvecklas mer an vad som är nödvändigt för att effektivisera testning och debugging.

Utvecklingsmetod
Övergripande valdes en *lättrörlig (”agil”)* utvecklingsmetod för projektet med en användarcentrerad vinkling, tyngdpunkten läggs på att skapa ett funktionellt och användarvänligt system som faktiskt kommer användas och utvecklas kontinuerligt. Att bygga system med en välgenomtänkt metod och plan, även när utvecklingen bara gäller en person, ger en bra grund för att bygga ett system som blir välfungerande och kan ge en överblick i hur lång tid projektet tar att genomföra.

> *Explicitly “architected” systems seem to turn out faster, better and cheaper.*
> *(IEEE Architecture Working Group)*

Med arkitektur för ett system så syftar det till den övergripande metoden för att modellera och planera ett IT-system i block och hur blocken fungerar tillsammans. En *agil* utvecklingsmetod är *Extrem Programmering*, en metod som i huvudsak baserar sig på 4 värderingar: *kommunikation, enkelhet, återkoppling* och *mod* (Kent Beck). Detta innebär att föra en fortlöpande kommunikation mellan utvecklarna och kunden, så denne kan styra projektet till att uppfylla de mål som finns från början eller upptäcks under utvecklingens gång. *Enkelhet*

fås genom att kontinuerligt refaktorisera koden samt att skapa minimalt med arbete som inte är en del av det slutgiltiga systemet. Ständiga enhetstester och delleveranser av systemet ger en god *återkoppling*. Med mod menas utvecklarens mod att säga ifrån vad som denna inte klarar av att utföra och att ge kunden information om vad som är realistiskt. En av de mer exotiska delarna av extrem programmering är iden om att programmera i par. Tidigare erfarenheter av denna utvecklingsmetod har visat den är tveksamt effektiv, och då examensarbetet utfördes av en ensam utvecklare fick denna del tas bort utan att beprövas.

Kravhanteringsmodell

Initiellt fanns löst definierade önskemål och påvisade brister vilket var ursprunget till beslutet om att skapa en testklient från NGM's sida. Utifrån sitt önskemål om ett bättre testverktyg valdes det att annonsera det som ett examensarbete. Det ingick i examensarbetet att utforma krav, mål och omfattning för klienten. Att finna krav och att dokumentera dem på ett entydigt och begripligt sätt är svårt (s 111, Systemutveckling på 2000-talet) men det ligger till grunden för ett funktionellt och välanpassat verktyg. Det räcker inte att endast samla in och formulera kraven, de måste kunna prioriteras, spåras, ändringshanteras och valideras mot det resulterande systemet. Det finns även en skillnad mellan mål och krav; användaren har en uppfattning om hur systemet skall användas och vad resultatet av denna användning är. Detta uttrycker användaren som en målbild, där målen är ett uttryck för hur användaren vill använda produkten. Dessa mål förädlas och skapar krav, vilka är i nära anknytning till produkten och dess egenskaper snarare än ändamålet produkten skall användas till. Ett krav måste vara formulerat så att det går att verifiera det i slutsystemet och mäta till vilken grad det har uppfyllts. *Institute of Electrical and Electronics Engineers* (IEEE) rekommenderar *Recommended Practice for Architectural Description of Software-intensive Systems* som model för kravhanteringsprocessen som bygger på just denna princip om verifierbarhet.

Att genomföra en sådan väldokumenterad och utarbetad metod gynnar dock inte alltid en ensam utvecklare i samma grad utan det blir till stor del en pappersexercis, och kan flytta fokus från projektet till dokumentation av det. Skaparen av Extrem Programmering, Kent Beck, menar att det är bättre att använda sig av mindre detaljerade användningsfall och frångå kravet på noggrant uppsatta krav och verifierbarhet. Systemet beskrivs då utifrån användarens synvinkel och hela systemets krav byggs av flera ofta informella sådana berättelser. Nackdelen blir då problem för andra att verifiera att ställda krav uppnåddes samt koordinationsproblem i större projekt. Men mer fokus läggs på utveckling och material som faktiskt bygger projektet. Även denna metod har även krediterad effektivitet och accepteras som en väl fungerande metod (Wake, 2000). Eftersom utvecklingen genomfördes av endast en utvecklare och tid var en starkt begränsande resurs valdes därför denna kravhanteringsmodell.

Hur identifierades målen?

Identifieringen av mål, vilka utvecklades till krav, skedde fortlöpande under hela utvecklingstiden. De grundläggande målen fastställdes genom de initiella samtalen med de anställda på NGM (framför allt, Mikael Brännström, Henrik Bergström och John Sjöberg). Återkommande samtal gjorde att nya mål och visioner lades till under projektets hela genomförande i takt med a projektet utvecklades. NGMs väl dokumenterade bugghistorik och lösningar till dessa gav en god insikt i arbetsgången för bugglösning. En bugg eller funktionalitetsbrist löstes vanligtvis i gemensamma ansträngningar, och testresultat och loggar diskuterades i ett forum. Genom att studera arbetsgången blev det tydligt att alla resultat som klienten producerar måste kunna sparas och vara möjliga att lägga ut på forumet.

Flera av kraven lades även till efter inläst litteratur, framför allt de om kostnader och användarvänlighet.

Metod

För att konstruera ett verktyg som blir effektivt, det vill säga effektivt i relation till sin egen utvecklingskostnad, är arbetet planerat i tre dimensioner; funktionalitet, användbarhet och utvecklingsbarhet. Verktyget måste uppfylla de situationer där ett verktyg är ett måste, och det måste vara tillräckligt omfattande för att kunna användas vid en större del av testningen (funktionalitet) för att vara så pass heltäckande som möjligt. Det måste även vara så pass användarvänligt att det faktiskt används, samt att inlärningskostnaden för nya användare hålls så låg som möjlig (användbarhet). En tredje aspekt på verktyget är att planera arbetet så testverktygets egenkostnad vid utveckling blir så låg som möjlig samt att det kan användas en längre tid (utvecklingsbarhet). Målsättningarna blev följaktligen:

Funktionalitet

Klienten sitter mellan användaren och servern och har därför två roller; dels mot användaren, dels mot servern. Funktionalitetskraven mot servern (*back-end*) är följande:

Klienten skall kunna skicka alla paket som finns specificerade i protokollen som den kommunicerar med börsen. Samtliga fält i alla inkomna paket skall enkelt gå att extrahera och verifiera, samt alla fält i utgående paket skall kunna sättas. Klienten skall kunna ta emot och skapa alla paket som finns specificerade i protokollet. Vidare skall klienten kunna öppna, stänga och hålla kanaler vid liv på ett korrekt sett. Den måste implementera en extremt snabb paketmotor, idealiskt sett snabbare än serverns paketmotor, detta för att kunna utföra belastningstester. Användaren skall kunna sätta alla fält utan att veta detaljerad information om hur paketet ser ut (hur flyttal representeras, eller om paketet är i binär eller textformat etc).

För front-end finns följande funktionalitetskrav:
Användaren skall kunna kommunicera med olika servrar utan att behöva göra ändringar i sitt användande oavsett vilket protokoll som används i back-end. Omställningen från att olika versioner av protokoll skall vara minimal. Det skall gå att analysera paketen på logiskt och fysisk nivå, det vill säga vad det är för värde på en variabel, samt hur paketen ser ut i rådata. Det skall finnas ett filtreringssystem som kan filtrera bort ointressanta paket och endast lyfta fram de intressanta. Filtrering skall vara möjlig på både paketnivå samt fältnivå. Vidare skall användaren kunna spara och ladda sparade paket samt att skriva ut dem till fil eller på ett presenterbart sätt. Det bör även finnas möjlighet att skapa en bild av hur marknaden ser ut på logisk nivå med instrument, sammanhörande ordrar, orderdjup, levels etc. Dessa logiska representationer av hur marknaden ser ut skall gå att spara och ladda från fil.

Användbarhet

För att klienten skall bli lyckad behöver stor vikt läggas på att den blir användarvänlig. Att lära sig verktyget skall vara enkelt och ta minimalt med tid, fullständig förståelse av det skall för en van programmerare inte ta mer än högst ett par timmar.

> *”Många verktyg blir hyllvärmare, det vill säga företaget slutar att använda verktyget efter en kortare eller längre tid. En anledning till det är att testverktyg ofta är personberoende. När den som brinner för testverktyget slutar på företaget, slutar företaget att använda verktyget. Nackdelarna som är förknippade med*

> *verktyg är sällan av teknisk karaktär, det är snarare organisatoriska. Det är inte tekniskt avancerat att skriva testprogram eller att använda ett verktyg för prestandatest, problemet är snarare vem som skall göra det i organisationen." Ulf Eriksson (261)*

För att verktyget inte skall dö ut om den som brinner för verktyget slutar behöver all nödvändig information om hur verktyget skall användas finnas tillgänglig om någon vill ta upp användningen av det, likaså är det nödvändigt att fler än en är välbekant med verktyget. Hur verktyget införs i organisationen är således viktig för om verktyget kommer uppfattas som användarvänligt eller inte. Om användarna inte kommer i kontakt med det spelar det ingen roll hur användarvänligt det är.

En viktig komponent för användarvänlighet är verktygets gränssnitt. Enligt Joshua Bloch (How to Design a Good API & Why it Matters) är gränssnittet det mest fundamentala för om en produkt blir lyckad eller inte. Det går heller inte att göra större ändringar i det; det skulle göra att användarna ogillar produkten. Man måste därmed lyckas redan på första försöket.

Den andra viktiga komponenten för att bygga ett användarsystem är instruktionerna runt gränssnittet. För att användaren snabbt skall sätta sig in i hur systemet kan användas bör det finns gott om exempel och förklaringar steg för steg hur de fungerar. Användaren kan sedan använda sig av API för att hitta detaljerad information. Exemplen kan därför fungera som en introduktion, där modifieringar eller genomgång av dessa kan leda användaren vidare i användandet av systemet.

En annan del är tillgången till hjälp och var den presenteras och introduceras. Alla utvecklare och systemtestare på NGM har en viss grad av programmeringsvana, kunskap om vad skript är och hur de används. Sådan grundläggande information behöver därför inte finnas för detta projekts användargrupp. Men kunskapsnivåerna skiljer sig ändå, från mediokra till expertnivå. Det behöver således finnas anpassat stöd för flera kunskapsnivåer; från handböcker som beskriver hur klienten kan användas till specificerade referenser där detaljerad kunskap enkelt kan hittas. Både handböcker och exempel bör därför vara upplagt på ett sätt som användarna är vana med, och finnas på flera olika nivåer, från introduktion till mer avancerade fall, där användaren direkt kan börja läsa på den nivå denna känner är lagom. Presentationen av informationen, både från grundnivå till expertnivå måste vara enkelt att hitta och använda, den bör därför finnas lättillgänglig för alla eventuella användare och hållas kontinuerligt uppdaterad.

Utvecklingsbarhet

En mycket viktig aspekt av klienten är att den bör vara utvecklingsbar, både i front-end och back-end. Det kommer att ske omfattande förändringar i hur miljön som klienten opererar i ser ut, men verktuget måste följa med och enkelt kunna anpassas. Verktyget bör, trots att utveckling av det sker, inte behöva ändra sin grundläggande arkitektur.

> *"Ett ytterligare problem med automatiserade tester är att de måste vara skrivna på så sätt att de går att återanvända utan att omfattande ändringar behöver göras varje gång det testade systemet ändras. Eftersom testprogrammen ska kunna användas under hela systemets livslängd, är det viktigt att det är dokumenterade så att det går att underhålla programmen." Ulf Eriksson (264).*

Utvecklingen bör även vara enkel att genomföra och dokumentationen av hur sådan utveckling bör ske bör vara väldokumenterad så för att sätta sig in i arbetet och genomföra detta skall vara enkelt, även om personen inte deltog i den initiella utvecklingen av klienten.

Designval

Eftersom testfallen skiljer sig mycket åt, från att använda klienten för att simulera överbelastning till att kontrollera att paketen har rätt struktur, blev det uppenbart att en färdig klient som kunde ta hand om alla tänkbara fall var en omöjlighet med given tid och resurser. Istället behövs det ett bibliotek av metoder och verktyg som sedan kan användas för att producera preciserade applikationer specifikt anpassade för det testfallet. Efter en analys av vilka som skall komma använda programmet framkom det snabbt att både testare och utvecklingspersonal alla är väl bevandrade i programmering.

Slutprodukten kommer således bli ett bibliotek med metoder och hjälpklasser, allt från en motor att generera meddelanden till analysverktyg för att verifiera marknadsdata. Användaren binder sedan ihop de olika metoderna med skript eller använder dem som klassbibliotek i program de själva skriver.

Valet att göra ett bibliotek istället för en färdig applikation baserar sig på uppgifternas diversitet, användarnas kompetensnivå och tidigare erfarenheter av testverktyg på företaget. Eftersom det är omöjligt att förutse alla möjliga buggar skulle en komplett klient som kan lösa alla tänkbara buggar kräva enorma resurser, att utveckla en så komplett klient i takt med handelssystemet utveckling skulle vara ekonomiskt oförsvarbart.

