Wer hat an der Uhr gedreht?

Leistungsanalyse

Stefan Reisner

Beim Thema Performance geht es um sehr fundamentale Dinge wie Zeit, Leistung, Last und Ressourcen. Deshalb unterscheiden sich die Performance-Probleme der Java-Welt im Kern auch in keiner Weise von denen anderer Technologien. Wir sehen heute sehr weit verteilte und in heterogenen Technologien realisierte Systemarchitekturen. Bei der Leistungsanalyse stellt sich nun im ersten Schritt die Frage, wo der Leistungsengpass überhaupt liegt. Beantworten lässt sie sich durch eine technologieneutrale Blackbox-Betrachtungsweise, die hier vorgestellt wird. Um aber aus EJB-Komponenten die dazu nötigen Messdaten zu erhalten, muss man sich zuvor ein konkretes Messverfahren für die Java-Welt aneignen.

Java-Instrumentierung

Die Leistungsanalyse eines Systems stützt sich in erster Linie auf eine besondere Art von Rohdaten, nämlich die Zeitpunkte, zu denen die Ausführung von Geschäftsvorfällen bestimmte Messpunkte passiert. Aus diesen Zeitpunkten lassen sich im nächsten Schritt die Durchlaufzeiten der Prozessabschnitte zwischen den Messpunkten berechnen. Am Rande sei bemerkt, dass man quasi in der anderen Dimension aus diesen Daten auch die Wartezeiten zwischen *aufeinanderfolgenden* Ereignissen am *selben* Messpunkt bilden kann, was Informationen über die Charakteristik der Last liefert.

Wir müssen in den einzelnen Komponenten unserer Systemarchitektur also solche Messpunkte einrichten. Das bezeichnet man auch als „Instrumentierung“. Oft ist allerdings nur ein Teil des Codes am Einsatzort neu entwickelt worden, während der Rest Legacy-Systeme oder 3rd-Party-Produkte sind. Deshalb ist eine spannende Frage, ob und wie man bereits kompilierten Code instrumentieren kann, ohne den Quellcode zu verändern (oder überhaupt zu benötigen).

Da Code auch nur eine Form von Daten darstellt, ist es im Prinzip auf so gut wie jeder technologischen Plattform möglich, den Code entsprechend zu verändern. Ob es praktikabel ist, steht auf einem anderen Blatt.

Da die Spezifikation der Maschinensprache einer Hardwareplattform in der Regel offengelegt ist, ist ein natives Executable für jeden lesbar. Bestimmte COBOL-Compiler erstellen jedoch unter Unix keinen nativen Maschinencode, sondern einen proprietären Zwischencode, der in einer eigenen Engine abläuft. Eventuell kann man dann Instrumentierungslösungen des Compiler-Herstellers nutzen. Ansonsten bleibt keine andere Wahl, als die Instrumentierung in den hoffentlich vorhandenen Quellcode der Programme einzubauen.

Auf der Java-Plattform dagegen, die ebenfalls einen Zwischencode verwendet, bestehen ganz andere Möglichkeiten. Dank aspektorientierter Programmierung kann man ohne proprietäre Werkzeuge jede Methode eines beliebigen Packages selbst instrumentieren.

Beispielinstrumentierung

Ich möchte Sie in diesem Abschnitt durch ein minimalistisches Beispiel führen. Pate gestanden hat dafür das Open-Source-Performance-Werkzeug Glassbox. Man kann Glassbox auch zur Rohdatengewinnung einsetzen, obwohl dies nicht dessen Haupteinsatzzweck ist.

