ZSQtLib

Remote Tracing via TCP/IP

Benutzerhandbuch

Copyright

Marken

Inhaltsverzeichnis

1 Einführung

Da das Debuggen von Code oft nicht ausreicht, um Fehler in einem Programm aufzudecken oder gar nicht möglich ist, weil der Fehler z.B. nur in der Release-Version der Software auftritt, bedient man sich meist Trace Ausgaben. Aber auch, um die Ausführungszeit von Programmen zu optimieren, können Trace Ausgaben sehr hilfreich sein.

Häufig werden diese Trace Ausgaben über „printf“ Anweisungen entweder auf dem Bildschirm oder in eine Datei ausgegeben. Besonders in der Embedded Software-Entwicklung sind diese „printf“ Ausgaben aber nicht ausreichend, wenn z.B. die Geräte nicht über einen Bildschirm oder ein geeignetes Speichermedium verfügen. In der Regel verfügen die Geräte lediglich über eine USB, parallele, serielle, TCP/IP oder eine anders geartete Schnittstelle.

Darüber hinaus fehlt oft die Möglichkeit, die „Tiefe“ der Ausgaben festzulegen und z.B. auf bestimmte Module, Klassen, Methoden einer Klasse oder Instanzen einer Klasse zu begrenzen.

Diesen Anforderungen soll das Trace Subsystem der ZSQtLib gerecht werden. Aber nicht nur im Bereich der Embedded Software Entwicklung kann das Remote Tracing hilfreiche Dienste leisten. Auch bei der Entwicklung von Applikationen für Windows und/oder Linux kann es von großem Nutzen sein kann, da die Trace-Ausgaben in einem von der Applikation getrennten Prozess erfolgen und auch dann noch untersucht werden können, wenn die Applikation selbst aufgrund eines Programmfehlers nicht mehr reagiert.

2 Grundlagen

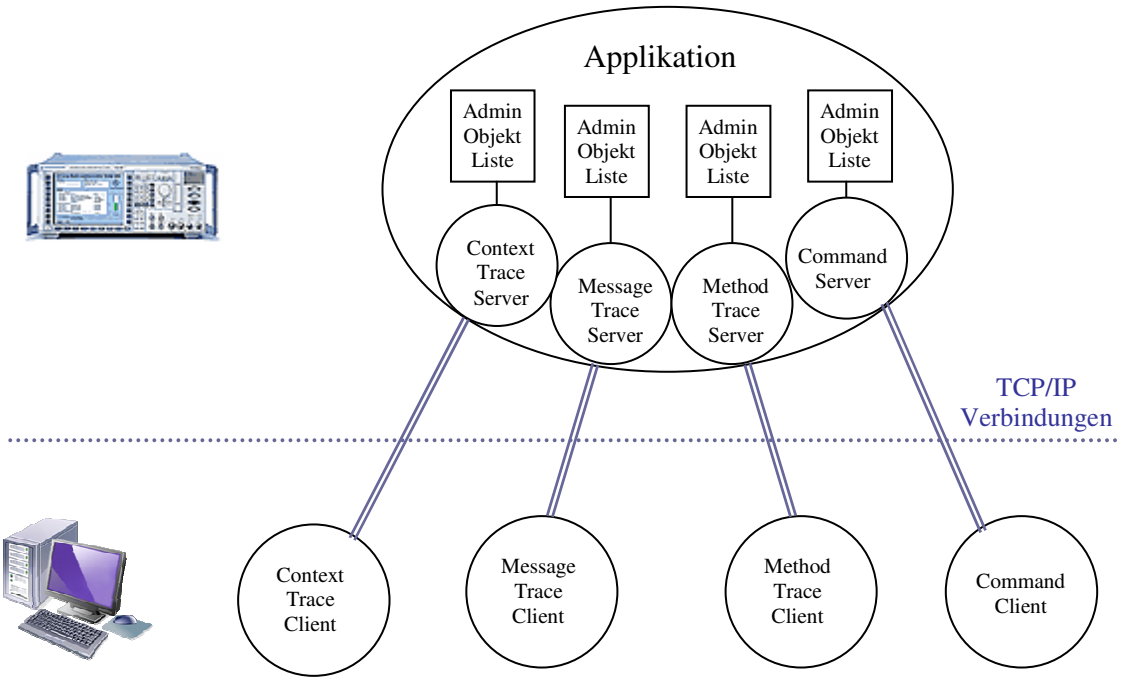
Drei verschiedene Arten des Tracings werden unterstützt:

- Ein Methoden-Trace, bei dem der Eintritt und der Austritt in Methoden protokolliert werden.
- Ein Kontext-Trace, über den eine Abarbeitungsreihenfolge protokolliert werden kann.
- Ein Message-Trace, über den insbesondere in einer Event getriebenen Multi Threaded Umbebung gesendete und empfange Nachrichten protokolliert werden.