Det andra alternativet är att utveckla klienten efter det att en bugg uppstått och det var så tidigare testverktyg fungerade. Då de tidigare testverktygen inte var speciellt anpassningsbara krävde varje ny bugg eller test ofta en ändring djup in i källkoden på testverktyget, vilket krävde en god inblick i hur det fungerade. De hade heller inte stöd för flera protokoll och hade således inget gemensamt interface för större handelssystemändringar, som t.ex. protokoll byte eller tillägg. Testverktygen var inte utvecklingsvänliga vare sig i sin arkitektur eller dokumentation, men verktygen har fungerat hjälpligt.

Eftersom programmeringsnivån är mycket god hos samtliga användare lades därför fokus på att skapa ett så komplett bibliotek som möjligt, där användaren kan plocka de delar denna önskar till sin specifika applikation. Användaren skall kunna kommunicera på fler olika nivåer med biblioteket, från bytenivå till en överblick över hela handelssystemet utan att behöva göra modifieringar i biblioteket. För att minimera tiden det tar att producera specialiserade verktyg är biblioteket skapat i lager, där användaren kan använda väldigt enkla anrop för att göra vanliga handlingar på börsen, likväl förmågan att göra specialanpassade paket att skicka till handelssystemet ner på byte-nivå.

Detta verktyg har således inget klassiskt användargränssnitt i och med att det till stor del skapas av användaren själv, det som dock används som användargränssnitt är API (*Application Programming Interface*) vilket i detalj beskriver vilka metoder och klasser som finns att tillgå. Ett bra API beskriver i detalj hur metoderna fungerar, vad de returnerar och en noggrann genomgång av deras anrops- och returneringsvariabler. Ett användarvänligt API bygger inte enbart på om informationen presenteras noggrant och korrekt; informationen måste också vara logisk och lättförståelig. Klasserna och dess metoder måste därför vara logiskt sammankopplade och namngivna så att de är enkla att förstå. Verktyget bör således

vara uppbyggt på ett sätt som är logiskt, lättöverskådligt och enkelt att sätta sig in i. Detta påverkar i allra högsta grad hur verktyget arkitektur ser ut, den måste alltså inte bara vara välfungerande och mycket snabbt (för att uppfylla funktionalitet), den måste även vara logisk och lättöverskådligt för en utomstående att sätta sig in i. Vidare skall verktyget vara konstruerat så det kan tillgodose utvecklingar och ändringar, utan att ändringar i uppbyggnaden av verktyget behövs, utan endast påbyggnader. Arkitekturen av hur systemet är uppbyggt är alltså i största grad en avgörande faktor om systemet blir lyckat eller inte.

Valet av programmeringsspråk av denna klient baserades på användarnas kompetens samt möjligheten att återanvända redan konstruerade standardbibliotek. Java är det språk som i huvudsak används på NGM och att använda ett språk som samtliga utvecklare är väl bevandrade i leder till en lägre utvecklingskostnad. Att använda ett språk som merparten av utvecklarna och testarna på NGM är välbekanta med kommer att minska motviljan att sätta sig in i koden för användning och utveckling. Det minskar även tiden det tar att komma igång med användandet av klienten för systemtestarna då de inte behöver lära sig ett nytt programmeringsspråk. Om standarder för namngivning, klasshantering och dokumentation även följer namnkonventionerna på hur Java och dess dokumentation skall se ut (framtagna av Sun Developer Network) ger det en familjär miljö för utvecklare och anställda på NGM, vilka alla är vana vid Java-dokumentation. Valet av Java som programmeringsspråk blev därför självklar, liksom att följa de rekommendationer och standarder som finns. Vidare är biblioteket anpassat så det är kompatibelt med Javas standardbibliotek, vilket gör det enkelt att använda dessa för tidtagning eller att rita upp tabeller grafiskt.

Testarna ville gärna ha möjlighet till att använda sig av skript då det skulle vara enkelt och snabbt att modifiera dem samt att de inte behöver kompileras. Att använda sig av skript gör också att skripten enkelt kan postas i buggforumet där lösningar och resultat diskuteras för att lösa problem i handelssystemet. Då kan skripten enkelt finnas till hand för verifiering av resultaten och kod återvinnas när nya skript produceras. Det överlämnar även problemet med plattformsoberoende på klienten, många utvecklare använder sig av olika versioner av Java eller andra specialbibliotek, vilket kan skapa problem om samma kompilerade program körs på olika maskiner. Om man inte implementerar en skriptmotor utan låter användaren kompilera sitt program gör det dock programmet snabbare, samt att skriva mer komplicerade fall blir enklare, vilket med skript kan bli svårt och otympligt om det kräver en mer arkitekturerad lösning uppdelad i flera klasser och objekt.

En noggrann jämförelse genomfördes mellan olika skriptmotorer (alternativen som utforskades var; Jacl, Jython, Rhino, JRuby, Beanshell, Groovy, Pnuts, JudoScript, JLUA). Noggranna prestandatester visade enligt *David Kears* att Rhino var den snabbaste skriptmotorn. Egna tester och implementationer av Jacl, Jruby och Rhino verifierade Kears resultat. Dessutom är JavaScript (vilket Rhino använder sig av) välbekant på NGM , samt är väldokumenterat. Det har även många bra handböcker samt funktionalitet att importera Javas alla bibliotek vilket ger ett stort metodbibliotek. GTC använder sig således av Rhino.

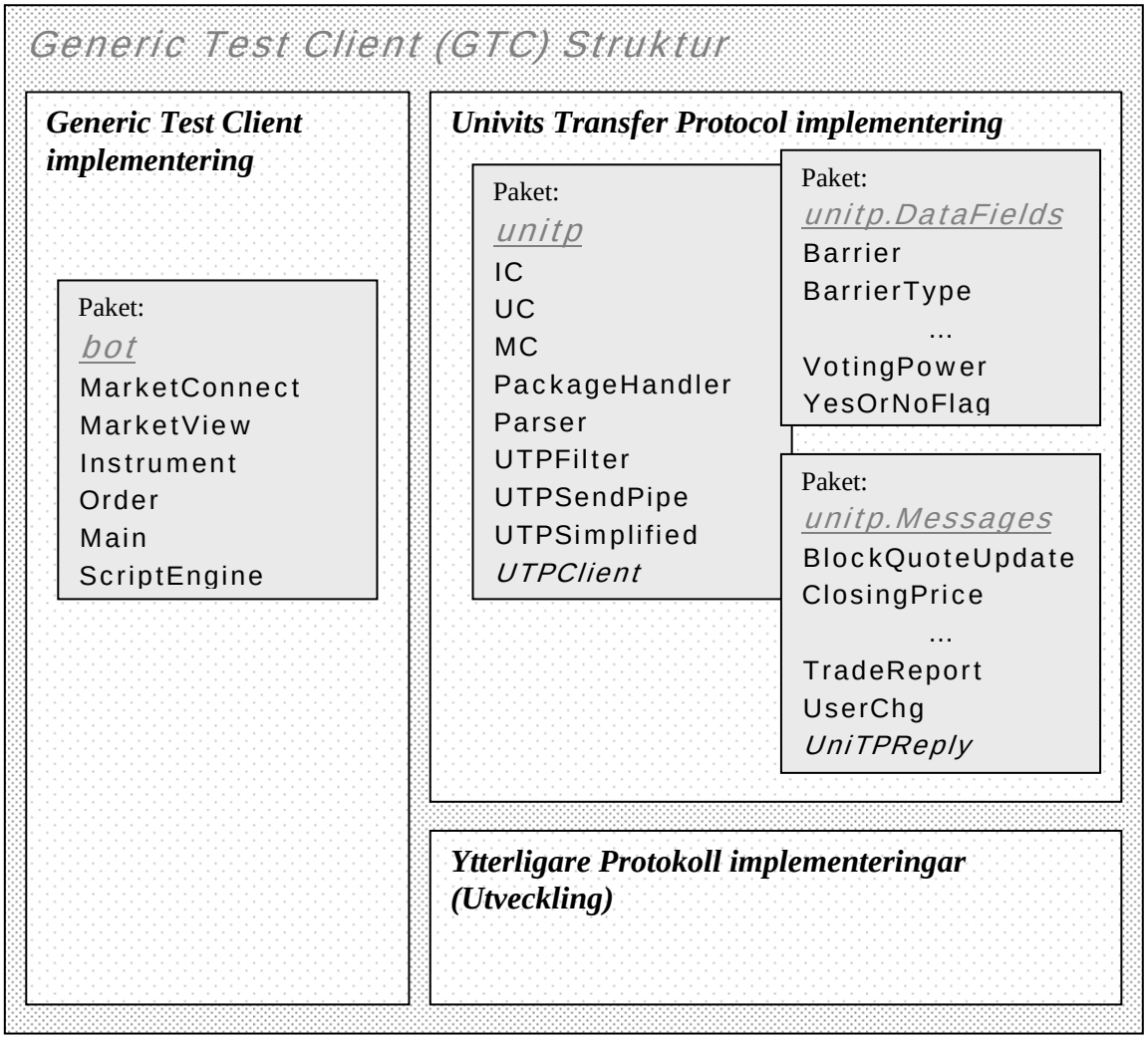
Valet mellan att skapa ett bibliotek som användaren importerar som användaren importerar till sin applikation eller om klienten drivs av en skriptmotor, där användaren kan skriva skript vilka befinner sig i en kontext som kan anroppa metoder i biblioteket blev lätt. Efterforskningar visade att man inte behövde välja det ena eller andra, Rhino är en skriptmotor till Javaskript i vilken man kan importera Java-bibliotek. De som vill använda biblioteken för att skriva en fullständig applikation kan göra så, och den som vill använda

Rhino som skriptmotor, och importera biblioteken till skriptet kan göra det. Detta utan att det begränsar det ena eller andra det minsta.

Hur klienten fungerar

Det finns två huvudsakliga komponenter i klienten; front-end och back-end. Front-end är den del användaren i huvudsak använder metoder och klasser ur. Det är i denna del som det finns funktioner att skapa uppkopplingar till handelssystemet, sortera och analysera inkomna paket samt spara och ladda data till fil. Det finns bara ett front-end och detta kommer stå sig likt med mindre modifieringar när nya protokoll implementeras. Back-end översätter anropen gjorda till front-end till det protokoll handelssystemet använder. I back-end är allt protokollspecifikt och ett nytt back-end måste läggas till för varje nytt protokoll som implementeras. De två delarna är särskiljda åt så de enkelt kan modifieras utan att stora ändringar behöver ske i andra delen, om ett nytt protokoll skall implementeras eller specifikationen för ett redan implementerat paket ändras så kan back-end ändras medan användaren fortfarande arbetar mot samma interface i front-end. Det vill säga metoder för att skicka kommando till börsen ser exakt likadana ut på användarnivå, även om ändringar har gjorts på back-end. Likaså kan funktionalitet läggas till i front-end utan att det behöver modifieras något i back-end delarna.

Front-end paketet kallas för bot, och back-end kallas är uppkallade efter de protokoll de implementerar. I dagsläget finns det endast ett protokoll implementerat; Unitp.



Botpaketets centrala delar är MarketConnect och MarketView där MarketConnect är användarens anslutning till marknaden medan MarketView är en databas som sparar och presenterar innehållet på börsen. Den egentliga trafiken kontrolleras av MarketConnect, och MarketView läggs till som en observatör av denna och uppdateras så fort något kommer in och skickas vidare av MarketConnect. En MarketConnect kan ha flera observatörer och en MarketView kan observera flera MarketConnect.

Alla delar i bot (front end) strävar mot att vara protokolloberoende, av tekniska anledningar är detta dock i vissa fall omöjligt. Till exempel kräver UniTP mer information (flera inloggningsnamn/lösenords-par) än andra protokoll för att etablera en uppkoppling mot en server. Nästan alla metoder som klasserna i bot paketet är protokolloberoende, och de som inte är det är tydligt markerade.

Övergripande kommunicerar användaren med bot paketet, som i sin tur kommunicerar med backend (protokollimplementationerna), som sedermera kommunicerar med börsen. Det finns även en rad verktyg för lagring, inladdning och analysverktyg som finns tillgängligt i bot paketet. Det går även bra att ladda in utomstående paket, till exempel matematiska paket, eller grafiska paket för att göra grafiska interface, tack vare importeringsmöjligheterna. För en detaljerad beskrivning av hur de olika paketen interagerar samt hur dess metoder fungerar hänvisas till handboken samt referensbiblioteket (bifogade).

Ett exemepl på ett skript och hur resultatet ser ut. Notera att detta skript är förenklat och att det borde finnas kod för att ta hand om eventuella fel som kan uppstå (tex om man försöker läsa in en fil som inte finns).