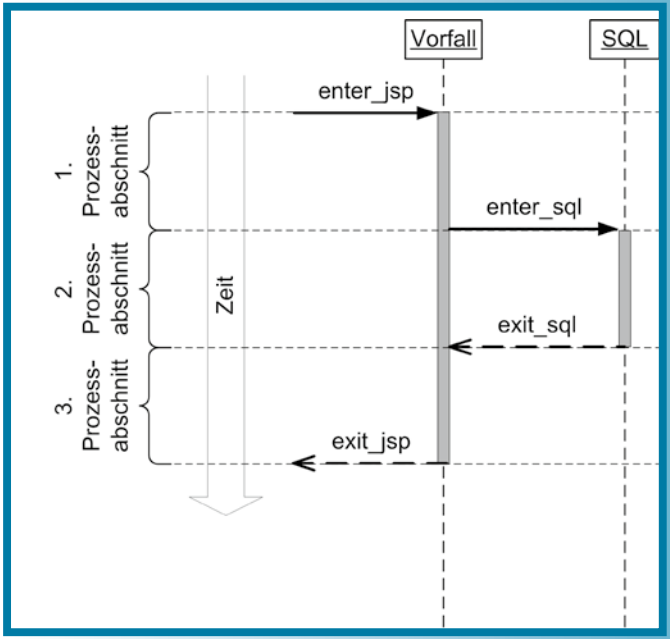

Abb. 1: Beispielprozess schematisch

Um den Prozess, den wir in diesem Beispiel untersuchen wollen, so einfach wie möglich zu halten, habe ich architektonische Tugenden ausnahmsweise außer Acht gelassen und alles in eine JSP-Seite gepackt. Diese führt ein SQL-Statement aus und zeigt das Ergebnis an. Schematisch ist dieser Prozess in Abbildung 1 dargestellt.

Dank Load-Time-Weaving ist es möglich, bereits deployte JEE-Anwendungen mittels Aspekten zu instrumentieren. Dazu muss lediglich beim Start der JVM die Option `-javaagent:aspectjweaver.jar` angegeben und die Aspekte (und deren Abhängigkeiten) müssen in den Classpath gebracht werden. Wir benötigen zwei Aspekte, um JSP-Seiten und JDBC-Zugriffe zu instrumentieren. Beide sind vom abstrakten Aspekt `Instrumentation` abgeleitet, der sich um das log4j-Handling [log4j] kümmert.

```
public aspect JspInstrumentation extends Instrumentation {
  private static AtomicInteger vorfallIdSeq = new AtomicInteger(0);

  public pointcut captureJsp(HttpServletRequest request,
    HttpServletResponse response) :
      execution(public void HttpJspPage._jspService(HttpServletRequest,
        HttpServletResponse)) && args(request, response);

  before(HttpServletRequest request, HttpServletResponse response):
    captureJsp(request, response) {
    int vorfall = vorfallIdSeq.incrementAndGet();
    NDC.push(String.valueOf(vorfall));
    threadLocalLogger.get().info(
      "enter_jsp ("+request.getContextPath()+
      request.getServletPath()+")");
  }

  after(HttpServletRequest request, HttpServletResponse response):
      captureJsp(request, response) {
    threadLocalLogger.get().info(
      "exit_jsp ("+request.getContextPath()+
        request.getServletPath()+")");
    NDC.pop();
  }
}
```

Listing 1: Aspekt JspInstrumentation

```
public aspect JdbcInstrumentation extends Instrumentation {
  public pointcut dynamicSqlExec(Statement statement, String SQL) :
      within(Statement+) && execution(* Statement.execute*(..)
      throws SQLException) && this(statement) && args(SQL, ..);

  public pointcut topLevelDynamicSqlExec(
      Statement statement, String SQL) :
        dynamicSqlExec(statement, SQL) &&
        !cflowbelow(dynamicSqlExec(*, *));
  before(Statement statement, String sql) :
      topLevelDynamicSqlExec(statement, sql) {
    threadLocalLogger.get().info("enter_sql ("+sql+")");
  }
  after(Statement statement, String sql) :
      topLevelDynamicSqlExec(statement, sql) {
    threadLocalLogger.get().info("exit_sql ("+sql+")");
  }
}
```

Listing 2: Aspekt JdbcInstrumentation

```
public abstract aspect Instrumentation {
  protected static ThreadLocal<Logger> threadLocalLogger =
      new ThreadLocal<Logger>() {
    @Override
    protected Logger initialValue() {
      String name = Thread.currentThread().getName();
      Logger log = Logger.getLogger(name);
      String filename = "instr_"+name+".csv";
      PatternLayout layout = new PatternLayout(
        "%d{yyyy-MM-dd HH:mm:ss,SSS};%x;%m;%n");
      FileAppender appender = new FileAppender(layout,filename);
      log.addAppender(appender);
      return log;
    }
  };
}
```

Listing 3: ThreadLocal Logger (ohne Exception Handling)

Im Aspekt **JspInstrumentation** (s. Listing 1) wird mittels eines **AtomicInteger**s jedem Request eine eindeutige Vorfall-ID zugeordnet. Diese wird in den log4j-NDC (nested diagnostic context) gepusht. Dadurch können die Protokolleinträge später dem richtigen Vorfall zugeordnet werden. Der Aspekt **JdbcInstrumentation** (s. Listing 2) ist sehr einfach gehalten, um nur das Prinzip zu demonstrieren. Wer den vollen Funktionsumfang des JDBC-API abdecken muss, findet in **glassbox.monitor.JdbcMonitor** eine Vorlage.

