Intelligent Service Trading and Brokering for Distributed Network Services in GridSolve

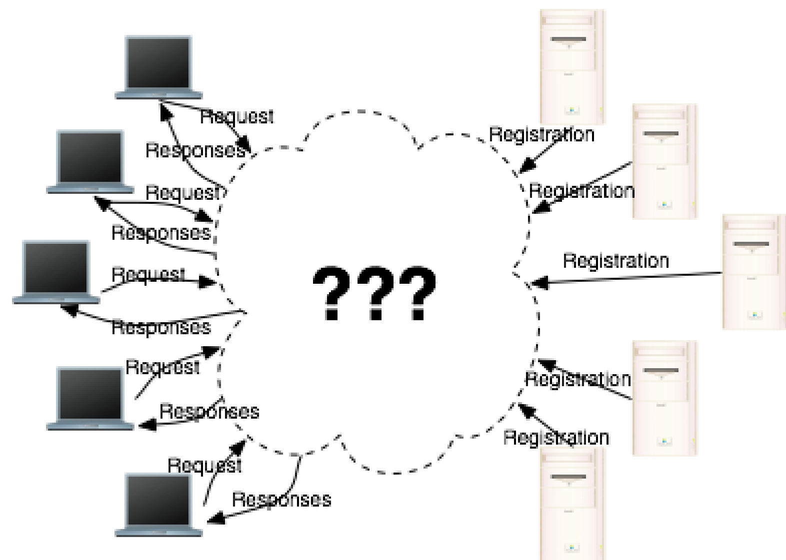
Aurélie Hurault[1], Asim YarKhan[2]

[1]IRIT - University of Toulouse

[2]ICL - University of Tennessee

June 2010

The problem



Introduction

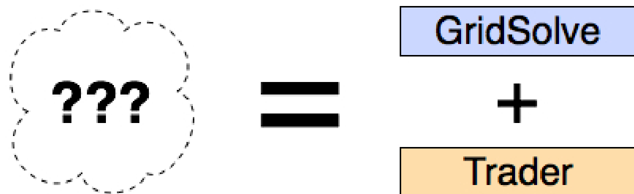
- Globalization of resources
  - Computers
  - Libraries
- Computational aspect
  - Multiple libraries
  - Different interfaces
  - Functional / non-functional aspects
- Functional aspects
  - Calculation carried out by the service
  - Solved by the trader
- Non-functional aspects
  - Performance, memory capacity necessary, quality of service, computing time, ...
  - Solved by grid middle-ware

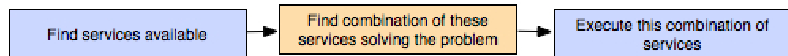
Goal

1. The user express his problem as naturally as possible.
2. The trader finds all the services or combinations of services which answer the request, and specify the value of the parameters.
3. The trader interacts with a middle-ware to find the most interesting solution.
4. The execution result is given back to the user.

Transparency for the user

Our proposition





The trader

- Trading mechanism :
  1. Specialist : description of the service functionality.
  2. User : description of the required functionality.
  3. Algorithm : comparison of the descriptions to find the services and combinations of services which solved the problem.
- Current descriptions :
  - Service names, signatures, keywords and/or ontologies
  - Limits :
    - Service name : knowledge of the nomenclature
      BLAS : SGEMM, E04-NAG : E04ABF
    - Signature : not precise enough
      addition and multiplication : Matrix $\times$ Matrix $\rightarrow$ Matrix
    - Keywords : ambiguous
      BLAS, SGEMM : $\alpha * A * B + \beta * C$
    - Ontologies :
      Difficult to solve our problem with usual logic
      No control of the solver

The trader : input

- A description of the domain as an algebraic specification, with sub-sorting and equations for the operators properties.