Innerhalb der Applikation ist für jede dieser drei Arten ein Server zu instanziieren und zu starten. Über Verwaltungs- (Admin-) Objekte, die den jeweiligen Servern hinzugefügt werden, lassen sich Module, Klassen, Methoden, Instanzen, Kontexte sowie die Empfänger und Sender von Nachrichten aktivieren bzw. deaktivieren.

Für jede Tracing-Art steht eine Client-Anwendung zur Verfügung, die bei Bedarf mit dem jeweiligen Server zu verbinden ist. Beim Verbindungsaufbau fordern die Clients eine Liste der beim jeweiligen Server hinzugefügten Verwaltungsobjekte an. Diese werden dann im Client in einer Baumstruktur wiedergegeben und können so per Mausklick aktiviert und deaktiviert werden. Dabei werden Remote-Kommandos vom Client an den Server geschickt, der diese Anforderungen an das jeweilige Verwaltungsobjekt weiterleitet.

Diese Struktur bietet sich auch an, um allgemein Remote-Kommandos von einem Remote-Command-Client an die Applikation zu schicken. In diesem Fall würde jeder Remote-Befehl einem Verwaltungsobjekt innerhalb der Applikation entsprechen, das dem Command-Server hinzugefügt wird. Da auch der Command-Client beim Verbindungsaufbau eine Liste der beim Command-Server vorhandenen Verwaltungsobjekte anfordern kann, würde man so auf relativ einfache Weise eine Übersicht über alle Remote-Befehle erhalten, die die Applikation unterstützt. Da die Verwaltungsobjekte hierarchisch gegliedert sind, erhielte man auf diese Weise einen sog. Befehlsbaum, dessen Struktur eine sehr große Ähnlichkeit mit dem SCPI Kommandosatz von Messgeräten aufweist.



3 Instrumentieren der Server innerhalb der Applikation

In diesem Kapitel soll die Verwendung des Trace Subsystems der ZSQtLib anhand der Beispiel-Applikation „ZSAppTraceServer“ veranschaulicht werden.

3.1 Instanziieren und Starten der Trace-Server

In der Beispiel-Applikation werden die Trace-Server in der „main“-Funktion der Applikation als Heap-Objekte instanziiert.

```
//-------------------------------------------------------------------------------
int main( int argc, char* argv[] )
//-------------------------------------------------------------------------------
{
    int iAppResult = 0;

    ZS::Trace::CTrcServer*   pTrcServerCtx;
    ZS::Trace::CTrcServer*   pTrcServerMsg;
    ZS::Trace::CTrcServer*   pTrcServerMth;

    ZS::Examples::TrcServer::CApplication* pAppExample;
```

Innerhalb der „main“-Funktion der Beispiel-Applikation werden die Trace-Server instanziiert, bevor die Instanz der Qt-Applikation angelegt und deren Event-Loop gestartet wird. Beim Beenden der Applikation werden die Trace-Server wieder zerstört. Bevor die Trace-Server zerstört werden, wird der aktuelle Zustand ihrer Trace-Verwaltungsobjekte in einem XML-File gespeichert. Beim Start der Applikation wird das XML-File eingelesen, so dass eine neue Trace-Sitzung wieder mit den zuletzt verwendeten Einstellungen begonnen wird.

```
    pTrcServerCtx = new ZS::Trace::CTrcServer(
        /* traceType  */ ZS::Trace::ETraceTypeCtx,
        /* strObjName */ "TrcCtxServer" );
    pTrcServerCtx->recallAdminObjs("TrcCtxAdminObjs.xml");

    pTrcServerMsg = new ZS::Trace::CTrcServer(
        /* traceType  */ ZS::Trace::ETraceTypeMsg,
        /* strObjName */ "TrcMsgServer" );
    pTrcServerMsg->recallAdminObjs("TrcMsgAdminObjs.xml");

    pTrcServerMth = new ZS::Trace::CTrcServer(
        /* traceType  */ ZS::Trace::ETraceTypeMth,
        /* strObjName */ "TrcMthServer" );
    pTrcServerMth->recallAdminObjs("TrcMthAdminObjs.xml");
```