Natürlich möchte man vermeiden, dass bei höherer Last die Instrumentierung selbst das Latenzverhalten der Anwendung spürbar verändert. Dies könnte vor allem durch Serialisierung der Threads an gemeinsam genutzten Ressourcen passieren. Deshalb empfiehlt es sich, für jeden Thread eine separate Protokolldatei vorzusehen. Dazu legt man den Logger nicht wie üblich als Instanzvariable in diesem Fall des Aspekts an, sondern als ThreadLocal-Variable. Der Name der Protokolldatei wird dynamisch aus dem Thread-Namen gebildet (s. Listing 3). Ich verwende eine log4j-Formatierung, welche die Zeitpunkte in Millisekundengenauigkeit ausgibt und die Felder durch Semikolons trennt. Letzteres vereinfacht die weitere Verarbeitung des Protokolls.

Als Ergebnis erzeugt nun jedes Passieren eines Messpunktes einen Protokolleintrag mit Zeitstempel, Vorfall-ID und Bezeichnung des Messpunktes. Diese Einträge sehen aus wie in Listing 4 gezeigt.

```
2010-12-20 09:28:37,850;102;enter_jsp (response.jsp);
2010-12-20 09:28:41,663;102;enter_sql (INSERT ... INTO ...);
2010-12-20 09:28:42,678;102;exit_sql (INSERT ... INTO ...);
2010-12-20 09:28:42,678;102;exit_jsp (response.jsp);
```

➔

```
2010-12-20 09:28:45,694;103;enter_jsp (index.jsp);
2010-12-20 09:28:45,694;103;exit_jsp (index.jsp);
2010-12-20 09:28:46,803;104;enter_jsp (response.jsp);
2010-12-20 09:28:48,147;104;enter_sql (INSERT ... INTO ...);
2010-12-20 09:28:49,147;104;exit_sql (INSERT ... INTO ...);
2010-12-20 09:28:49,147;104;exit_jsp (response.jsp);
```

Listing 4: Beispielprotokoll

Keine Problem (ohne Anforderung)

Ein akutes Performance-Problem, das als „Fehler" einzustufen und abzustellen ist, kann es nur in Bezug auf eine Anforderung geben, die durch eine klare Messvorschrift festlegt, welche Leistung das System bzw. ein bestimmter Prozess erbringen muss. Wenn man auf solche Anforderungen verzichtet, sind ausufernde Analyse- und Tuning-Aufwände ohne klares Ziel („Best Effort") begleitet von zermürbenden Eskalationen vorprogrammiert.

Eine Performance-Anforderung sollte festlegen, welche Kennzahlen bei welchen Lastverhältnissen unter welchen Grenzwerten liegen müssen. Die Systemleistung in Bezug auf interaktive Funktionen wie Webanwendungen oder Webservices macht man an der Antwortzeit fest, im Gegensatz zu Batchverarbeitungen, bei denen es auf die gesamte Durchlaufzeit oder den Durchsatz ankommt. In Antwortzeitanforderungen werden sehr häufig Grenzwerte für Perzentilen [Wiki] und oft auch für den Mittelwert der Antwortzeitverteilung festgelegt.

Messdaten aufbereiten

Um uns ein Bild davon zu machen, wie es um die Anforderung bestellt ist, laden wir die Messprotokolle zur weiteren Auswertung in eine relationale Datenbank, wie z. B. die kostenlose Oracle Express Edition [XE].