$S = \{Scalar, Matrix\}$

$$
\begin{aligned}
\Sigma = \{ & \\
0, I\ &: Matrix \\
0, 1\ &: Scalar \\
+\ &: Matrix \times Matrix \rightarrow Matrix \\
+\ &: Scalar \times Scalar \rightarrow Scalar \\
*\ &: Matrix \times Matrix \rightarrow Matrix \\
*\ &: Scalar \times Matrix \rightarrow Matrix \\
*\ &: Scalar \times Scalar \rightarrow Scalar \\
\} &
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E} = \{ & \\
& Matrix\ x, Matrix\ y : x + y = y + x \\
& Matrix\ x : 1 * x = x \\
& Matrix\ x : I * x = x \\
\} &
\end{aligned}
$$

The trader : input

- A description of the services as terms under the algebraic specification.
  - $Matrix\ x,\ y : serv_1(x,y) = x + y$
  - $Matrix\ x,\ y : serv_2(x,y) = x * y$
  - $Scalar\ \alpha, Matrix\ x : serv_3(\alpha,x) = \alpha * x$
  - $Scalar\ \alpha, \beta, Matrix\ x,\ y,\ z : serv_4(\alpha,x,y,z) = \alpha * x * y + \beta * z$
- A description of the request as a term under the algebraic specification.
  - $Matrix\ a,\ b : a + b + a$

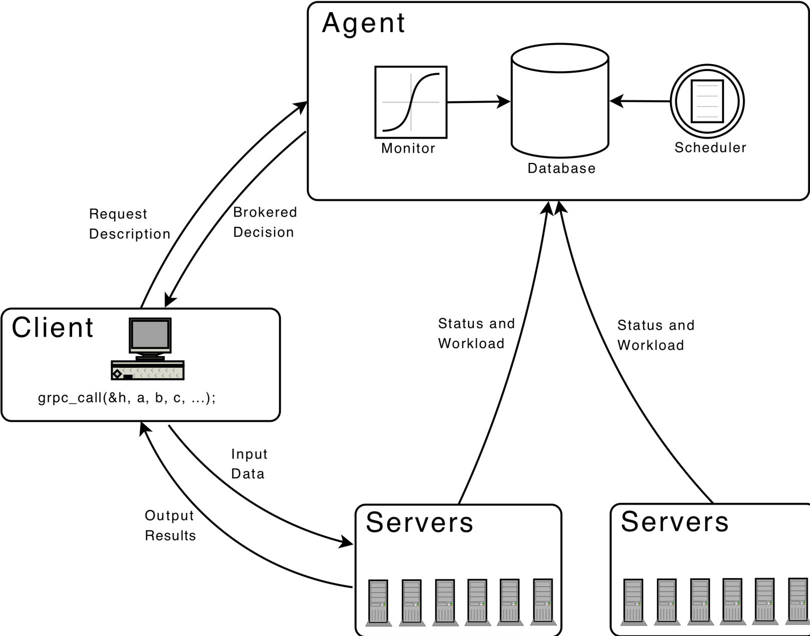
The trader : output

- A list of services and combination of services that solve the request.
  - `saxpy(a,saxpy(b,a))`
  - ```
    affect, res2, copy a
    affect, res1, copy a
    call, saxpy, m(b)*n(b), 1.0, copy b, 1, res2, 1
    call, saxpy, m(res2)*n(res2), 1.0, res2, 1, res1, 1
    ```
    ```

GridSolve

- Purpose : create the middleware necessary to provide a seamless bridge between computational scientists using desktop systems and the rich supply of services supported by the Grid architecture.
- Goal : the users of desktop systems can easily access and reap the benefits (in terms of shared processing, storage, software, data resources, etc.) of using grids.
- Respect GridRPC standard.

GridSolve



Integration

1. GridSolve provides the available services.
2. The trader find the services and combination of services that solve the user request.
3. The output of the trader is analyzed and the services called.
4. The response is given back to the user.

Integration : Get the available services

- GridSolve give back the available services.
- The description of the services are parsed to compute one of the input of the trader.
- The request is also parsed to compute the other input of the trader.
- We supposed the user give well typed expressions (doesn't check the symmetry, the invertibility,... of the matrices).

Examples of additional description

```
SUBROUTINE dgemm
APPLICATION_DOMAIN="LinearAlgebra"
TRADER_DESCRIPTION="
c <-((alpha*((op transa a)*(op transb b)))+(beta*c)) ;
value m = (nbRow c) || (nbRow (op transa a)) ;
..."