Um Trace-Verwaltungsobjekte instanziieren zu können, müssen die zugehörigen Trace-Server bereits existieren. Aus diesem Grund - aber auch, um die Kontrolle über die Reihenfolge darüber zu behalten, in der statische Objekte (Klassenvariablen) angelegt werden – werden statische Variablen in der ZSQtLib über „initDll“ und „createClassVar“ Methoden auf dem Heap angelegt. Beim Beenden der Applikation werden die statischen Heap-Variablen der Module und Klassen wieder entsprechend über „deleteClassVar“ und „removeDll“ freigegeben. Ein weiterer Grund für diese Vorgehensweise ist die Verwendung einer Memory-Leak-Detection Methode, die am Ende der Main-Funktion aufgerufen wird. Vor dem Aufruf von „dumpClientHook“ sollten alle per „new“ Operator dynamisch angelegten Instanzen wieder freigegeben worden sein, um tatsächliche Speicherlöcher aufdecken zu können. Deshalb wird auch die Qt-Applikation nicht auf dem Stack der Main-Funktion sondern ebenfalls als Heap-Objekt über den „new“-Operator angelegt, damit die Qt-Applikation und damit auch alle „Childs“ der Qt-Applikation zerstört sind, bevor die noch nicht freigegebenen Heap-Objekte protokolliert werden.

```
    ZS::System::initDll();
    ZS::Examples::TrcServer::CApplication::createClassVar();
    ZS::Examples::TrcServer::CWdgtTest::createClassVar();
    ZS::Examples::TrcServer::CTrcTestModule1::createClassVar();
    ZS::Examples::TrcServer::CTrcTestModule2::createClassVar();
    ZS::Examples::TrcServer::CTrcTestModule3::createClassVar();

    pAppExample = new ZS::Examples::TrcServer::CApplication(
        /* argc           */ argc,
        /* argv           */ argv,
        /* strOrgName     */ "ZeusSoft",
        /* strOrgDomain   */ "ZeusSoft.de",
        /* strRegSettings */ "ZSQtLib\\" + ZS::System::c_strVersionNr + "\\Examples\\TrcServer",
        /* strWindowTitle */ "ZSQtLib: Trace Server Example" );
    iAppResult = pAppExample->exec();
```

Wie bereits erwähnt werden beim Beenden der Applikation die statischen Heap-Variablen der Module und Klassen wieder über „deleteClassVar“ und „removeDll“ Methoden freigegeben.

```
    ZS::Examples::TrcServer::CWdgtTest::deleteClassVar();
    ZS::Examples::TrcServer::CTrcTestModule3::deleteClassVar();
    ZS::Examples::TrcServer::CTrcTestModule2::deleteClassVar();
    ZS::Examples::TrcServer::CTrcTestModule1::deleteClassVar();
    ZS::Examples::TrcServer::CApplication::deleteClassVar();

    ZS::System::removeDll();
```

Die Qt-Applikation und damit auch alle „Childs“ der Qt-Applikation werden zerstört, bevor die noch nicht freigegebenen Heap-Objekte protokolliert werden.

```
    delete pAppExample;
    pAppExample = NULL;
```

Bevor die Trace-Server zerstört werden, wird der aktuelle Zustand ihrer Trace-Verwaltungsobjekte in einem XML-File gespeichert.

```
    if( pTrcServerMth != NULL )
    {
        pTrcServerMth->saveAdminObjs();
    }
    delete pTrcServerMth;
    pTrcServerMth = NULL;

    if( pTrcServerMsg != NULL )
    {
        pTrcServerMsg->saveAdminObjs();
    }
    delete pTrcServerMsg;
    pTrcServerMsg = NULL;

    if( pTrcServerCtx != NULL )
    {
        pTrcServerCtx->saveAdminObjs();
    }
    delete pTrcServerCtx;
    pTrcServerCtx = NULL;
```

Am Ende der Main-Funktion wird die Memory-Leak Detection Methode aufgerufen, um mögliche Speicherlöcher aufzudecken.

```
    #ifdef _DEBUG
    _CrtDoForAllClientObjects( ZS::System::dumpClientHook, NULL );
    #endif

    return iAppResult;

} // main
```

Da sowohl das Starten als auch das Beenden der Trace-Server asynchrone Vorgänge sind, werden in der Beispiel-Applikation die Trace-Server in einer Kette von Startup- bzw. Shutdown-Methoden der von `QApplication` abgeleiteten Klasse `CApplication` gestartet bzw. beendet. Dies ist zwar nicht unbedingt erforderlich und ein einfacher Startup-Aufruf würde reichen, aber zu jeder „stabilen“ Applikation gehört ein geordneter Hochlauf und eine sauberes Herunterfahren (Aufräumen) der Applikation.