Ein Ausschnitt aus der resultierenden Tabelle **MESSDATEN** ist in Tabelle 1 zu sehen. Beim Laden der Daten wurde eine zusätzliche Tabellenspalte gefüllt, die die Messung identifiziert. So kann man mehrere Messungen parallel auswerten. Das ermöglicht es, Messungen bei niedriger und hoher Last zu vergleichen. Zunächst müssen wir aus den Rohdaten (Zeitpunkte) die Durchlaufzeiten der einzelnen Prozessabschnitte bilden. Diese Arbeit lassen wir die Datenbank in einer View tun (s. Listing 5).

```
create view view_abschnitt_durchlaufzeit as
select
  von.messung,
  von.vorfall,
  von.messpunkt||'-'||bis.messpunkt abschnitt,
  bis.zeitpunkt-von.zeitpunkt durchlaufzeit
from messdaten von
inner join messdaten bis
on von.messung=bis.messung
and von.vorfall=bis.vorfall
and bis.zeitpunkt>von.zeitpunkt
```

Listing 5: View zur Berechnung der Durchlaufzeiten der Prozessabschnitte

Messdaten validieren

Ein sicherer Weg, das ganze Projekt in Unruhe zu versetzen und viel Zeit mit ins Leere laufenden Analysen zu verschwenden, ist die Arbeit mit fehlerhaften Messdaten. Eine einfache Auswertung hilft, dies zu vermeiden. Dazu trage ich für je-

MESSUNG	ZEITPUNKT	VORFALL	MESSPUNKT
Niedriglast	09:28:37,850	102	enter_jsp (response.jsp)
Niedriglast	09:28:41,663	102	enter_sql (INSERT ... INTO ...)
Niedriglast	09:28:42,678	102	exit_sql (INSERT ... INTO ...)
Niedriglast	09:28:42,678	102	exit_jsp (response.jsp)
Niedriglast	09:28:45,694	103	enter_jsp (index.jsp)
Niedriglast	09:28:45,694	103	exit_jsp (index.jsp)
Niedriglast	09:28:46,803	104	enter_jsp (response.jsp)
Niedriglast	09:28:48,147	104	enter_sql (INSERT ... INTO ...)
Niedriglast	09:28:49,147	104	exit_sql (INSERT ... INTO ...)
Niedriglast	09:28:49,147	104	exit_jsp (response.jsp)
…	…	…	…
Hochlast	10:34:12,790	10006	enter_jsp (response.jsp)
Hochlast	10:34:13,697	10007	enter_jsp (response.jsp)
Hochlast	10:34:14,463	10006	enter_sql (INSERT ... INTO ...)
Hochlast	10:34:14,541	10008	enter_jsp (response.jsp)
Hochlast	10:34:15,119	10007	enter_sql (INSERT ... INTO ...)
Hochlast	10:34:15,291	10009	enter_jsp (response.jsp)
Hochlast	10:34:15,463	10006	exit_sql (INSERT ... INTO ...)
Hochlast	10:34:15,463	10006	exit_jsp (response.jsp)
…	…	…	…

Tabelle 1: Zwei Messungen abgelegt in Tabelle MESSDATEN

den vermessenen Vorfall des Prozesses einen Punkt (Zeitpunkt, Antwortzeit) in ein XY-Diagramm ein. Der entstehenden Punktwolke kann man auf einen Blick ansehen, ob sich der Prozess über den gesamten Messzeitraum gleich verhalten hat oder ob es zeitlich begrenzte Störungen wie in Abbildung 2 gab.

In den statistischen Kennzahlen wie Mittelwert oder Perzentile geht die zeitliche Auflösung verloren. Es wäre dann rätselhaft, warum sich ein Teil der Vorfälle anders verhalten hat. Die mögliche Ursache einer zeitlich begrenzten externen Störung (z. B. durch eine Batchverarbeitung) hat man dann jedenfalls bereits ausgeschlossen. Wurden tatsächlich bestimmte Messwerte verfälscht, kann man die betroffenen Messwerte aus der Tabelle `MESSDATEN` entfernen.