Detta är ett exempel för handel kan se ut. Ofta är skripten kortare, då de vanligtvis loopar igenom en kort kodbit vid överbelastningstest eller bara skickar ett par paket efter inloggning och kontrollerar att svaret är korrekt. Skripten kan givetvis vara längre också, men då bör nog alternativet att skapa en applikation som kompileras övervägas.

Modifiering av marknadsdata

```
// Här börjar vi med att importera de delar vi vill använda, notera att några är Javas
// standardbibliotek. importPackage importerar alla klasserna i ett paket.

importPackage(java.awt);
importPackage(java.lang);
importPackage(Packages.UniTP);
importPackage(Packages.UniTP.Messages);
importPackage(Packages.UniTP.DataFields);
importPackage(Packages.bot);

// Här skapar vi en marknadsvy som hanterar alla inkomna paket, vi skapar sedan en
// anslutning till marknaden och meddelar vår marknadsvy att observera denna .
// Klienten loggar sedan in till handelssystemet och vi läser in marknadsdata
// från en sparad fil (från en tidigare session.)
```

```
marketview = new MarketView();
market = new MarketConnect("sv-dev-02.x.ngm.se", "239.255.0.10",  11910, 1);
market.addObserver(marketview);
marketview.setVerbose(3);
market.login("ORG1","ABC1","ABC1","ABC2","ABC2","ORG1-M1");
marketview.readFromFile("saveddata");

// Här hämtas alla instrument ut ur  marknadsvyn och vi lägger dem i en lista vilken
// vi sedan itererar igenom. För de 5 första instrumenten försöker vi lägga in en
// köporder och en säljorder  på ett slumpat pris. Vi skriver även ut att vi försöker göra
// detta till textprompten för att meddela användaren att ett försök gjorts.

stocklist = marketview.getStocks();
index = 1;
for (i = 5; i > 0 ; i--) {
        price = Math.round((Math.random()*10) + 91);            market.printLine("["
+ index + "]\t" + stocklist.get(i-1).getName()
                        + " - B 1000 @" + price
                        + " -> *ATTEMPTING INSERT*");
        market.buy(stocklist.get(i-1), price, 1000);
        index++;
        price = Math.round((Math.random()*10) + 71);            market.printLine("["
+ index + "]\t" + stocklist.get(i-1).getName()
                         + " - S 1000 @" + price
                         + " -> *ATTEMPTING INSERT*");
        market.sell(stocklist.get(i-1), price, 1000);
        index++;
}

// För de 5 nästkommande instrumenten försöker vi modifiera volymen på redan
//existerande ordrar och meddelar användaren om detta.

for (i = 10; i > 5 ; i--) {
        if (stocklist.get(i-1).getOrders().size() > 0) {
        market.printLine("[" + index + "]\t"
                        + stocklist.get(i-1).getOrders().get(0).getId() + " - "
                        + stocklist.get(i-1).getOrders().get(0).getBuySell() + " "
                        + stocklist.get(i-1).getOrders().get(0).getVolume()
                        + " @" + stocklist.get(i-1).getOrders().get(0).getPrice()
                        + " -> * INC. VOL. 1000*");
                       market.modifyOrderVolume(stocklist.get(i-1),
                         stocklist.get(i-1).getOrders().get(0), 1000, 0);
        index++;
        } else {
                       market.printLine("There are no Buy orders that you can
                         modify in " + stocklist.get(i-1).getName());
        }
}

// På de 5 instrumenten efter detta tar vi bort den första ordern vi kan finna och skriver
// även ut detta på textprompten.
```

```
for (i = 15; i > 10 ; i--) {
        if (stocklist.get(i-1).getOrders().size() > 0) {
                        market.printLine("[" + index + "]\t"
                          + stocklist.get(i-1).getOrders().get(0).getId() + " - "
                          + stocklist.get(i-1).getOrders().get(0).getBuySell() + " "
                          + stocklist.get(i-1).getOrders().get(0).getVolume()
                          + " @" + stocklist.get(i-1).getOrders().get(0).getPrice()
                          +" -> *ATTEMPTING DELETE*");
        market.removeOrder(stocklist.get(i-1),
                          stocklist.get(i-1).getOrders().get(0));
                        index++;
        } else {
                        market.printLine("There are no Buy orders that you can
                          delete in " + stocklist.get(i-1).getName());
        }
}

// Vi inväntar sedan svar från handelssystemet, skriver ut svaren och loggar ut.

market.waitOnCatchup();
Thread.sleep(500);
marketview.printReplies(1);
market.logoff();
```

Resultaten av att köra detta skript kan se ut på följande sätt:

Resultat av modifiering av marknadsdata

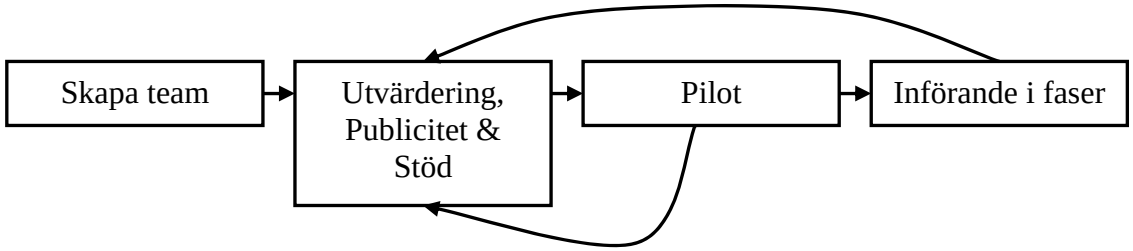
```
[1]     A102 - B 1000 @100 -> *ATTEMPTING INSERT*
[2]     A102 - S 1000 @74 -> *ATTEMPTING INSERT*
[3]     A101 - B 1000 @99 -> *ATTEMPTING INSERT*
[4]     A101 - S 1000 @78 -> *ATTEMPTING INSERT*
[5]     A100 - B 1000 @96 -> *ATTEMPTING INSERT*
[6]     A100 - S 1000 @71 -> *ATTEMPTING INSERT*
[7]     A10 - B 1000 @97 -> *ATTEMPTING INSERT*
[8]     A10 - S 1000 @72 -> *ATTEMPTING INSERT*
[9]     A1 - B 1000 @94 -> *ATTEMPTING INSERT*
[10]    A1 - S 1000 @77 -> *ATTEMPTING INSERT*
[11]    20070911133316849428 - B 1000 @85.0 -> *INC. VOL. 1000*
[12]    20070911133316851430 - B 1000 @84.0 -> *INC. VOL. 1000*
[13]    20070911133316854432 - B 1000 @84.0 -> *INC. VOL. 1000*
[14]    20070911133316856434 - B 1000 @91.0 -> *INC. VOL. 1000*
[15]    20070911133316859436 - B 1000 @87.0 -> *INC. VOL. 1000*
[16]    20070911133316837418 - B 1000 @84.0 -> *ATTEMPTING DELETE*
[17]    20070911133316840420 - B 1000 @83.0 -> *ATTEMPTING DELETE*
[18]    20070911133316842422 - B 1000 @89.0 -> *ATTEMPTING DELETE*
[19]    20070911133316845424 - B 1000 @87.0 -> *ATTEMPTING DELETE*
[20]    20070911133316847426 - B 1000 @83.0 -> *ATTEMPTING DELETE*

[Order Insert:  B A102  (20070911134752217426) 1000 @ 100.0]
[Order Insert:  S A102  (20070911134752225427) 1000 @ 74.0]
[Order Deleted: S A102  (20070911134752225427) Reason: 1]
[Order Deleted: B A102  (20070911134752217426) Reason: 1]
[Trade in:      S A102  (20070911134752225427) 1000 @ 100.0]
[Trade in:      B A102  (20070911134752217426) 1000 @ 100.0]
[Order Insert:  B A101  (20070911134752228428) 1000 @ 99.0]
[Order Insert:  S A101  (20070911134752228429) 1000 @ 78.0]
[Order Deleted: B A101  (20070911134752228428) Reason: 1]
[Order Deleted: S A101  (20070911134752228429) Reason: 1]
[Trade in:      B A101  (20070911134752228428) 1000 @ 98.0]
[Trade in:      S A101  (20070911134752228429) 1000 @ 98.0]
[Order Insert:  B A100  (20070911134752229430) 1000 @ 96.0]
[Order Insert:  S A100  (20070911134752229431) 1000 @ 71.0]
[Order Deleted: B A100  (20070911134752229430) Reason: 1]
[Order Deleted: S A100  (20070911134752229431) Reason: 1]
[Trade in:      S A100  (20070911134752229431) 1000 @ 96.0]
[Trade in:      B A100  (20070911134752229430) 1000 @ 96.0]
[Order Insert:  B A10   (20070911134752231433) 1000 @ 97.0]
[Order Insert:  S A10   (20070911134752236434) 1000 @ 72.0]
[Order Deleted: S A10   (20070911134752236434) Reason: 1]
[Order Deleted: B A10   (20070911134752231433) Reason: 1]
[Trade in:      S A10   (20070911134752236434) 1000 @ 97.0]
[Trade in:      B A10   (20070911134752231433) 1000 @ 97.0]
[Order Insert:  B A1    (20070911134752265436) 1000 @ 94.0]
[Order Insert:  S A1    (20070911134752274439) 1000 @ 77.0]
[Order Deleted: S A1    (20070911134752274439) Reason: 1]
[Order Deleted: B A1    (20070911134752265436) Reason: 1]
[Trade in:      S A1    (20070911134752274439) 1000 @ 94.0]
[Trade in:      B A1    (20070911134752265436) 1000 @ 94.0]
[Order Update:  B A107  (20070911133316849428) 2000 @ 85.0]
[Order Update:  B A106  (20070911133316851430) 2000 @ 84.0]
[Order Update:  B A105  (20070911133316854432) 2000 @ 84.0]
[Order Update:  B A104  (20070911133316856434) 2000 @ 91.0]
[Order Update:  B A103  (20070911133316859436) 2000 @ 87.0]
[Order Deleted: B A111  (20070911133316837418) Reason: 2]
[Order Deleted: B A110  (20070911133316840420) Reason: 2]
[Order Deleted: B A11   (20070911133316842422) Reason: 2]
[Order Deleted: B A109  (20070911133316845424) Reason: 2]
[Order Deleted: B A108  (20070911133316847426) Reason: 2]
```

Klienten har även möjligheten att filtrera paket med hjälp av ett filter vilket filtrerar paket, både på paketnivå och på fältnivå i paket. På så vis kan man t.ex. lyssna på marknadsdata och skriva ut alla avslut, eller analysera trafik och filtera ut alla felmeddelanden. För fler exemepl på hur filter kan användas hänvisas till användarhandboken.

Införandet av verktyget i Organisationen.

Enligt Ulf Eriksson (Test och kvalitetssäkring av IT-system) listar 4 viktiga steg för införandet av ett testverktyg i en organisation:



Det första steget är att ta fram en grupp som ansvarar för införandet av verktyget, ofta samma personer som ansvarade för kravspecifikationen av det. Efter det behöver det utvärderas och få ett stöd hos ledning för att sedan publiceras och skapa ett intresse hos användarna av det. Verktyget sätts in hos en pilotgrupp, som får lära sig att använda det, varefter en pilotutvärdering sker. Om det visar sig troligt att verktyget fungerar bra och det får stöd, tas det i bruk stegvis i organisationen och en kontinuerlig utvärdering sker vid varje införande så inte samma misstag sker om och om igen vid införandet. Det är viktigt att införandet sker i ett lagom tempo, det får inte ske överilat, men det får heller inte gå segt. Introduktionen, inlärningen och användandet måste följa tätt inpå varandra.

Utvecklingen skedde i kontinuerligt samarbete med utvecklingsavdelningen på NGM, en konstant dialog följde hur verktyget skulle se ut och flera på utveckling och testavdelningen gav många bra idéer om hur verktyget kan komma fungera. Att de var en del av utvecklingen var ett första och mycket viktigt steg i införandet av verktyget i organisationen då det skapade ett intresse för verktyget samt att arbetet med utvecklingen gjorde utvecklingsavdelningen positivt inställda till att arbete med verktyget. När verktyget började ta form och det fanns en fungerande demoversion bjöds test, utveckling och supporten in för att prova på vid en demonstration där en kort presentation även hölls. Intresse för verktyget tag fart ordentligt och det gavs möjlighet att komma med förslag på hur det skulle kunna utvecklas vidare.

Dokumentationen lades tillgänglig på NGMs intranät, och skrevs i Javadoc format, vilket är ett standardiserat och bekräftat bra sätt att presentera API. För introduktion och lite kom-i-gång hjälp finns en handbok skriven vilket enkelt ger användaren en introduktion till testverktyget. Det finns även en rad dokumenterade och förklarande exempel vilka kan användas som grundstomme till de mer specificerade skripten. Själva verktyget finns färdigkompilerat att ladda ner på intranätet, samt att ladda ner från CVS. Vid efterfrågningar ansåg sig alla ha lätt att hitta verktyget, samt information om hur det används.