SUBROUTINE dsymm
APPLICATION_DOMAIN="LinearAlgebra"
PARAMETERS_PROPERTIES = "a symmetric"
TRADER_DESCRIPTION="
c <- if (side='l') then ((alpha*(a*b))+(beta*c))
     if (side='r') then ((alpha*(b*a))+(beta*c)) ;
value m = (nbRow c) ;
...
if ( a instanceof UpTriInvMatrix ) then ( uplo = 'u' );
..."
```

Integration : Get the solutions

- Do a call to the trader.
- Choose the "best" solution.
- Write a file as output.

```
affect, res2, copy a
affect, res1, copy a
call, saxpy, m(b)*n(b), 1.0, copy b, 1, res2, 1
call, saxpy, m(res2)*n(res2), 1.0, res2, 1, res1, 1
```

Integration : Choose the best solution

- In the GridSolve idl : a field "COMPLEXITY" ;
- The complexity is an arithmetic expression using the name of the parameters ;
- Example

```
SUBROUTINE dgemm
[...]
COMPLEXITY = "2.0*m*n*k+2.0*m*k"
```

- In linear algebra, the value used are mostly constants and size of the matrices ;
- The size of the matrices are given to the trader ;
- For each sub-request, the trader calculate the overage of the matrices involved ;
- For each service, the approximative cost is calculate, with the overage, to know if a best solution is already known.

Integration : Call the services

- Analyze of the output of the trader.
- Do the call.
- When there is several calls : use the DAG optimization of GridSolve

Integration : Results

Different possibilities

- Popped from the argument stack
- Printed
- Stored in a Matlab variable

The C API

- The C API
  int gs_call_service_trader(char *req,...);
  int gs_call_service_trader_stack(char * req, grpc_arg_stack *argsStack);
- The C call

```
float *a = malloc (sizeof(float)*nbla*nbca);
float *b = malloc (sizeof(float)*nblb*nbcb);
...
gs_call_service_trader("(a+(b+a))","a",a,nbla,nbca,...)
```

- The result

```
12.000000 24.000000 36.000000
48.000000 60.000000 72.000000
84.000000 96.000000 108.000000
```

The Matlab interface

- The Matlab interface
  int gs_call_service_trader(char *req);
- The Matlab call

```
a=[1,2,3;4,5,6;7,8,9]
b=[10,20,30;40,50,60;70,80,90]
[output]=gs_call_service_trader("(a+(b+a))");
```

- The result

```
output =
   12.  24.  36.
   48.  60.  72.
   84.  96.  108.
```

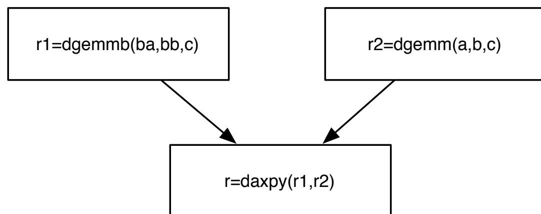
- The variable "output" can be used later.

Example

- Request : $(((a * b) + c) + ((ba * bb) + c))$
- a, b and c are 3x3 matrices
- ba is a 3x3000 matrix
- bb is a 3000x3 matrix.
- Using the Matlab interface:

```
gs_call_service_trader("(((a*b)+c)+((ba*bb)+c)))")
```

- Result :



- Where : dgemmb use the Strassen-Winograd algorithm (efficient for "big" matrices) and dgemm use the "classic" algorithm.

Conclusion

- Work done
  - Combination of a trader and a middleware.
  - Transparent for the user.
  - Try to be efficient : take into account the complexity of the services and parallelize the calls if possible.
  - Two interfaces : C or Matlab.
- Application
  - Doing quick test with some unknown environments.
  - Prototyping applications.
  - Using transparently resources even if there are in evolution
    - Modification of libraries
    - appearing / disappearing of resources / services
- Limitations / Futur work
  - Application domains limited to the ones that can be described with algebraic specification.
  - Complexity of the services
  - Performance of the trader.