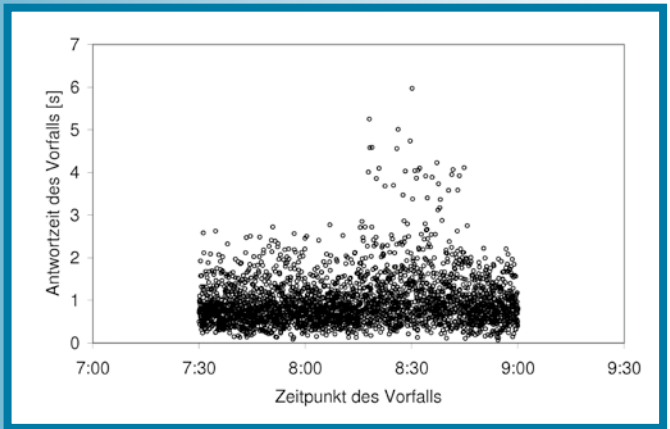

Abb. 2: XY-Diagramm zeigt zeitlich begrenzte Störung zwischen 8:15 und 8:45 Uhr

Zumindest im Test halte ich es um dieser Validierung willen für äußerst sinnvoll, zunächst *alle* Messdaten zu sammeln und nicht nur die der Langläufer, wie dies Glassbox und andere Performance-Werkzeuge üblicherweise tun, die primär für die Produktionsüberwachung gedacht sind.

Anforderung überprüfen

Nachdem wir nun valide Messdaten zur Verfügung haben, möchten wir natürlich wissen, ob unser Beispielprozess seine Antwortzeitanforderung erfüllt: Der Mittelwert soll auch bei Hochlast 0,75 s und die 99. Perzentile 2 s nicht überschreiten. Die SQL-Abfrage in Listing 6 liefert die Ist-Werte dieser Kennzahlen (s. Tabelle 2).

```
select
  messung,
  avg(durchlaufzeit) mittelwert,
  percentile_cont(0.99) within group (order by durchlaufzeit) p99
from view_abschnitt_durchlaufzeit
group by messung
```

Listing 6: Berechnung von Mittelwert und 99. Perzentile

MESSUNG	MITTELWERT	P99
Niedriglast	0,62	1,91
Hochlast	0,91	3,82

Tabelle 2: Mittelwert und 99. Perzentile der beiden Messungen (Einheit Sekunden)

Während also in der Niedriglastmessung alles im grünen Bereich ist, zeigt die Hochlastmessung ein Performance-Problem. Nun beginnt die Suche nach dem Bottleneck.

Suche nach dem Bottleneck

Ich spreche grundsätzlich von einem Bottleneck, wenn eine Performance-Anforderung verletzt ist. Zu viele Vorfälle dauern also zu lange. Die einzige Erkenntnis, die man zu diesem Zeitpunkt hat, ist die Lasthöhe, ab der sich das Bottleneck zeigt. Wir wissen noch nicht, *wo* im Prozessablauf das Bottleneck liegt, und schon gar nicht, welcher Natur es ist. Es kann sein, dass der Prozess eine Ressource beansprucht, von der es zu wenige gibt, also z. B. einen Thread, oder einen Datensatz mit sperrendem Zugriff. Oder es kann sein, dass ein Prozessabschnitt lange dauert, weil er sehr viele Rechenoperationen nacheinander ausführen muss.

Es liegt nahe, nun die Extremfälle der Messreihe herauszusuchen und näher zu untersuchen. Das führt jedoch nicht selten in die Irre. Der Grund ist, dass Antwortzeiten grundsätzlich schwanken und dass Langläufer immer wieder einmal auftreten, ohne dass dies ein Problem aus Sicht der Anforderung darstellt. Mehr als eine Handvoll Einzelfälle zu untersuchen, kostet viel Zeit, und der Erkenntniswert ist dann immer noch nicht sichergestellt, weil sich oft kein eindeutiges Bild ergibt. Es gibt einen besseren Weg, bei dem wir die gesammelten Messdaten insgesamt auswerten.

Die Grundlage sind die Instrumentierungsmesspunkte. Sie unterteilen den Gesamtprozess, der mit dem Eingang des http-Requests beginnt und mit der vollständig übertragenen Ergebnisseite endet, in Abschnitte (s. Abb. 1). Das Bottleneck sitzt in dem Abschnitt mit dem größten Beitrag zum Mittelwert oder zur Perzentile. Dort sollte die Optimierung ansetzen. Diese Beiträge zu ermitteln, ist also unser Ziel.

In Bezug auf den Mittelwert ist das einfach, denn der Beitrag eines Prozessabschnittes zum Mittelwert ist – sein Mittelwert. Perzentilen verhalten sich jedoch nicht in dieser Weise additiv. Hier behelfe ich mir mit einem „Was-wäre-wenn"-Ansatz: Wie würde die 99. Perzentile ausfallen, wenn ein bestimmter Prozessabschnitt überhaupt keine Zeit beanspruchen würde?