Resultat - de tre aspekterna

Funktionalitet

Klienten implementerar samtliga paket i UniTP protokollet och det erbjuder ett interface till användaren där merparten av funktionerna är oberoende av det bakomliggande protokollet. Tyvärr går det inte att göra skripten helt oberoende av protokollet som klienten kommunicerar

med servern. Vissa funktioner skiljer sig så vitt att de kommer att vara omöjliga att implementera som en gemensam metod, exempelvis kräver login anropen i UniTP två användarnamn och två lösenord, medan andra protokoll enbart kräver ett sådant par. I kravspecifikationen togs det fram ett antal testfall (dessa finns i bilagsdelen) vilka samtliga uppfylldes.

Vid belastningstest visade det sig att klientens pakethantering samt multitrådningen av dessa fungerade utmärkt, vid 5% CPU använding på en pentium 1000 MHz kunde klienten överbelasta test-servern (vilken har betydligt sämre prestanda än live-servrarna bör tilläggas). Då testning av klientens fulla kapacitet blir svår att genomföra, och den är till stor del beroende av vilka funktioner (exempelvis databaslagring samt utskrivning av data till *standard out* kräver förhållandevis mycket resurser) borde den klara av en orderhantering i storleken 100 000 ordrar i sekunden.

Klienten implementerar ett filter vilket dynamiskt kan skapas i xml. I filtret matchas alla inkomna paket i *PackageHandler* innan de behandlas av övriga delar. Då PackageHandlar kan lyssna på en annan PackageHandler går det att specificera vilka paket som behandlas på vilket sätt och hur de skickas vidare i en trädstruktur. Exempelvis kan felmeddelanden komma till en kanal, medan viss data lagras och annan kasseras. För närmare exempel om detta hänvisas till användarhandboken samt exempelskripten.

Testklientens möjligheter sträcker sig dock långt utöver de givna testfallen. Av vikt att nämna är möjligheten att kontrollera paketlängd, paketfält och att mäta responstider. Utvecklare, support och testare är samtliga nöjda med funktionaliteten i klienten.

Användbarhet

Stor vikt lades på att göra klienten användarvänlig. Klienten är avsedd för expertnyttjande och fokus har därför lagts på dokumentation och en intuitiv design för att snabbt kunna sätta sig in i användandet av klienten. Responsen från användarna har varit mycket positiv, och samtliga har enkelt lyckats komma igång själva med hjälp av dokumentationen som fanns. Intressant nog ville även en användare utan programmeringskunskaper använda klienten och skriva skript. Med hjälp av användarhandboken och exemplen lyckades denne utmärkt. Det är för tidigt att se vilken genomslagskraft verktyget har i organisationen och hur mycket verktyget kommer användas, men det har mottagits varmt av flera. Då användandet tog fart omedelbart av flera användare är det troligt att användandet kommer vara fortsatt använt även efter verktygets första introduktion och inte vara beroende av en eldsjäl.

Bilagor

Testfall

4 sidor testfall vilka kan verifieras med hjälp av skript och bibliotek vilka var en delmängd av kravspecifikationen.

Användarhandbok

26 sidor användarhandbok, skriven som komplement till Javadoc som hjälp för att komma igång med användandet av skriptning och användandet av biblioteket.

Programkod

248 java filer, totalt något över 20 000 eget skrina rader kod vilket är biblioteket som används av skriptmotorn. På grund av sin storlek kan dessa dokument ej bifogas utskrivet, utan finns tillgängliga i bifogad zipfil samt på http://caesar.honorbound.se/xjobb

Javadoc API

547 html dokument vilka presenterar bibliotekets API. På grund av sin storlek kan dessa dokument ej bifogas utskrivet, utan finns tillgängliga i bifogad zipfil samt på http://caesar.honorbound.se/xjobb

Exempel

13 javaskript exempel samt 2 xml filter som exempel på hur skript kan se ut och fungera. På grund av sin storlek kan dessa dokument ej bifogas utskrivet, utan finns tillgängliga i bifogad zipfil samt på http://caesar.honorbound.se/xjobb

Källhänvisningar

David Kearns, JavaWorld.com, Choosing a Java scripting language: Round two, 2005, (http://www.javaworld.com/javaworld/jw-03-2005/jw-0314-scripting.html?page=1)
Jan Gulliksen & Bengt Göransson, Användarcentrerad Systemdesign, Studentlitteratur, 2006
John Sjöberg, Univits Transfer Protocol Specification, Nordic Growth Market, 2007
Joshua Bloch, How to Design a Good API & Why it Matters, 2007, (http://www.infoq.com/presentations/effective-api-design)
Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 2000
Lars Wiktorin, Systemutveckling på 2000-talet, Studentlitteratur, 2007
Ulf Eriksson, Test och Kvalitetssäkring av IT-system, Studentlitteratur, 2004

Övriga referenser

Sun Developer Network, Code Conventions for the Java Programming Language, (http://java.sun.com/docs/codeconv/)

Testfall

SYNTAX

Order (a) köp 1RL@4TICK betyder, order med id/ref (a) som är en köp-order med volymen 1RL = 1 ggr round lot size och priset 4 ggr tick size.

TESTFALL

RL = Round lot size (volym). TICK = tick size (pris).

Marknadens status är CONTINOUS_TRADING om inget annat är angett.

Orderhantering:
---

1. Skapa order och ta bort order 1.
 * Lägg in en order (a) köp 5RL@7TICK.
 * Förväntat resultat:
  - Order (a) har lagts in, både som privat svar till klienten samt som marknadsdata till alla klienter.
 * Ta bort order (a).
 * Förväntat resultat:
  - Order (a) har tagits bort, både som privat svar till klienten samt som marknadsdata till alla klienter.

2. Modifiera order 1.
 * Lägg in en order (a) köp 2RL@4TICK.
 * Förväntat resultat:
  - Order (a) har lagts in, både som privat svar till klienten samt som marknadsdata till alla klienter.
 * Sätt fältet client_ref till "abc".
 * Förväntat resultat:
  - Client_ref i order (a) har satts till "abc", samt övriga fält är oförändrade (endast last_modified el motsv är ändrad).
 * Ändra alla fält utom pris och volym, ett i taget och verifiera att så sker.

Korrekta levlar:
---

1. Levlar 1.
 * Lägg in ett par ordrar
 * Verifiera att levlarna i marknadsdatat är korrekta.

2. Levlar 2.
 * Lägg in ett par ordrar i samma orderbok.
 * Verifiera att levlarna i marknadsdatat är korrekta.
 * Modifiera ngt oviktigt fält en order, ex client_ref.
 * Verifiera att levlarna är oförändrade
 * Modifiera volymen nedåt i en order
 * Verifiera att levlarna är oförändrade, förutom att volymen har minskat i en order.
 * Modifiera volymen uppåt i en order
 * Verifiera att levlarna har uppdaterats (och är korrekta).
 * Modifiera pris i en order.
 * Verifiera att levlarna har uppdaterats (och är korrekta).
 * Ta bort den bästa köp-ordern.
 * Verifiera att levlarna har uppdaterats (och är korrekta).

Handelsregler i continous trading:

---

1. Direktmatch 1
  * Org 1 lägger order (a) sälj 2RL@1TICK
  * Org 1 lägger order (b) köp 1RL@2TICK
  * Förväntat svar:
    - Avslut mellan (a) och (b) 1RL@1TICK
    - Order (a) ligger kvar med volym 1RL, order (b) finns ej kvar.

2. Direktmatch 2
  * Org 1 lägger order (a) köp 2RL@7TICK
  * Org 2 lägger order (b) sälj 3RL@7TICK
  * Förväntat svar:
    - Avslut mellan order (a) och (b) 2RL@7TICK
    - Order (a) tas bort, order (b) är kvar med volym 3RL

3. Lilla orderboken 1
  * Gör ett internavslut med org 1, 1RL@4TICK (lägg in 2 ordrar som matchas)
  * Org 1 lägger order (a) sälj 0.5RL@3TICK
  * Org 2 lägger order (b) köp 0.5RL@3TICK
  * Förväntat svar: inget avslut, båda ordrar ligger kvar
  * Org 1 lägger order (c) köp 1RL@3TICK
  * Org 1 lägger order (d) sälj 1RL@3TICK
  * Förväntat svar:
    - Avslut mellan order (c) och (d) 1RL@3TICK samt ordrarna tas bort.
    - Avslut mellan order (a) och (b) 0.5RL@3TICK samt ordrarna tas bort.

4. Internavslut 1
  * Org 1 lägger order (a) sälj 2RL@1TICK
  * Org 2 lägger order (b) sälj 2RL@1TICK
  * Org 2 lägger order (c) köp 2RL@1TICK
  * Förväntat svar:
    - Avslut mellan order (b) och (c) 2RL@1TICK, samt ordrarna finns ej kvar
    - Order (a) är orörd.

5. Avslut med tidsprio 1
  * Org 1 lägger order (a) sälj 2RL@1TICK
  * Org 2 lägger order (b) sälj 2RL@1TICK
  * Org 3 lägger order (c) köp 2RL@1TICK
  * Förväntat svar:
    - Avslut mellan order (a) och (c) 2RL@1TICK, samt ordrarna finns ej kvar
    - Order (b) är orörd.

6. Avslut mot isbergsorder 1
   * Org 1 lägger isbergsorder (a) sälj 10RL varav 2RL öppen @7TICK. DisplayMethod=WhenFilled.
   * Org 2 lägger order (b) köp 5RL@7TICK.
   * Förväntat resultat:
     - Avslut mellan order (a) och (b) 5RL@7TICK
     - Order (b) finns ej kvar
     - Order (a) ligger kvar med 5RL varav 1RL öppen
   * Org 1 lägger order (c) köp 1RL@8TICK.
   * Förväntat resultat:
     - Avslut mellan order (a) och (c) 1RL@7TICK
     - Order (c) finns ej kvar
     - Order (a) ligger kvar med 4RL varav 2RL öppen
     - I FIX-fallet kan man även verifiera att en ny order (nytt ID) skapades för den nya öppna volymen i det publika marknadsdatat.

Admingrejer:

Hur är marknader, instrument, sessioner osv modellerade? I vissa protokoll (ex FIX) så tillhör ett instrument en viss marknad men följer en viss sessions öppettider. Sedan ändras endast status mm för sessionen. I UniTP så ändras t ex status för varje instrument och sessioner finns inte (iaf inte med den här betydelsen).

1. Öppna marknaden 1.
 * Se till att alla marknader (och instrument) är CLOSED.
 * Sätt T till nuvarande tid.
 * För alla marknader:
  - Schemalägg nästa PRE_TRADE till T+30s.
  - Schemalägg nästa CALL_AUCTION till T+60s.
  - Schemalägg nästa CONTINOUS_TRADING till T+90s.
  - Schemalägg nästa POST_TRADE till T+120s.
  - Schemalägg nästa CLOSING till T+150s.
 * Förväntat resultat:
  - Alla marknader och deras instrument har status enligt ovan.
  - Tiderna stämmer ungefärligt (+- 30 sekunder kanske?)

2. Trade halt av ett instrument 1.
 * Se till att alla marknader (och instrument) är CLOSED.
 * Sätt T till nuvarande tid.
 * För alla marknader:
  - Schemalägg nästa PRE_TRADE till T+30s.
  - Schemalägg nästa CALL_AUCTION till T+60s.
  - Schemalägg nästa CONTINOUS_TRADING till T+90s.
  - Schemalägg nästa POST_TRADE till T+120s.
  - Schemalägg nästa CLOSING till T+150s.
 * Invänta status PRE_TRADE, schemalägg nästa till T+180:
 * Invänta status CALL_AUCTION, schemalägg nästa till T+210:
 * Invänta status CONTINOUS_TRADING, schemalägg nästa till T+240:
 * Sätt instrument (a) i tradehalt, verifiera att så sker.
 * Se till att instrument (a) öppnar (sätts i PRE_TRADE) osv samtidigt som övriga instrument i samma marknad.
 * Invänta status POST_TRADE, schemalägg nästa till T+270:
 * Invänta status CALL_AUCTION, schemalägg nästa till T+300:
 * Verifiera att instrument (a) öppnar och stänger samtidigt som övriga instrument i samma marknad, dvs status passerar PRE_TRADE, CALL_AUCTION, CONTINOUS_TRADING, POST_TRADE och CLOSING.

Prestanda:
---

1. Lägg in och ta bort ordrar.
 * Sätt maximalt antalt utestående (ej ackade) operationer till 50
 * Repetera följande 100 ggr:
  - För varje instrument, org 1 lägger in 1 köp-order (med omsändning om så behövs) med volym 1RL och pris slumpmässigt i intervallet [1TICK, 10TICK] och 1 sälj-order (omsändning) med volym 1RL och pris slumpmässigt i intervallet [11TICK,20TICK].
 * Räkna ut hur många order-insert per sekund som man kan göra i snitt.
 * Ta bort alla ordrar (en i taget).
 * Räkna ut hur många order-delete per sekund som man kan göra i snitt.