Die Trace-Server kommunizieren über TCP/IP Ports mit den Clients und verwenden folgende Default-Einstellungen für die Ports:

Server	Port
Context-Trace	1437
Message-Trace	1438
Method-Trace	1439

Diese Default-Einstellungen lassen sich mit den „setPort“-Aufrufen überschreiben. Hierfür wird im Konstruktor der CApplication-Klasse „Settings“ der Applikation eingelesen und entsprechend an die Server weitergegeben. Im Folgenden das Beispiel für den Method-Trace-Server:

```
    pServer = CTrcServer::getTrcServer(ETraceTypeMth);
    if( pServer != NULL )
    {
        uPortWrite = settings.value("Server/Mth/PortWrite",1441).toUInt();
        uPortRead  = settings.value("Server/Mth/PortRead",1442).toUInt();
        pServer->setPort(ETransmitDirSend,uPortWrite);
        pServer->setPort(ETransmitDirReceive,uPortRead);
    }
```

Die Server werden zeitverzögert über einen Timer Single Shot gestartet, damit das Main-Window sichtbar wird, bevor der eigentliche Startup-Prozess in Gang gesetzt wird. In der Beispiel-Applikation geschieht das Starten der Trace-Servers in der Methode „onStartupServers“ der Klasse CApplication:

```
//------------------------------------------------------------------------------
void CApplication::onStartupServers()
//------------------------------------------------------------------------------
{
    CSrvCltBase* pServer;
    int          idxServer;
    bool         bWaitForConfirmation = false;

    for( idxServer = 0; idxServer < EServerCount; idxServer++ )
    {
        pServer = m_arpServers[idxServer];

        if( pServer != NULL && pServer->getStateCurr() == EStateIdle )
        {
            if( !connect(
                /* pObjSender   */ pServer,
                /* szSignal     */ SIGNAL(stateChanged(QObject*,int)),
                /* pObjReceiver */ this,
                /* szSlot       */ SLOT(onStartupConServers(QObject*)) ) )
            {
                CException::throwException(__FILE__,__LINE__,EResultSignalSlotConnectionFailed);
            }
            pServer->startup();
            bWaitForConfirmation = true;
        }
    }
    if( !bWaitForConfirmation )
    {
        // Current startup process finished
        onStartupServersFinished();
    }

} // onStartupServers
```

Als „Slot“ wird die Methode „onStartupConServers“ mit dem „Signal“ „stateChanged“ der Klasse CSrvCltBase verbunden, die die Basisklasse für die Trace-Server-Klassen aber auch für die Clients dient und anschließend werden die Server über den „startup“ Aufruf gestartet. Ändert ein Server seinen internen Zustand, sendet er das Signal „stateChanged“ aus, woraufhin die Slot-Methode „onStartupConServers“ aufgerufen wird. Die Slot-Methode geht alle Server durch, die in der Liste „m_arpServers“ abgelegt wurden, und prüft deren aktuellen Zustand. Steht der Zustand auf Running bedeutet dies, dass der entsprechende Server gestartet worden ist.

```
//-------------------------------------------------------------------------------
void CApplication::onStartupConServers( QObject* i_pServer )
//-------------------------------------------------------------------------------
{
    CSrvCltBase* pServer;
    int          idxServer;
    bool         bWaitForConfirmation = false;

    for( idxServer = 0; idxServer < EServerCount; idxServer++ )
    {
        pServer = m_arpServers[idxServer];

        if( pServer != NULL && pServer->getStateCurr() < EStateRunning )
        {
            bWaitForConfirmation = true;
        }
    }
    if( !bWaitForConfirmation )
    {
        // Current startup process finished
        onStartupServersFinished();
    }

} // onStartupConServers
```

Falls alle Server gestartet wurden und sich im Zustand `Running` befinden, wird die Methode „`onStartupServersFinished`“ aufgerufen, in der lediglich wieder die Verbindung der Slot-Methode „`onStartupConServers`“ mit dem Signal „`stateChanged`“ der Server aufgehoben wird, da sie von der Klasse `CApplication` im weiteren nicht mehr benötigt wird. Erst beim System-Shutdown, der analog – aber in umgekehrter Reihenfolge zum Startup - durchgeführt wird, wird diese Signal/Slot Verbindung wieder von der Klasse `CApplication` hergestellt.

```
//-------------------------------------------------------------------------------
void CApplication::onStartupServersFinished()
//-------------------------------------------------------------------------------
{
    CSrvCltBase* pServer;
    int          idxServer;

    for( idxServer = 0; idxServer < EServerCount; idxServer++ )
    {
        pServer = m_arpServers[idxServer];

        if( pServer != NULL )
        {
            if( !disconnect(
                /* pObjSender   */ pServer,
                /* szSignal     */ SIGNAL(stateChanged(QObject*,int)),
                /* pObjReceiver */ this,
                /* szSlot       */ SLOT(onStartupConServers(QObject*)) ) )
            {
                CException::throwException(__FILE__,__LINE__,EResultSignalSlotConnectionFailed);
            }
        }
    }

    // The whole startup process is finished.
    onStartupFinished();

} // onStartupServersFinished
```

3.2 Anlegen von Verwaltungs- (Admin-) Objekten

Wie bereits erwähnt, werden in der ZSQtLib statische Variablen über „createClassVar“ Methoden auf dem Heap angelegt, um sicherzustellen, dass die Server instanziiert sind, bevor die zugehörigen Verwaltungsobjekte erzeugt werden, da diese dem jeweiligen Server hinzugefügt werden müssen.