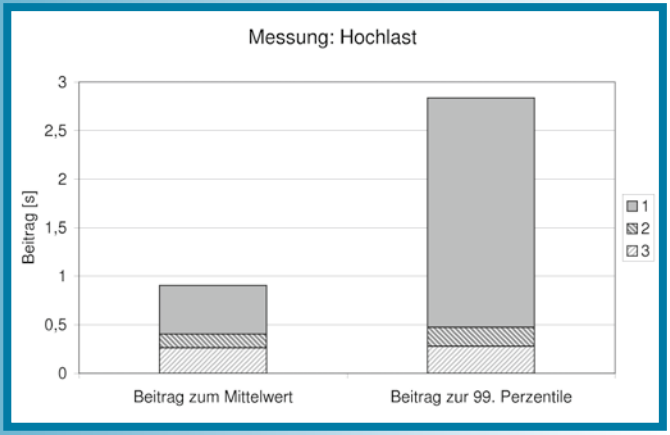

Abb. 3: Lokalisierung des Bottlenecks im 1. Prozessabschnitt

Dazu rechne ich seinen Beitrag aus jeder Antwortzeit heraus und bilde dann die Perzentile dieser „Was-wäre-wenn"-Antwortzeit. Die Differenz zur tatsächlichen Perzentile ergibt den Beitrag dieses Prozessabschnittes. Das Ergebnis dieser Auswertung für alle drei Prozessabschnitte zeigt Abbildung 3.

Der Übeltäter ist nicht etwa der JDBC-Zugriff im zweiten Prozessabschnitt. Mit über zwei Sekunden trägt vielmehr der erste Prozessabschnitt am weitaus meisten zur 99. Perzentile bei. Somit ist hier das Bottleneck zu suchen. Auch zum Mittelwert ist sein Beitrag am größten (was übrigens nicht zwangsläufig der Fall sein muss). Die genauere Analyse würde in diesem Beispiel ergeben, dass in diesem Prozessabschnitt ein Webservice aufgerufen wird, der auf einem zu leistungsschwachen Server läuft. Mit einer Instrumentierung des verwendeten Webservice-Frameworks (wie z. B. Axis2) könnte man diesen Aufruf in einem verfeinerten Test auch isoliert vermessen.

Wird ein Prozessabschnitt wie hier erst ab einer gewissen Last zum Bottleneck, kann der Administrator der betroffenen Komponente meist aufgrund ihrer Ressourcenauslastung erkennen, was das Problem ist. Bottlenecks sind jedoch nicht ausschließlich ein Lastphänomen. In sehr zeitkritischen Anwendungen kann die Anforderung durch Antwortzeitschwankungen durchaus bereits bei geringster Last verletzt werden. Weitere statistische Analyseverfahren helfen, die Herkunft dieser Schwankungen aufzuklären.

Zusammenfassung

In diesem Artikel habe ich versucht, in sehr komprimierter Form zu zeigen, wie man die Java-Plattform instrumentieren und welche Erkenntnisse man aus den Messdaten gewinnen kann. Weitere wichtige Auswertungen befassen sich mit der Lastgrenze des Systems sowie mit der Übertragung von Performance-Aussagen vom Test auf die Produktionsumgebung. Hierzu mehr in der weiterführenden Literatur.

Literatur und Links

[AspectJ] `http://www.eclipse.org/aspectj/`
[Glassbox] `http://glassbox.sourceforge.net/glassbox/Home.html`
[log4j] Apache log4j, `http://logging.apache.org/log4j/`
[Reis10] St. Reisner, IT-Performance richtig testen und optimieren, entwickler.press, 2010
[Wiki] `http://de.wikipedia.org/wiki/Perzentile`
[XE] Oracle Express Edition, `http://www.oracle.com/technetwork/database/express-edition/overview/index.html`

Dr. Stefan Reisner absolvierte Studium und Promotion an der Universität Köln im Fach Physik. Seit 1999 ist er als IT-Consultant, seit 2002 als Senior Consultant bei der syngenio AG tätig. Seine Tätigkeitsschwerpunkte liegen in den Bereichen Performance-Analyse, Testmanagement und Testautomatisierung.
E-Mail: Stefan.Reisner@syngenio.de