2. Modifiera ordrar 100 ggr per sekund.
 * Sätt maximalt antalt utestående (ej ackade) operationer till 50
 * För varje instrument, org 1 lägger in 1 köp-order (med omsändning om så behövs) med volym 1RL och pris slumpmässigt i intervallet [1TICK, 10TICK] och 1 sälj-order (omsändning) med volym 1RL och pris slumpmässigt i intervallet [11TICK,20TICK].
 * Repetera följande 100 ggr:

- För varje skapad order (x), repetera följande i en takt på 100 ggr/s:
  # Modifiera pris för order så att det för köp-order hamnar intervallet [1TICK, 10TICK] och för sälj-order hamnar intervallet [11TICK,20TICK].

- Räkna ut hur många order-modify per sekund som har gjorts i snitt (borde vara 100 ggr/s eller långsammare).

Version

1

NORDIC GROWTH MARKET NGM AB

UTVECKLING AV BÖRSSYSTEM

Användarhandbok för Generic Test Client (GTC)

UTVECKLING AV BÖRSSYSTEM

Användarhandbok för Generic Test Client

Sveavägen 21
Stockholm
Telefon 08 – 566 390 00 • Fax 08 – 556 39 001

Innehåll

Kapitel 1

Introduktion till Generic Test Client (GTC)

Den skriptbara klienten är enkel att anpassa till specifika önskemål. Denna handbok skall enkelt kunna ge en överblick i hur den kan användas på ett enkelt sätt.

Syftet med klienten är att på ett enkelt sätt skall kunna kommunicera med börsen och analysera dess handlingar. Den implementerar ett enkelt API mot olika protokoll och med minimala förändringar skall samma skript kunna användas mot olika protokoll och servrar. Klienten är trots sin generella karaktär fullt förmögen att utföra förändringar ner på lägsta protokollnivå och kan manipulera specifika fält i kommunikationen utan att användaren behöver sätta upp komplicerade miljöer. Istället kan användaren använda sig av de enkla metoder GTC erbjuder. Vidare erbjuder klienten möjligheter att skapa grafiska gränssnitt så användare utan programmeringserfarenheter enkelt kan använda skript och hjälpa till med felsökning om så skulle behövas. I version 1 av GTC finns enbart stöd för Univits Trading Protocol.

Denna handbok ger ingen information om hur marknaden fungerar med instrument, orderböcker eller ordrar eller detaljerad information om hur UniTP eller något annat protokoll fungerar. För information om detta hänvisas till de dokument som finns att tillgå på NGM's intranet.

Handbokens upplägg.
Fokus ligger på att ge användaren en snabb insyn i hur GTC används, och det finns en rad exempel på olika uppgifter som kan utföras. Alla exempel är kompletta, dvs de kan kopieras och köras rakt av (med nödvändiga ändringar användarnamn och lösenord förståss). Handboken börjar med att ge en överblick över GTC, hur den är uppbyggd och hur den kan användas och vad dess begränsningar är, för att sedan gå vidare och presentera de mest centrala delarna i GTC. Detta är dock inte ett fullständigt API, utan då hänvisas till referensbiblioteket i javadoc format (medföljer i distributionen).

Systemkrav. GTC är plattformsoberoende och har körtestats under Windows, Macintosh och Debian Linux. GTC är skriven i Java och stödjer JavaScript version 1.5 samt UniTP version 1.11, den är kompilerad under java 6 och kräver att jre 1.6 är installerat samt att Rhino 1.6 biblioteket finns tillgängligt (bifogas i distributionen). API är också bifogat och finns med i standardutförandet av javadoc för Java 6. UniTP paketet är en egen del av GTC och lämpar

sig mycket väl till att användas i andra projekt, den kräver även en kontinuerlig uppdatering i takt med att UniTP protokollet utvecklas.

Vad kan botten utföra?

GTC är byggd för att undersöka hur marknaden svarar på olika instruktioner, till exempel kan man undersöka om handelsregler efterföljs eller se om utlovad funktionalitet beter sig som förväntat. Det är även passande till att göra belastningsundersökningar och se hur marknaden reagerar vid överbelastning och var flaskhalsar sitter. Med enklare modifikation och utveckling skulle den även kunna användas till kontinuerlig marknadsövervakning för att till exempel identifiera insiderhandel. Den passar även bra att jämföra protokoll och beräkna trafikmängder eller vilka protokoll som ger bäst prestanda i termer av overhead eller skillnaden mellan att skicka paket synkront eller asynkront. Tack vare enkelheten att direkt modifiera paket ner till fältnivå kan GTC användas till att undersöka serverns förmåga att hantera trasiga eller felaktiga paket och hur den reagerar på dessa, eller kontrollera ny funktionalitet.

Ej en garanterat fullständig representation

GTC garanterar inte på något sätt att den representerar marknaden korrekt, den är fullt kapabel till att göra det, men saknar helt funktioner för att själv synkronisera och utföra självkontroller på sin marknadsdata. Om du till exempel läser in marknadsdata från en fil, åligger det användaren att kontrollera att marknaden faktiskt ser ut som inläst data gör. Det går tämligen enkelt att göra detta, men GTC gör det inte automatiskt själv. Detta för att användaren skall ha full kontroll över den kommunikation som sker, och det inte händer en massa saker under ytan som användaren inte känner till. Det finns dock ett undantag till denna regel; GTC skickar hjärtslag till servern för att uppkopplingen inte skall gå förlorad. Det är därför inte rekommendabelt att använda GTC för som underlag till faktiska inköp eller försäljningar på en ”live” server.

Hur fungerar GTC?

Det finns två huvudsakliga komponenter i GTC, front-end delen vilket erbjuder användaren ett API för att kommunicera med börsen samt en back-end del som kommunicerar protokollspecifikt med börsen. För att implementera fler protokoll till GTC läggs de till och metoderna i front-enddelen anpassas för att kommunicera med back-end. Front-end ligger i paketet ”bot” och back-end ligger i de olika paketen som specificerar protokoll.

I botpaketet finns följande klasser:

MarketConnect. Sköter kommunikationen mellan GTC och börsen, och har begränsade möjligheter att skriva ut paket, men är huvudsakligen tänkt att identifiera inkomna paket och skicka dem vidare till en klient (exempelvis MarketView beskrivet nedan) för vidare behandling och/eller lagring.

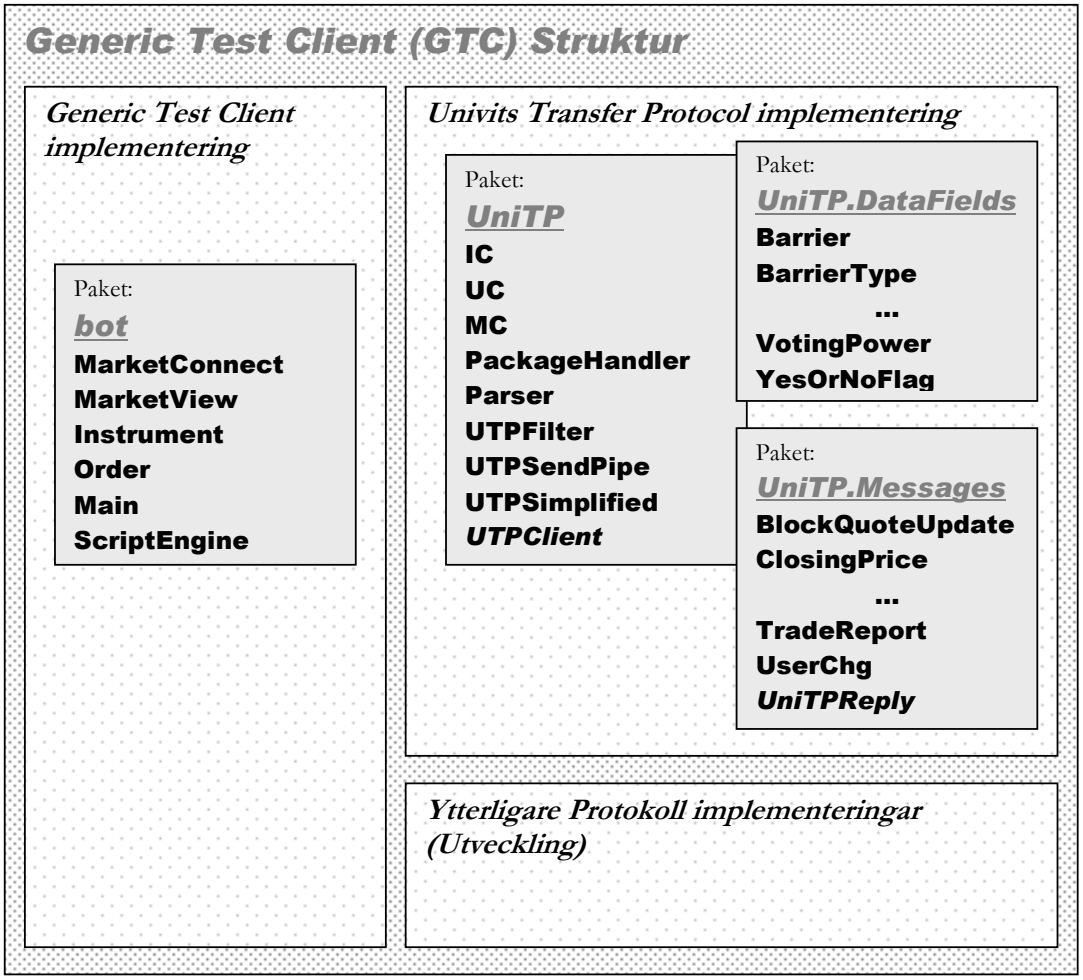
MarketView. Samlar upp data och gör den enkelt överskådlig och tillgänglig i en protokolloberoende vy. Innehåller en intern databas bestående av instrument och ordrar och implementerar funktioner för att skriva eller läsa data från en fil. Har även lite mer avancerade funktioner för att identifiera ändringar i marknadsdatan och belysa och presentera dessa.

Instrument. Är en protokolloberoende representation av ett instrument på marknaden

Order. Är en protokolloberoende representation av ordrar på marknaden

Main. Tar emot information från kommandoraden och startar upp de olika delarna.

Scriptengine. Startar trådar för skript och analyserar och exekverar dem.



Kapitel

2

Market Connect & Market View

Dessa två objekt är den centrala delen för kommunikation med marknaden. Förståelse och funktionaliteten av dessa är en stomme för skriptskrivande till GTC.

Marketconnect är användarens anslutning till marknaden, därigenom sker all kommunikation till marknaden, i market view samlas inkommen data upp och ger en bild över hur marknaden ser ut för tillfället. Det finns även funktionalitet för att se vad skillnaden mellan två marknader är, för att identifiera vilka ändringar som har skett mellan två snapshots av marknaden.

Den egentliga trafiken kontrolleras av MarketConnect, och MarketView läggs till som en observatör av denna och uppdateras så fort något kommer in och skickas vidare av MarcetConnect. En MarketConnect kan ha flera observatörer och en MarketView kan observera flera MarketConnects. Det går även att skapa mer avancerade former av hur data propageras genom tex trädstrukturer, detta diskuteras i närmare detalj i kapitel 6.

Dessa två objekt ger användaren ett enkelt interface till marknaden de tillhandahåller de funktioner användaren kan behöva använda för att utföra merparten av användaroperationerna på börsen utan att behöva bry sig om att deras uppkoppling avbryts pga att inget sänds, eller vilken kanal paketet eller svaren kommer. MarketView implementerar dessutom interfacet JTable för att på ett enkelt sett kunna ritas upp grafiskt av användaren.

Både MarketView och MarketConnect är protokolloberoende i merparten av deras funktioner. Tanken är att samma skript skall kunna exekveras med minimala ändringar på olika protokoll. Tyvärr gör olikheterna i protokollen det omöjligt att göra alla funktioner totalt protokolloberoende, exempelvis behövs det flera användarnamn och lösenord för att logga in på vissa protokoll, emedan andra endast kräver ett lösen-användarnamn par. För att inte begränsa användaren om han vill kunna skicka protokollspecifika paket finns det möjlighet att skicka in råa paket direkt. Även dessa funktioner är givetvis inte protokolloberoende. Metoder som inte är protokolloberoende är listade för sig.

MarketConnect

MarketConnect är den del som kontrollerar kommunikationen mellan börsen och GTC. Ett närmare API finns bifogat i distributionen och är kontinuerligt uppdaterad. Det är därför rekommenderat att det API används primärt. Här är en översikt över metoderna i MarketConnect. Observera att även om du kan skapa flera MarketConnect objekt är det inte garanterat att servern accepterar det, till exempel har UniTP (1.11) endast stöd för totalt en UC kanal.