In der Beispiel-Applikation werden z.B. für das Modul „CWdgtTest“ Trace-Admin-Objekte in der Methode „createClassVar“ erzeugt, nachdem die Server in der „main“-Methode der Applikation auf dem Heap angelegt wurden. Über „deleteClassVar“ werden die Admin-Objekte wieder zerstört, bevor die Server in der „main“-Methode der Applikation entfernt werden. Durch die „getTrcServer“ Aufrufe und anschließendem Prüfen auf NULL-Pointer wird in der „createClassVar“ Methode sichergestellt, dass die Server angelegt wurden, denen die entsprechenden Verwaltungsobjekte hinzugefügt werden sollen.

Die Verwaltungs- (Admin-) Objekte werden hierarchisch in einer Baumstruktur geordnet. Hierzu verwenden die Server die Klasse CObjPool, die zwar Bestandteil des Basissystems der ZSQtLib ist, jedoch in diesem Handbuch nicht näher beschrieben wird. Jeweils mit einem „::“ wird ein neuer Ast im Baum der Verwaltungsobjekte angelegt. Wird das Admin-Objekt verwendet, um die Methoden einer Klasse zu protokollieren, sollte dem Admin-Objekt sowohl der Namespace als auch der Name der zu protokollierenden Klasse übergeben werden, wie in nachfolgendem Beispiel verdeutlicht:

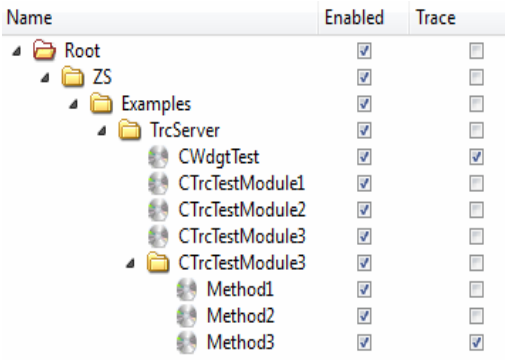
```
//-------------------------------------------------------------------------------
void CWdgtTest::createClassVar()
//-------------------------------------------------------------------------------
{
    CTrcServer* pTrcServerMsg = CTrcServer::getTrcServer(ETraceTypeMsg);
    CTrcServer* pTrcServerMth = CTrcServer::getTrcServer(ETraceTypeMth);
    CCmdServer* pCmdServer    = CCmdServer::instance();

    if( pTrcServerMsg != NULL )
    {
        s_pTrcAdminObjMsgClass = new CTrcAdminObj(
            /* traceType         */ ETraceTypeMsg,
            /* strClassNameSpace */ "ZS::Examples::TrcServer",
            /* strClassName      */ "CWdgtTest" );
    }
    if( pTrcServerMth != NULL )
    {
        s_pTrcAdminObjMthClass = new CTrcAdminObj(
            /* traceType         */ ETraceTypeMth,
            /* strClassNameSpace */ "ZS::Examples::TrcServer",
            /* strClassName      */ "CWdgtTest" );
    }
} // createClassVar
```

Zusätzliche Anmerkung bzgl. Remote-Kommandos:

Verwaltungs- (Admin-) Objekte für Remote-Befehle wären auf ähnliche Weise anzulegen, wobei hier nur der Name des Befehls sowie eine Callback-Funktion zu übergeben wäre, die dann aufgerufen wird, wenn über den Remote-Client der entsprechende Befehl an den Command-Server geschickt wird.

```
if( pCmdServer != NULL )
{
    s_pCmdAdminObjCreateModule1 = new CCmdAdminObj(
        /* strCmd       */ "ZS::Examples::TrcServer::Module1::Create",
        /* pFCmdHandler */ createModule1 );
    s_pCmdAdminObjDeleteModule1 = new CCmdAdminObj(
        /* strCmd       */ "ZS::Examples::TrcServer::Module1::Delete",
        /* pFCmdHandler */ deleteModule1 );
}
```

In der Beispiel-Applikation „ZSAppTrcServer“ werden z.B. dem Methoden Trace Server folgende Verwaltungsobjekte hinzugefügt. Auch das Objekt „ZS::Examples::TrcServer::CWdgtTest“, das in obiger „`createClassVar`“ Methode erzeugt wurde, findet sich in nachfolgender Baumstruktur wieder:

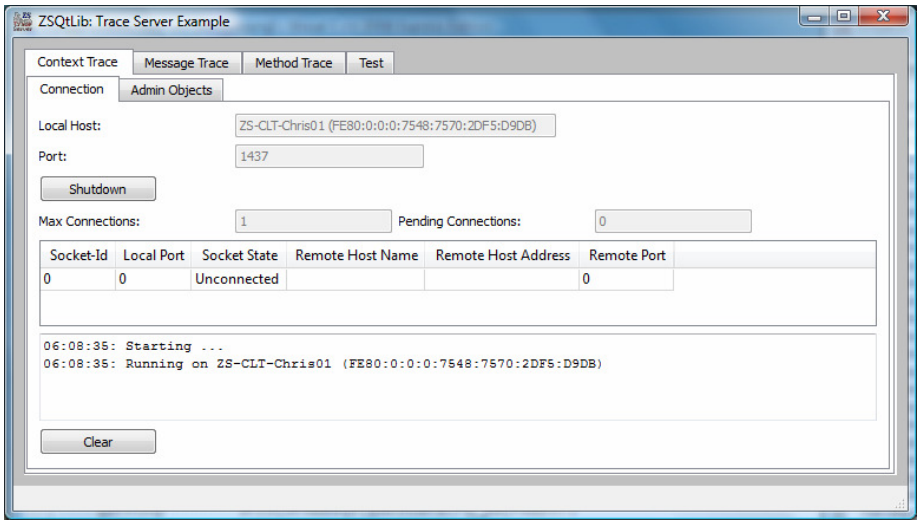


Die Check-Boxen für „Enabled“ und „Trace“ sollen die Zustände verdeutlichen, die ein Trace-Admin-Objekt einnehmen kann. Nur für Trace-Admin-Objekte, die „Enabled“ sind und deren Trace-State auf „On“ steht, werden Trace-Ausgaben auf die Clients vorgenommen.

4 Verwendung der Beispiel-Applikation sowie der Trace-Clients

4.1 Die Beispiel-Applikation „Trace Server Example“

Wenn Sie die Beispiel-Applikation „ZSAppTrcServer“ starten, zeigt sich Ihnen zunächst folgendes Bild:

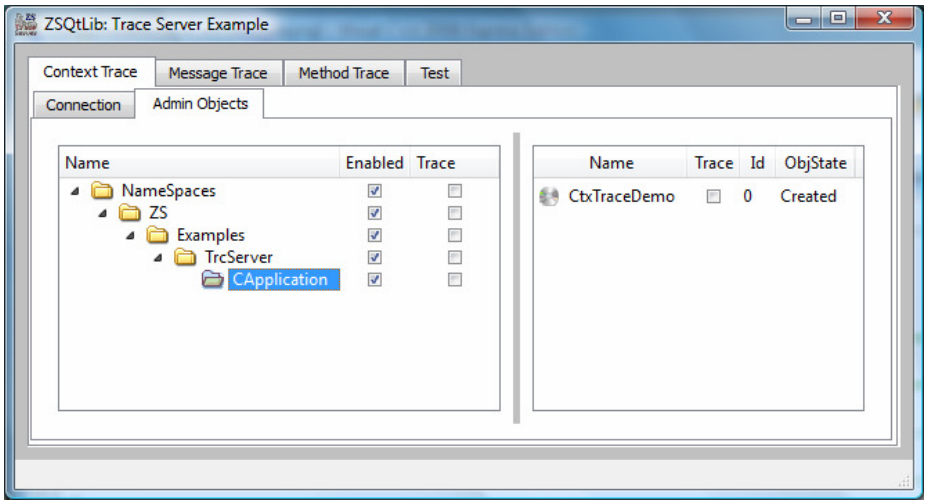


Über die oberste Reihe der Karteikarten-Reiter wählen Sie den Server aus. Im vorliegenden Fall ist der Context-Trace-Server ausgewählt.

Für jeden Server gibt es zwei weitere, untergeordnete Karteikarten. Eine „Connection-“ und eine „Admin Objects“ Karteikarte.

Über die Karteikarte „Connection“ lassen sich die wesentlichen Parameter für die TCP/IP Verbindung einstellen. Dabei handelt es sich um die Port-Nummern für den Sende- und den Empfangskanal. Die Parameter lassen sich jedoch nur Ändern, wenn der Server nicht läuft. Anhalten lassen sich die Server über den „Shutdown“ Button, der in einen „Startup“ Button wechselt, wenn der Server angehalten wurde. Im darunter liegende Fenster werden Verbindungsereignisse protokolliert.