MarketConnect

Konstrukor **MarketConnect**(String server, String multicast, int port, int protocol)

Skapar en ny instans av MarketConnect, kopplad till angiven server och angiven multicast grupp. Porten är för UniTP servrar IC kanalen och UC kanalen antas vara port + 1 och MC port + 2. Protokoll variabeln anger vilket språk servern pratar
1: UniTP

void **addObserver**(UTPclient client)
Denna metod används för att lägga till en observatör av trafiken, till exempel en MarketView. Även om denna metod är specificerad till UTP protokollet skall denna metod overloadas och användas för andra protokoll när dessa implementeras (se kapitel 7). Dvs en motsvarande addObserver(FIXclient client) borde finnas för FIX protokollet.

void **removeObserver**(UTPclient client)
Liksom metoden ovan skall denna metod overloadas och användas för andra protokoll. Med denna metod tas en observatör bort från MarketConnect

void **printLine**(String s)
Denna funktion skriver ut angiven text till samma utkanal som MarketConnect.

void **setClientFilter**(String filterfile)
De paket som skickas ut till observatörerna passeras först genom detta filter, man kan till exempel filtrera bort heartbeats eller en hel kanal. För att filtrera UniTP paket använder man sig av UniTPFilter och strängen skall ange sökväg och filnamn till det xml dokument som specificerar filtret. Filter är närmare beskrivna i kapitel 6.

void **setVerbose**(int verboselevel)
Använd detta för att enkelt sätta på och stänga av utskrivning av inkomna paket. 0 betyder ingen utskrift, 1 betyder utskrift. Standard är 0.

void **setVerboseFilter**(String filterfile)
Använd detta för att filtrera paketen för utskrift, om inget filter anges passerar alla inkommna paket filtret. Ange sökvägen och filnamn till den fil till det xml dokument som specificerar filtret. Användning av filter presenteras närmare i kapitel 6

void **waitOnCatchup**()
Denna metod blockerar tills dess att klienten och servern inte har några väntande paket. Den exakta implementationen kan variera mellan olika protokoll. I UniTP blockerar den tills ett svar på senaste skickade IC paket inkommit på IC kanalen.

void **getAllMarketData**()
Hämtar ner alla tillgänglig data från servern förutom orderhistorik (då detta kan vara extremt tidskrävande) vilket i UniTP inkluderar, instrument, orderböcker och alla ordrar för alla submarkets

void **getAllMarketData**(String submarket)
Med denna metod begränsar du nerhämtad data till angiven submarket, tex ”NDX”.

void **requestDataForInstrument**(Instrument i)
Begär information om all tillgänglig data för instrumentet förutom orderhistorik. Vilket i UniTP inkluderar; instrument, orderböcker och ordrar.

void **requestHistoryForInstrument**(Instrument i, int daysBack)
Med denna metod kan du begära alla avslut som gjorts i orderboken instrumentet är associerat med för angivet antal dagar tillbaka.

void **login**(String participant, String session, String password, String UCsession, String UCpassword, String username)
Logga in på en UniTP server, observera att denna metod inte kan användas till att logga in på börser som använder andra protokoll.

void **logoff**()
Loggar av och stänger ner alla eventuella trådar som körs för att hålla sessionen vid liv eller analyserar inkomna paket.

void **buy**(Instrument stock, Double price, int volume)
Lägger in en köporder för angivet Instrument till angivet pris och volym. MarketView antar att det är ett ett-till-ett förhållande mellan orderbok och instrument. Ordern läggs in med omedelbar verkan.

void **queBuy**(Instrument stock, Double price, int volume)
Precis som metoden ovan lags en order in, men först när ett ack på att alla föregående paket (skickat genom que eller direkt) har inkommit. Observera att om du blandar att skicka paket genom kön (vänta på att tidigare paket skickats) och med direkt metod, finns det ingen garanti i vilken ordning de kommer. Kön väntar till första luckan då senast ack stämmer överens med senast skickade paket, vilket kan inträffa innan metoden som direkt sänder alla sina paket har avslutats.

void **sell**(Instrument stock, Double price, int volume)
Lägger in en säljorder för angivet Instrument till angivet pris och volym. MarketView antar att det är ett ett-till-ett förhållande mellan orderbok och instrument. Ordern läggs in med omedelbar verkan.

void **queSell**(Instrument stock, java.lang.Double price, int volume)
se queBuy.

void **removeOrder**(Instrument i, Order o)
Ger en instruktion att ta bort order o tillhörande instrument i till börsen

void **modifyOrderVolume**(Instrument i, Order o, long volumechange, long openvolume)
Gör en relativ ordervolyms ändring *volumechange* I ordern o tillhörandes instrumentet i. Även modifieringen av den öppna volymen är relativ till det tidigare värdet.

void **queModifyOrderVolume**(Instrument i, Order o, long relativetotalvolumechange, long openvolume)
se queBuy.

void **sendUTPPacket**(UTPPacket packet)
Skickiar det angivna paketet direkt på IC kanalen som MarketConnect har.

void **queSendUTP**(UTPPacket p)
se queBuy.

MarketView

MarketView är en generell pakethanterare och kan observera en MarketConnect (och även direkt IC, UC och MC kanaler om du skapar dem manuellt). Den tillhandahåller ett protokolloberoende interface till marknaden och erbjuder funktioner att lagra eller läsa marknaden till fil. Den kan observera flera objekt (till exempel en IC, en UC och en MC kanal) eller inget och istället läsa in data från en fil och arbeta med data så. Nedan följer en översikt över vilka metoder MarketView erbjuder, men för ett komplett API hänvisas till det med distributionen tillhörande javadoc specifikationen. MarketView implementerar även JTable och kan därför enkelt ritas upp grafiskt om så önskas.

MarketView

Konstruktor **MarketView**()
Skapar en ny instans av MarketView.

void **clear**()
Tömmer marknadsvyn från all data.

void **freeze**(boolean freeze)
Fryser marknadsvyn som den ser ut nu, och ignorerar alla paket som skickas till den medan den är fryst.

void **setVerbose**(int verboselevel)
Vilken nivå av utskrift som marknadsvyn skall ge
0: Tyst.
1: Tämligen tyst. Orderuppdateringar presenteras på ett slimmat sätt. (se modifiering av marknadsdata i kapitel 4 för närmare specifikation).
2: Pratig. Alla inkomna paket med all information skrivs ut när de kommer in (för att bara skriva ut vissa paket se kapitel 6 eller använd dig av setVerboseFilter i MarketConnect och låt dem skrivas ut direkt från det objektet).
3: Spara alla meddelanden enligt verboselevel 1 och skriv ut dem med

metoden printReplies (se nedan). Ett exempel som använder denna metod kan ses i kapitel 4.

void **printReplies**(int verboselevel)
Skriv ut alla sparade meddelanden och sätt sedan verboselevel till given nivå, se ovan för specificering av de olika nivåerna.

void **readFromFile**(String filename)
Läs in marknadsdata från en fil, angivet filnamn skall vara utan ändelse (.mv).

void **saveToFile**(String filename)
Skriv all marknadsdata till angiven fil. Filnamnet skall vara utan ändelse (.mv).

Order **getOrder**(String orderId)
Hämta Order med angivet order-ID (i UniTP; timestampOrig) om den finns i marknadsdatan, annars returneras null.

Instrument **getStock**(String name)
Returnerar första Instrumentet med angivet namn, om ingen sådan finns returneras null.

ArrayList<Order> **getOrders**()
Returnerar all order som finns I marknadsvyn.

ArrayList<Instrument> **getStocks**()
Returnerar alla instrument som finns i marknadsvyn, med tillhörande ordrar.

void **setSaveOrderValue**(String valuename)
Sparar specifik variabel från inkomna paket och kan på så sätt användas till att studera protokollspecifika variabler utan att behöva kommunicera på en lägre nivå. Förklaras i närmare detalj i kapitel 5.

ArrayList<Order> **subtract**(MarketView mv)
Subtraherar alla order som finns i given marknadsvy från sparad data, om ett Instrumentet kan matchas identiskt med den givna marknadsvyn tas hela instrumentet bort.

Instrument

Instrument är en protokolloberoende klass som representerar marknads olika instrument (tex aktier, warranter eller optioner). Ett instrument kan a noll eller flera ordrar.

Konstruktor **Instrument**(String name, String id, long orderbookId, long roundlotsize, TreeMap< Double, Double> ticksize, ArrayList<Order> orders)
Skapar en ny instans av Instrument. Ticksize är en sorterad mängd av sammankopplade nyckel-värdepar där nyckeln är priset och värdet är tillhörande tickstorlek.

String **getId**()
Returnerar ID för instrumentet.

String **getName**()
Returnerar namnet för instrumentet.

long **getOrderbookId**()
Returnerar orderboks ID som är sammankopplat med instrumentet. GTC antar att det är ett ett-till-ett förhållande mellan instrument och orderbok.

ArrayList<Order> **getOrders**()
Returnerar ordrarna som är associerade med detta instrument.

long **getRoundLotSize**()
Returnerar storleken för en börspost för detta instrument.

Double **getTickSizeForPrice**(Double price)
För angivet pris returneras vad tick storleken är

Order

En Order är en protokolloberoende representation av en order (tex köp 1000 aktier för 100 kr styck) De kan även skapas med en extra variabel om så önskas för att kunna studera vissa specifika variabler, man kanske är intresserad av vem som satte in ordern, då kan man lägga till detta som en String i fältet special. Se kapitel 5 för närmare instruktioner hur detta fungerar.

Konstruktor **Order**(String buySell, Double price, long volume, String id, Object special)
Skapar en ny instans av Order. Special är en extra variabel, den används till exemepl när marketview skapar order objekt och den observerar speciella värden. Se kapitel 5 för närmare beskrivning.

String **getBuySell**()
Returnerar "B" för köpordrar och "S" för säljordrar

String **getId**()
Returnerar orderns ID, för UniTP är det TimeStampOrig

double **getPrice**()
Returnerar priset som ordern begär.

Object **getSpecial**()
Returnerar den extra sparade variabeln. Se Kapitel 5 för närmare information

long **getVolume**()
Returnerar kvarvarande storlek på ordern.

Kapitel 3

Skriptmotorn

Skriptmotorn är en Rhino skriptmotor och är ett kraftfullt verktyg för att exekvera javascript.

Rhino är en kraftfull javascript motor och körs under jvm, den går enkelt at byta ut och uppdatera (byt bara ut biblioteket). För en komplett Rhino guide rekommeneras att besöka Rhinos hemsida: www.mozilla.org/rhino. Nedan följer endast några små tricks som kan göras för att GTC skall bli användarvänligare. Rhinos scopelevel är satt till top-level (global) så det finns tillgänglighet att importera alla klasser och paket.

ImportClass(klass) ”Importerar” en klass genom att göra namnet tillgängligt som en egenskap i Rhinos scope
importPackage(paket) ”Importerar” alla klasser I ett paket.

Det går även bra att skapa grafiska komponenter med hjälp av Rhino, eller skapa flera contexter inne i ett skript och på så starta körningen av en skriptfil från en annan skriptfil både parallellt och seriellt. För exempel av detta se exempel och dokumentation från ovanstående länk.

Det är alltså fullt möjligt att skapa ett grafiskt gränssnitt i javascript för att köra andra skript och på så sätt göra ett egen och bättre anpassat grafiskt interface än det i förväg konstruerade grafiska interfacet.

Kapitel 4

Exempel och körning

I följande kapitel kommer en rad exempel på skript och kommentering till dem för att ge en överblick på hur skript kan se ut.

Exemplen nedan har exekverats i Debian Linux mot UniTP 1.11. Exemplen kan köras direkt som de står här, med modifiering av användarnamn etc. Den mest grundläggande körningen består av att köra skript vilka startas med kommandot *java –jar bot-jar –e <filnamn>*. Om du till exempel vill köra skriptet getandsave.js skriver du: *java –jar bot.jar –e getandsave.js*. I kapitel 6 finns lite mer komplicerade skipt förklarade.

Körning

> java -jar bot.jar

```
java -jar bot.jar [-i][-g][-e file]
 -i     Interactive mode.
 -g     Start bot with the default graphical user interface.
 -e     Execute <file>, a javascript file.
```

Interaktivt: I interaktiv körning startas ett Rhino shell, från detta kan du sedan evaluera metoder eller köra skript.

Grafiskt Användargränsnitt: Ett grafiskt interface där senast körda skript finns enkla att köra med en knapptryckning

Exekvera fil: Det normala sättet att köra ett skript

För att undvika att varje skript börjar med att ladda ner data från marknaden (det blir onödigt långa skript, kräver mycket trafik och tar tid) rekommenderas det att man gör ett skript som laddar ner all data och sedan skriver det till en fil som sedan andra skript använder sig av. Givetvis kan problemet vara att det inladdade datan inte överrensstämmer med hur marknaden ser ut (den kan ju ha ändrats efter det att datan sparades, men innan filen lästes in), så i situationer där detta är ett måste rekommenderas att all marknadsdata blir inläst vid varje uppstart. Det första exemplet blir således ett skript för att ladda ner all marknadsdata och spara den till en fil. I flera exempel som följer (och även i kapitel 6) krävs det att detta skript är kört och en fil med aktuell marknadsdata finns tillgänglig för inläsning.