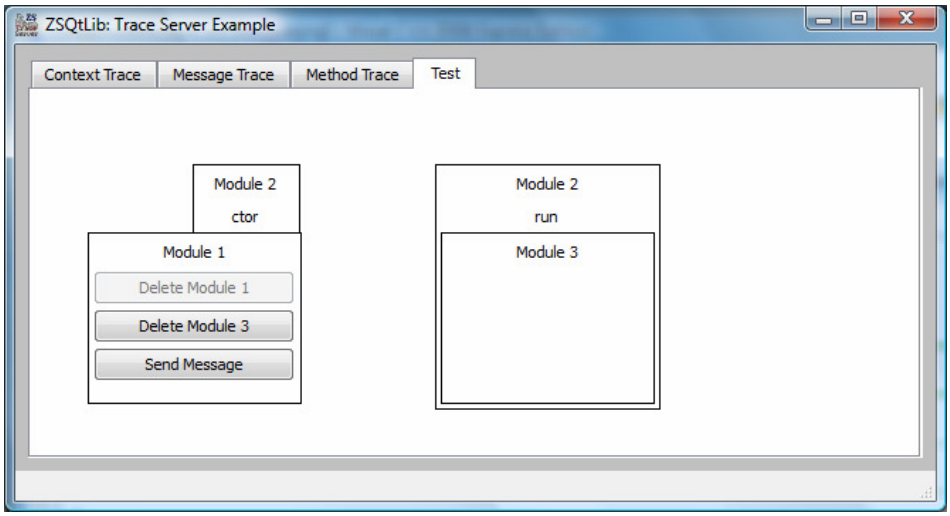
In der Karteikarte „Admin Objects“ wird über eine Baumstruktur, ähnlich wie beim Windows-Explorer, die Vewaltungsobjekte sowie deren aktuelle Zustände angezeigt, die dem jeweiligen Server hinzugefügt wurden.



Mit dem linken, dem sog. „Namespace-Browser“-Fenster, können Sie durch die Baumstruktur navigieren. Im rechtem, dem sog. „Namespace-Content“-Fenster, sehen Sie die innerhalb der Beispiel-Applikation „real“ angelegten Verwaltungsobjekte. Über die Check-Boxen „Enabled“ und „Trace“ lassen sich die Zustände der Verwaltungsobjekte editieren. Durch die Check-Boxen im „Namespace-Browser“-Fenster lässt sich die gewünschte Einstellung gleichzeitig für alle untergeordneten Äste und Verwaltungsobjekte übernehmen, falls dies von Ihnen gewünscht wird.

Der Enabled-State eines Namespaces dient dazu, auf recht einfache Weise das Tracing für alle untergeordneten Verwaltungsobjekte zu unterdrücken. Der Trace-State der Verwaltungsobjekte wird dabei nicht mit verändert. Falls Sie z.B. unter dem Knoten „CApplication“ nicht nur – wie in obigem Beispiel ein einziges Verwaltungsobjekt - sondern eine Vielzahl von Verwaltungsobjekten angelegt hätten, sie für ihre aktuellen Untersuchungen jedoch nur für eine bestimmte Auswahl das Tracing aktiviert haben, können sie über die Enabled-Check-Boxen recht schnell die Trace-Ausgaben für ihre Auswahl „disablen“ – und wieder „enablen“.

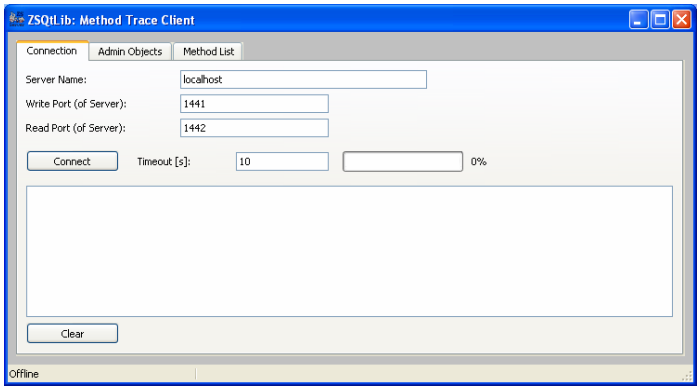
Über die letzte Karteikarte „Test“ der obersten Karteikartenreiter-Reihe können Sie über Buttons Aktionen auslösen, die Trace-Ausgaben zur Folge haben. Die Trace Ausgaben sehen Sie allerdings nur, wenn Sie den entsprechenden Client gestartet und mit dem jeweiligen Server verbunden haben.