Ladda ner och spara marknadsdata

```
importPackage(Packages.bot);

marketview = new MarketView();
market = new MarketConnect("sv-dev-02.x.ngm.se", "239.255.0.10",  11910, 1);
market.setVerbose(0);
market.addObserver(marketview);

market.login("ORG1","ABC1","ABC1","ABC2","ABC2","ORG1-M1");
market.getAllMarketData("NDX");
market.waitOnCatchup();
market.printLine("Please stand by while closing down connections.");
market.logoff();
marketview.saveToFile("saveddata");
market.printLine("Got and saved all market data for market: NDX");
```

Det första vi gör är att importera de paket vi använder oss av, följt av att vi skapar en marknadsvy samt en uppkoppling mot marknaden. Ingen utskrift sker av inkomna paket med kommandot *setVerbose(0)* och vi lägger sedan till marknadsvyn som observatör av kopplingen. Inloggning sker och vi begär sedan all marknadsdata för submarknaden NDX och väntar på att alla förfrågningar det innebär med *waitOnCatchup()* vilket blockerar tills svar inkommit på alla förfrågningar. Vi loggar sedan av och sparar all data vi fått in till filen saveddata.mv

```
> savedata.mv
```

Notera att saveddata.mv är enbart en representation av data som hämtades ner och hur den såg ut då. Om inte en fullständig representation laddades ner eller om ändringar har gjorts efter att den sparades till fil, stämmer inte sparad marknadsdata och marknaden överens.

I följande exempel läser vi in data från den sparade filen och använder det som bas för modifieringar på marknaden. Vi går igenom köp, sälj, modifiering och borttagning av ordrar. Exemplet använder sig dessutom av lite mer avancerade verbose funktioner för att göra en trevlig presentation av datan.

Modifiering av marknadsdata

```
importPackage(java.awt);
importPackage(java.lang);
importPackage(Packages.UniTP);
importPackage(Packages.UniTP.Messages);
importPackage(Packages.UniTP.DataFields);
importPackage(Packages.bot);

marketview = new MarketView();
market = new MarketConnect("sv-dev-02.x.ngm.se", "239.255.0.10",  11910, 1);
```

```
market.addObserver(marketview);
marketview.setVerbose(3);
market.login("ORG1","ABC1","ABC1","ABC2","ABC2","ORG1-M1");
marketview.readFromFile("saveddata");
stocklist = marketview.getStocks();

index = 1;
for (i = 5; i > 0 ; i--) {
        price = Math.round((Math.random()*10) + 91);
        market.printLine("[" + index + "]\t" + stocklist.get(i-1).getName()
                        + " - B 1000 @" + price
                        + " -> *ATTEMPTING INSERT*");
        market.buy(stocklist.get(i-1), price, 1000);
        index++;
        price = Math.round((Math.random()*10) + 71);
        market.printLine("[" + index + "]\t" + stocklist.get(i-1).getName()
                         + " - S 1000 @" + price
                         + " -> *ATTEMPTING INSERT*");
        market.sell(stocklist.get(i-1), price, 1000);
        index++;
}

for (i = 10; i > 5 ; i--) {
        if (stocklist.get(i-1).getOrders().size() > 0) {
                        market.printLine("[" + index + "]\t"
                          + stocklist.get(i-1).getOrders().get(0).getId() + " - "
                          + stocklist.get(i-1).getOrders().get(0).getBuySell() + " "
                          + stocklist.get(i-1).getOrders().get(0).getVolume()
                          + " @" + stocklist.get(i-1).getOrders().get(0).getPrice()
                          + " -> * INC. VOL. 1000*");
                        market.modifyOrderVolume(stocklist.get(i-1),
                          stocklist.get(i-1).getOrders().get(0), 1000, 0);
                        index++;
        } else {
                        market.printLine("There are no Buy orders that you can
                          modify in " + stocklist.get(i-1).getName());
        }
}

for (i = 15; i > 10 ; i--) {
        if (stocklist.get(i-1).getOrders().size() > 0) {
                        market.printLine("[" + index + "]\t"
                          + stocklist.get(i-1).getOrders().get(0).getId() + " - "
                          + stocklist.get(i-1).getOrders().get(0).getBuySell() + " "
                          + stocklist.get(i-1).getOrders().get(0).getVolume()
                          + " @" + stocklist.get(i-1).getOrders().get(0).getPrice()
                          +" -> *ATTEMPTING DELETE*");
                        market.removeOrder(stocklist.get(i-1),
                          stocklist.get(i-1).getOrders().get(0));
                        index++;
        } else {
                        market.printLine("There are no Buy orders that you can
```

```
                        delete in " + stocklist.get(i-1).getName());
        }
}

market.waitOnCatchup();
Thread.sleep(500);
marketview.printReplies(1);
market.logoff();
```

I stället för att begära all marknadsdata från servern så använder vi oss av *readFromFile(saveddata),* vilket läser in saveddata.mv. Vi skapar sedan en list på all instrument i marknadsdatan och för de 5 första tar vi och lägger in en köporder och en säljorder. Notera att priserna överlappar varandra och de borde därför resultera i ett avslut. Sedan modifierar vi volymen på nästföljande 5 ordrar och därefter tar vi bort de 5 ordrarna efter det. *SetVerbose(3)* satte verboselevel till 3, vilket innebär att alla svar sparas istället för stt skrivas ut, om vi skrev ut dem direkt skulle vi få svar innan alla nya ordrar, modifieringar och borttagningar skickas, vilket skulle göra att det blev en blandning av utgående och inkommande trafik. Vi skriver därför först ut all utgående trafik först och sedan all inkommande trafik när den är avslutad. Med kommandot *printReplies(1)* skrivs alla sparade svar ut, och efterföljande svar skrivs ut direkt.

Se upp! (UniTP relaterat) MarketConnect.logOff() stänger ner förbindelsen till marknaden för samtliga kanaler, och MarketConnect.waitOnCatchup() blockerar endast till IC kanalen är i synk. Om du loggar av direkt efter waitOnCatchup är det möjligt att det finns paket på UC eller MC kanalen som inte tagits emot eller hunnit skickats från servern än. Om matchningsmotorn är belastad kan det till exempel ta en liten stund innan en transaktion går igenom. Det finns tyvärr inte något sätt att kontrollera om alla resulterande handlingar av en orderändring har gjorts och skickats eller något sätt att ta reda på matchiningmotorns belastning och förväntad responstid. Användaren uppmanas därför att vänta (till exempel med Thread.sleep()) en tid som troligen är tillräcklig innan *MarketConnect.logoff()* tillkallas.

Resultat av modifiering av marknadsdata

```
[1]     A102 - B 1000 @100 -> *ATTEMPTING INSERT*
[2]     A102 - S 1000 @74 -> *ATTEMPTING INSERT*
[3]     A101 - B 1000 @99 -> *ATTEMPTING INSERT*
[4]     A101 - S 1000 @78 -> *ATTEMPTING INSERT*
[5]     A100 - B 1000 @96 -> *ATTEMPTING INSERT*
[6]     A100 - S 1000 @71 -> *ATTEMPTING INSERT*
[7]     A10 - B 1000 @97 -> *ATTEMPTING INSERT*
[8]     A10 - S 1000 @72 -> *ATTEMPTING INSERT*
[9]     A1 - B 1000 @94 -> *ATTEMPTING INSERT*
[10]    A1 - S 1000 @77 -> *ATTEMPTING INSERT*
[11]    20070911133316849428 - B 1000 @85.0 -> *INC. VOL. 1000*
[12]    20070911133316851430 - B 1000 @84.0 -> *INC. VOL. 1000*
[13]    20070911133316854432 - B 1000 @84.0 -> *INC. VOL. 1000*
[14]    20070911133316856434 - B 1000 @91.0 -> *INC. VOL. 1000*
[15]    20070911133316859436 - B 1000 @87.0 -> *INC. VOL. 1000*
[16]    20070911133316837418 - B 1000 @84.0 -> *ATTEMPTING DELETE*
[17]    20070911133316840420 - B 1000 @83.0 -> *ATTEMPTING DELETE*
[18]    20070911133316842422 - B 1000 @89.0 -> *ATTEMPTING DELETE*
```

```
[19]    20070911133316845424 - B 1000 @87.0 -> *ATTEMPTING DELETE*
[20]    20070911133316847426 - B 1000 @83.0 -> *ATTEMPTING DELETE*

[Order Insert:  B A102  (20070911134752217426) 1000 @ 100.0]
[Order Insert:  S A102  (20070911134752225427) 1000 @ 74.0]
[Order Deleted: S A102  (20070911134752225427) Reason: 1]
[Order Deleted: B A102  (20070911134752217426) Reason: 1]
[Trade in:      S A102  (20070911134752225427) 1000 @ 100.0]
[Trade in:      B A102  (20070911134752217426) 1000 @ 100.0]
[Order Insert:  B A101  (20070911134752228428) 1000 @ 99.0]
[Order Insert:  S A101  (20070911134752228429) 1000 @ 78.0]
[Order Deleted: B A101  (20070911134752228428) Reason: 1]
[Order Deleted: S A101  (20070911134752228429) Reason: 1]
[Trade in:      B A101  (20070911134752228428) 1000 @ 98.0]
[Trade in:      S A101  (20070911134752228429) 1000 @ 98.0]
[Order Insert:  B A100  (20070911134752229430) 1000 @ 96.0]
[Order Insert:  S A100  (20070911134752229431) 1000 @ 71.0]
[Order Deleted: B A100  (20070911134752229430) Reason: 1]
[Order Deleted: S A100  (20070911134752229431) Reason: 1]
[Trade in:      S A100  (20070911134752229431) 1000 @ 96.0]
[Trade in:      B A100  (20070911134752229430) 1000 @ 96.0]
[Order Insert:  B A10   (20070911134752231433) 1000 @ 97.0]
[Order Insert:  S A10   (20070911134752236434) 1000 @ 72.0]
[Order Deleted: S A10   (20070911134752236434) Reason: 1]
[Order Deleted: B A10   (20070911134752231433) Reason: 1]
[Trade in:      S A10   (20070911134752236434) 1000 @ 97.0]
[Trade in:      B A10   (20070911134752231433) 1000 @ 97.0]
[Order Insert:  B A1    (20070911134752265436) 1000 @ 94.0]
[Order Insert:  S A1    (20070911134752274439) 1000 @ 77.0]
[Order Deleted: S A1    (20070911134752274439) Reason: 1]
[Order Deleted: B A1    (20070911134752265436) Reason: 1]
[Trade in:      S A1    (20070911134752274439) 1000 @ 94.0]
[Trade in:      B A1    (20070911134752265436) 1000 @ 94.0]
[Order Update:  B A107  (20070911133316849428) 2000 @ 85.0]
[Order Update:  B A106  (20070911133316851430) 2000 @ 84.0]
[Order Update:  B A105  (20070911133316854432) 2000 @ 84.0]
[Order Update:  B A104  (20070911133316856434) 2000 @ 91.0]
[Order Update:  B A103  (20070911133316859436) 2000 @ 87.0]
[Order Deleted: B A111  (20070911133316837418) Reason: 2]
[Order Deleted: B A110  (20070911133316840420) Reason: 2]
[Order Deleted: B A11   (20070911133316842422) Reason: 2]
[Order Deleted: B A109  (20070911133316845424) Reason: 2]
[Order Deleted: B A108  (20070911133316847426) Reason: 2]
```

Notera hur all utgående data är skriven först och svaren kommer sen och det inte är en blandning av dem. Notera också hur orderhanteringen går till med de överlappande ordrarna och hur avslut fungerar.

Kapitel 5

Direktåtkomst av protokoll

Ibland kan Marketview och Market connect vara knapphändiga om man vill göra specifika ändringar och se hur marknaden reagerar på olika protokollspecifika instruktioner eller kanaler. Då kan man behöva direktåtkomst till protokollen!

För att göra specifika ändringar på protokollnivå, tex se hur marknaden reagerar om man stänger ner en kanal eller om man vill ha flera sessioner igång samtidigt av en viss typ av kanal etc kan man prata direkt med protokollen. I version 1 finns det bara stöd för UniTP, så alla exempel kommer vara hänvisade till detta. Det går även bra att kombinera de olika nivåerna av botten för att kommunicera med börsen i enklare fall, och då det är möjligt rekommenderas detta. Det går till exempel att direkt skicka in UniTP paket i market connect med metoden *sendUTPPacket(UTPPacket p)* och det går att lägga till en MarketView som pakethanterare direkt till IC, UC och MC kanalerna

I det första exemplet skall vi just se hur modifiering av ett protokollspecifikt fält (I detta fall *UnmanagedAction*) kan ske utan att behöva hantera olika kanaler eller specialla marknadsvyer.