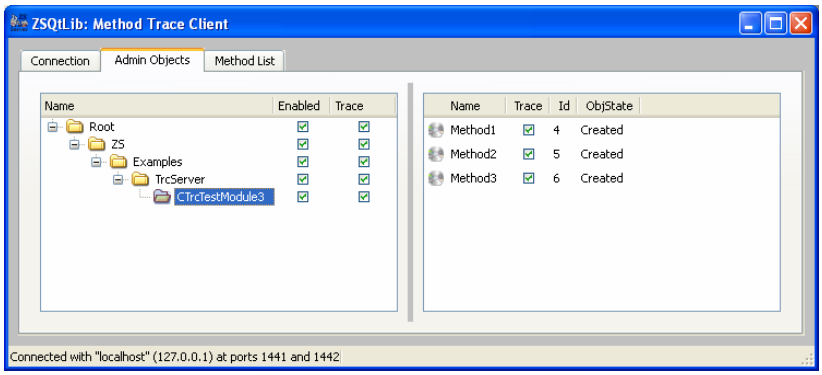
Der Inhalt der Test-Karteikarte (auch die Beschriftung der Buttons) ändert sich, je nachdem welche Aktion Sie gerade ausgeführt haben. Falls Sie auf den Button „Create Module 3“ drücken, der dann sichtbar ist, wenn Sie das „Module 1“ erzeugt haben, wird ein Timer gestartet, der eine Vielzahl von Trace-Ausgaben auf die Clients zur Folge haben soll – falls das entsprechende Verwaltungsobjekt aktiviert ist. Das „Erzeugen“ des Moduls 3 nimmt aus diesem Grund ein wenig Zeit in Anspruch… (nicht wegen der Trace-Ausgaben, sondern wegen der vielen Timer-Ticks).

4.2 Die Trace-Clients

Auch die Trace-Clients verfügen, wie die „Trace Server Example" Applikation, über eine Karteikarte zum Einstellen der Verbindungsparameter und einer Karteikarte, in der sie die Verwaltungsobjekte einsehen und editieren können. Um die Verwaltungsobjekte sehen zu können, müssen Sie den Client aber erst über den „Connect"-Button mit dem Server verbinden. Erst nach erfolgreichem Verbindungsaufbau fordert der Client die Liste der Verwaltungsobjekte vom Server an und zeigt diese in der „Admin Objects" Karteikarte an.



Im aktuellen Beispiel wurde der Method-Trace-Client mit dem Server verbunden und das Tracing für alle Verwaltungsobjekte aktiviert, indem die Trace-Check-Box des Root-Eintrags im „Namespace-Browser" Fenster aktiviert und diese Einstellung für alle „Childs" des Namespaces „Root" übernommen wurde.

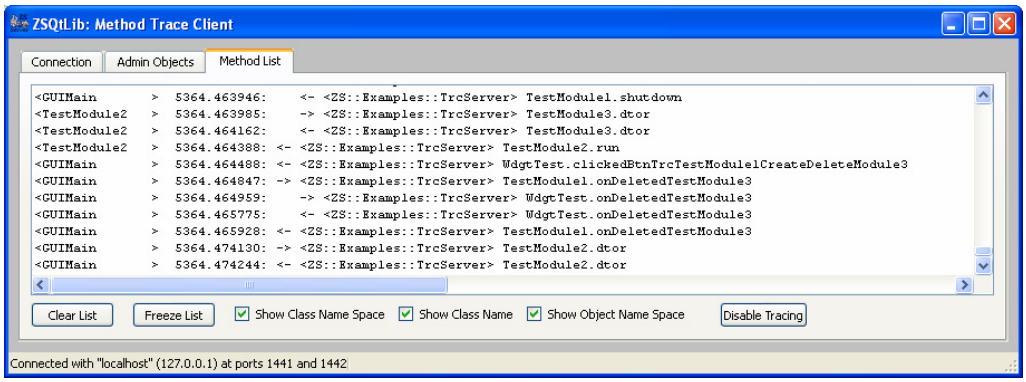


Jetzt wechseln wir zurück zur Server-Applikation in den Test-Karteireiter, drücken folgende Buttons in folgender Reihenfolge:

1. Create Module 1
2. Create Module 3
3. Delete Module 3
4. Delete Module 1

Ist Module 3 erzeugt worden, ist es möglich, auch das Message-Tracing zu testen, indem man auf den Button „Send Message" drückt. Um die zugehörigen Trace-Ausgaben zu sehen, müsste man allerdings auch den Message-Trace-Client gestartet und mit dem Message-Trace-Server verbunden haben.

Im Karteikartenreiter „Method List“ des Method-Trace-Clients werden die Trace-Ausgaben für die Methodenaufrufe protokolliert:



In der ersten „Spalte“ wird der Thread-Context ausgegeben, von dem aus die Methode aufgerufen wurde. In der zweiten Spalte wird die Systemzeit auf Sekunden normiert ausgegeben, zu der die Methode betreten bzw. wieder verlassen wurde. „–>“ markiert den Eintritt in eine Methode, „<–„ markiert den Austritt aus einer Methode. Ruft eine getracte Methode eine weitere, getracte Methode auf, wird entsprechend eingerückt.