Ändring av en specifik protokollberoende variabel

```
importPackage(java.awt);
importPackage(java.lang);
importPackage(Packages.UniTP);
importPackage(Packages.UniTP.Messages);
importPackage(Packages.UniTP.DataFields);
importPackage(Packages.bot);

marketview = new MarketView();
market = new MarketConnect("sv-dev-02.x.ngm.se", "239.255.0.10",  11910, 1);
market.addObserver(marketview);
marketview.setVerbose(2);
market.login("ORG1","ABC1","ABC1","ABC2","ABC2","ORG1-M1");
marketview.readFromFile("saveddata");
stock = marketview.getStock("A1");

packets = UTPSimplified.getInstance();
insertOrder = packets.orderInsert(stock.getOrderbookId(),  "S",
                                        50.0 , 1000, "ORG1-M1");
```

```
insertOrder.setUnmanagedAction("D");
market.sendUTPPacket(insertOrder);

market.waitOnCatchup();
Thread.sleep(500);
market.printLine("Insert sent. Closing in unclean way...");
System.exit(0);
```

Efter att ha laddat information från saveddata.mv och hämtat ut instrumentet A1 skapar vi en säljorder genom att använda oss av objektet UTPSimplified, vilket kan skapa paket och sätta en mängd fält till standardvärden (Se dokumentationen för UniTP för närmare detaljer). Om man vill ha fullständig kontroll går det även bra att skapa paketen från grunden genom att helt enkelt skapa en ny instans från UniTP.Messages paketen. Efter att ha skapat en standardutformad säljorder så ändrar vi värdet på UnmanagedAction genom att kalla på metoden *setUnmanagedAction("D")* vilket sätter detta fält till "D" (det betyder att ordern skall tas bort om vi inte loggar ut från servern på ett korrekt sätt). Efter att ha skapat ordern skickar vi den till marknaden med kommandot *sendUTPPacket(insertOrder)*. Vi väntar därefter på konfirmation att paketet inkommit samt lite extra tid för att se om den tas bort pga ett avslut eller liknande (se varningsrutan i kapitel 4) och simulerar därefter en klientkrasch med kommandot *System.exit(0)*.

Om man vill ändra ett värde och sedan se att det faktiskt ändrades kan man i marknadsvyn använda sig av metoden *setSaveOrderValue(namnpåvariablemanvillspara)*, och den sparas då om en sådan variabel finns i orderfältet *special*. Detta värde används även när man gör jämförelse om en order skiljer sig från en annan och går enkelt att kalla på om man vill skriva ut ordern. I exemplet nedan sparas variabeln ValidThru:

```
marketview.setSaveOrderValue("ValidThru");
```

Om man vill så kan man skapa mer protokollspecifika komponenter också, om man till exempel vill se vad för trafik som sker på MC kanalen i en UniTP uppkoppling är det enklare att direkt skapa en sådan utan att skapa en marknadsvy och arbeta med filter etc. Så här skapar man en enkel lyssnare på MC kanalen:

En lyssnare på MC-kanalen

```
importPackage(Packages.UniTP);

ph = new PackageHandler();
ph.setVerbose(true);
mc = new MC("239.255.0.10", 11912, ph);
```

Kapitel

6

Mer avancerade exempel

Ibland kan de grundläggande metoderna vara lite knapphändiga eller ge svåröverskådlig information. Det finns sätt att klara av detta och här är några lite mer avancerade metoder och exempel.

En av huvudkomponenterna i den mer avancerade delarna av GTC är förmågan att filtrera paket. Både PackageHandler i UniTP paketet och MarketView implementerar filtrering av paket. Filtren skrivs i xml och följer standarden nedan (ett fullständigt filter med alla paket finns bifogat i distributionen). Ett filter kan antingen vara av typen exclude eller include, dvs den exkluderar all paket förutom de specifikt angivna, eller inkluderar alla förutom de explicit exkluderade. Om ett paket har något annat värde än ”exclude” eller ”include” i fältet *includeExclude* (tex ”noChange”) räknas det som standardangivet värde angett i regeln *includeExcludeAll*. I version 1 av GTC finns det ingen filtrering på fältdata då detta var tämligen slött med reflections. Det planeras att tillkomma under senare versioner i en mer optimierad form. För ett filter som inte skall göra något (alla paket passeras) kan null anges som filter.

Enbart IC Error Messages filter:

```
<?xml version="1.0"?>
<filter>

<rule name="includeExcludeAll" value="exclude"></rule>

<packets name="UniTP">

<packet name="UniTP.Messages.ReplyError" includeExclude="include"></packet>
<packet name="UniTP.Messages.ReplyLogOn_IC" includeExclude="noChange"></packet>

                                        .
                                        .
                                        .


<packet name="UniTP.Messages.ReplyLogOff_IC" includeExclude="noChange"></packet>
<packet name="UniTP.Messages.ReplyTrade_3" includeExclude="noChange"></packet>

</packets>

</filter>
```

Som tidigare nämnt går det mycket bra att skapa grafiska interface med hjälp av skript. MarketView implementerar JTable modellen och det går bra att göra grafiska komponenter som styr eller startar skript. Nedan är ett exempel på ett grafiskt getandsave skript som laddar ner all marknadsdata och sparar den, men även presenterar den grafiskt.

Grafiskt Interface till marknadsvy

```
importPackage(java.awt);
importPackage(java.lang);
importPackage(Packages.javax.swing);
importPackage(Packages.UniTP);
importPackage(Packages.UniTP.Messages);
importPackage(Packages.UniTP.DataFields);
importPackage(Packages.bot);

frame = new JFrame("NDX börsen");
frame.setSize(new Dimension(400,400));
frame.setLayout(new BorderLayout());
marketview = new MarketView();
table = new JTable(marketview);
sp = new JScrollPane(table);
frame.add(sp, BorderLayout.CENTER);
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
frame.pack();
frame.show();

market = new MarketConnect("sv-dev-02.x.ngm.se", "239.255.0.10",  11910, 1);
market.setVerboseFilter("icerroronly.xml");
market.addObserver(marketview);
market.login("ORG1","ABC1","ABC1","ABC2","ABC2","ORG1-M1");

market.getAllMarketData("NDX");
market.waitOnCatchup();
market.logoff();
marketview.saveToFile("saveddata");
```

De centrala skillnaderna här ar att det skapas en tabell av marknadsvyn och den läggs in i ett scrollpane vilket i sin tur läggs till i en frame. Resultatet blir en grafiskt representation av marknaden i tabellform men högst, senast, lägst och volym presenterat i ett lättöverskådligt format.

MarketView kan även jämföras med en annan MarketView. Alla ordrar och instrument som finns i representerade tas bort och kvar blir skillnaden mellan de två vyerna. Instrument som har ordrar som skiljer sig åt sparas också, medan alla ordrar som finns representerade i båda tas bort. Returneringen av användandet av *subtract(marknadsvy2)* resulterar i en lista på de ordrar som fanns i angiven marknadsvy2 men som saknades i marknadsvy1. Ett exempel som enkelt illustrerar detta:

Jämförelse av marknadsvyer

```
importPackage(java.awt);
importPackage(java.lang);
importPackage(Packages.javax.swing);
importPackage(Packages.UniTP);
importPackage(Packages.UniTP.Messages);
importPackage(Packages.UniTP.DataFields);
importPackage(Packages.bot);

marketview = new MarketView();
marketview2 = new MarketView();
marketview2.readFromFile("saveddata");
market = new MarketConnect("sv-dev-02.x.ngm.se", "239.255.0.10",  11910, 1);
marketview.setVerbose(0);
market.addObserver(marketview);

market.login("ORG1","ABC1","ABC1","ABC2","ABC2","ORG1-M1");
market.getAllMarketData();
market.waitOnCatchup();
market.logoff();

amountoforders = marketview.getOrders().size();
oldOrders = marketview.subtract(marketview2);
newOrders = marketview.getOrders();
index = 1;

marketview.setVerbose(1);

for (i = newOrders.size() - 1; i >= 0 ; --i) {
        found = false;
        for (j = oldOrders.size() -1; j >= 0 ; j--) {
            if (newOrders.get(i).getId().equals(oldOrders.get(j).getId())) {
                    market.printLine("[" + index + "]\t" +
                                        newOrders.get(i).getId() + " - " +
                                        oldOrders.get(j).getBuySell() + " " +
                                        oldOrders.get(j).getVolume() + " @ " +
                                        oldOrders.get(j).getPrice() + " -> " +
                                        newOrders.get(i).getBuySell() + " " +
                                        newOrders.get(i).getVolume() + " @ " +
                                        newOrders.get(i).getPrice());
                                        found = true;
                                        break;
            }
        }
        if (!found) {
                    market.printLine("[" + index + "]\t" +
                                        newOrders.get(i).getId() + " - " +
                                        newOrders.get(i).getBuySell() + " " +
                                        newOrders.get(i).getVolume() + " @ " +
                                        newOrders.get(i).getPrice() +
```

```
                                        " -> *NEW*");
        }
        index++;
}

for (i = oldOrders.size() - 1; i >= 0 ; i--) {
        found = false;
        for (j = newOrders.size() -1; j >= 0 ; j--) {
                if (oldOrders.get(i).getId().equals(newOrders.get(j).getId())) {
                        found = true;
                        break;
                }
        }
        if (!found) {
                        market.printLine("[" + index + "]\t" +
                                        oldOrders.get(i).getId() + " - " +
                                        oldOrders.get(i).getBuySell() + " " +
                                        oldOrders.get(i).getVolume() + " @ " +
                                        oldOrders.get(i).getPrice() +
                                        " -> *DELETED*");
        }
        index++;
}
```

I exemplet ovan skapar vi två marknadsvyer, en som vi läser in från fil (marknadsvy2) och en som vi läser in frånmarknadsservern (marknadsvy1). Vi subtraherar sedan marknadsvy2 från marknadsvy1, resultatet av operationen är de ordrar som fanns i marknadsvy2 men inte i marknadsvy1, dvs de ordrar som har tagits bort. Vi hämtar sedan ut alla ordrar ur marknadsvy1, det är ordrar som är antingen nya eller modifierade. Om ordern även finns (har samma ID) bland de ordrar som det inte fanns en identiskt match med ifrån marknadsvy2 är det en modifierad order, emedan om den inte finns där är det en ny order. Vi kan på så sätt enkelt identifiera alla nya, borttagna eller modifierade ordrar som skiljer två marknadsvyer åt:

```
[1]     20070911133316849428 - B 1000 @ 85.0 -> B 2000 @ 85.0
[2]     20070911133316851430 - B 1000 @ 84.0 -> B 2000 @ 84.0
[3]     20070911133316854432 - B 1000 @ 84.0 -> B 2000 @ 84.0
[4]     20070911133316856434 - B 1000 @ 91.0 -> B 2000 @ 91.0
[5]     20070911133316859436 - B 1000 @ 87.0 -> B 2000 @ 87.0
[6]     20070911163956132499 - S 1000 @ 99.0 -> *NEW*
[7]     20070911163926089489 - S 1000 @ 94.0 -> *NEW*
[8]     20070911133316837418 - B 1000 @ 84.0 -> *DELETED*
[9]     20070911133316840420 - B 1000 @ 83.0 -> *DELETED*
[10]    20070911133316842422 - B 1000 @ 89.0 -> *DELETED*
[11]    20070911133316845424 - B 1000 @ 87.0 -> *DELETED*
[12]    20070911133316847426 - B 1000 @ 83.0 -> *DELETED*
[18]    20070911133316865441 - S 1000 @ 98.0 -> *DELETED*
[19]    20070911133316864440 - B 1000 @ 91.0 -> *DELETED*
```

Notera att detta inte representerar all händelse mellan två marknadsvyer utan endast skillnaden. Om vi till exempel sparar marknaden och efter det lägger in två nya ordrar vilka tillsammans resulterar i ett avslut och sedan hämtar ner marknadsdatan på nytt och jämför

kommer de se identiska ut. Ordermässigt skiljer de sig ingenting åt. 2 ordrar lades in och samma två ordrar togs bort.

Kapitel

7

Utveckling av GTC

GTC kräver utveckling i takt med att UniTP utvecklas och nya paket läggs till, men den är även byggd med åtanke att andra protokoll skall kunna implementeras.

För att implementera ett nytt protokoll behöver ett antal steg tas. Först och främst så behöver en komplett implementation av protokollet erhållas. Efter detta behöver följande klasser modifieras:

MarketConnect: Samtliga metoder behöver modifieras så det utför motsvarande handligar specificerade i referensbiblioteket. Protokollet behöver även få ett unikt nummer (*int protocol*) som kan användas när objektet konstruktor anropas så objektet vet vilket protokoll som skall användas.

MarketView: GetPackage metoden behöver modifieras så den kan ta emot och förstå inkommna paket. Dessa behöver översättas till de protokolloberoende formaten Instrument och Order.