Remote Data Access for Interactive Visualization

Steffen Prohaska
Scientific Visualization Dept., Zuse Institute Berlin (ZIB)
prohaska@zib.de
Andrei Hutanu
Center for Computation and Technology, Louisiana State University (LSU)
ahutanu@cct.lsu.edu

Abstract

This paper investigates requirements for remote data access when used in an interactive visualization environment. A conceptually simple interface to deal with persistent and transient data that are organized on structured grids is specified. Although some parts of the interface are stated only very generally, or as requirements, a first prototype confirms that efficient utilization of network resources should be achievable for a wide range of applications.

1. Introduction

Remote visualization of large data is a challenging task. Distributing the visualization pipeline (1) in different ways might be optimal depending on parameters of the visualization algorithms (2; 3; 4). Reliable network protocols, like TCP, might be slow compared to unreliable protocols, like UDP (5). Network latency might degrade network performance if roundtrips are required for each access to a synchronous API (6). Visualization algorithms might require data reordering and optimized data structures to be efficient, which is especially true for out-of-core processing (7). These examples might suggest that specialized solutions are required for each use case. On the other hand, building reliable software from scratch is hard and error prone. Using common knowledge in the form of established APIs and design patterns is preferable.

Today, simulations running as batch jobs and measuring devices acquiring physical data generate massive data sets. In most cases, it is not possible to transfer complete data sets to the scientist's workstation. Often, simulations do not even store all data to disc due to limited resources. Data handling and processing is mostly guided by the simulation or measuring process. Data safety or a simple storage order might be primary concerns. Visualization might only be regarded as a secondary goal, although interactive visualization and steering of running simulations might simplify the process of understanding and help debugging simulation programs (8; 3). Besides transferring images or geometry, remote data transfer of filtered data is one way to create remote visualizations.

When dealing with large remote data, two questions must be answered. Firstly, what is the most appropriate API or data access language to select the remote data? Secondly, what is the most efficient data transport protocol that should be used for the given use case. In this article, we are investigating the remote access to three-dimensional data organized on structured grids, with optional time dependency, in a way, which is suitable for interactive visualization. Structured grids are one of the most basic ways to organize multi-dimensional data. Below we will discuss a couple of standardized APIs to access (and store) such data, like HDF5 (9), NetCDF (10), and DAP (11).

In conclusion, we propose a flexible data selection mechanism using SOAP-based remote procedure calls. The proposed API separates the data access and data transport tasks. The proposed protocol has the potential to enable maximum data transport performance by putting only minimal requirements on the data transport layer. It is our intention to leave the exact specification of the data transport protocol open in order to be able to use the best available protocol for a given client-server configuration.

2. Visualization Use Cases

Exploratory visualization—opposed to analytical and descriptive visualization (12)—is an interactive process. To be useful, a system for exploratory visualization should guarantee response times. As network bandwidths and latencies might vary over a wide range and most of the times no Quality of Service is guaranteed, visualization applications that use remote data access, might only hope to sustain an average response time in such an environment. They are more similar to video playback over the Internet than to reliable data transfer. To hide the network nature data must

be retrieved in the background. Software APIs should support asynchronous data access. In an interactive system, the user may change parameters, such as subvolume or targeted resolutions, at any time. Already requested data might become unnecessary due to these changes. APIs should therefore support a mechanism to cancel operations.

We restrict our following discussion to data organized on a three dimensional structured grid and time series of data on such a grid. We consider operations required for orthogonal slicing, volume rendering, and retrieval of subvolumes for further processing. For interactive visualization, data sets should be available in different resolutions allowing retrieval of low-resolution overviews as well as high-resolution details. We consider two basic types of data sources. Firstly, persistent data sets, which can be accesses at any time. In a preprocessing step lower resolution representations of the data might have been generated. The second type of data is transient data, which has a limited life time and will not be available in the future. Important examples are active simulations, which either do not store every time step, or store results to write only storage that becomes available for reading only after the end of the simulation. However, the current simulation step might be available for visualization. Sensors delivering a persistent stream of data are another example.

Next, a couple of interactive visualization scenarios are discussed to better understand the needs of such applications. The aim is to conclude with guidelines for designing a simple software interface for remote data access for interactive visualization, which helps in efficiently utilizing network bandwidth; guaranteeing responsiveness of the application; and facilitating portability to different environments, like local or wide area network, low bandwidth or high bandwidth, workstation or super-computer.

The first scenario is hardware based volume rendering with a limited memory budget adjusted to client resources. To provide overview and details at the same time, a focus point might be specified. The renderer will hierarchically arrange higher resolution data around this point and lower resolution data in the complete volume. A potential solution for persistent data is to start retrieving low-resolution versions first, continue with high-resolution versions around the focus point, and continue retrieving data until the memory budget is consumed. This will reliably present the same visualization to the user if the same parameters and the same data set is used again. As data blocks arrive, the visualization is updated to present preliminary results. For similar scenarios see (6) and references therein.

With transient data, the situation is more difficult. The client could send a list of all blocks that are inside the memory budget to the server. The server would send all available blocks. Low-resolution data and blocks around the focus point should be sent first. Ideally a running simulation should not block because of a visualization request. Therefore, it would send only as many blocks as possible and would not check if the blocks are reliably received by the client. The number of blocks received by the client might be limited by a resource, e.g. network bandwidth. As above, the volume renderer updates the visualization as new data becomes available, but this time, the result might change from run to run. To allow monitoring the simulation, new blocks could be sent for every time step and the rendering would update accordingly. Old blocks, which are not replaced, should be deleted after a limited lifetime. At any time a user should be allowed to change the focus point and the system should be able to change the priority of the blocks. The spirit of the described system is inspired by the description of Visapult (5), although the details are different.

The next scenario describes an orthogonal slice, i.e. a subblock of the structured grid with one index kept fixed. We assume that a full resolution slice can easily be stored at the client. Potential solutions are similar to the solution discussed for volume rendering. If data are persistent, the visualization application could retrieve them resolution level by resolution level. One level might be split into subslices, which may arrive in any order. A transient data source could, as above, just send available data, potentially following a prioritization scheme.

Retrieval of sub-volumes at user specified resolution is easily possible if the data are persistent. For transient data the sub-volume might be split into smaller blocks which are sent by the server without validating their arrival at the client. Such partial results might nonetheless be useful to inspect result early and identify problems or mark volumes for further analysis based on the final result data set.

3. Interface Requirements

In the previous section, a number of visualization use cases were discussed. In all examples, requests for subvolumes are sent to a server and binary blocks are received asynchronously, in the background. The requests might be complex, e.g. they might contain a list of blocks at various resolutions together with a prioritization scheme. Therefore, the protocol used to describe the request should be capable of passing complex data structures. The server should reliably receive the requests.

In response, the server starts to deliver blocks of binary data. Depending on the data source, such blocks might be generated and sent by the server, and received by the client reliably or unreliably. To avoid network latencies, the server should have a queue of pending requests at any time, and push data to the client. If the selection or the focus of the visualization is changed by the user, the client might need to cancel already sent requests and replace them

with new ones. It might be sufficient to cancel all pending requests and Reset the data pipe to its initial state.

If delivery of data is not reliable, no order can be enforced on the blocks. The client has no guarantee that blocks which are sent first, will arrive first. Even if blocks are delivered reliably, a client might accept out-of-order arrival. For example, a set of sub-volumes of the same resolution could be useful in any order. However, the order might be restricted to fulfill some general requirements, e.g. low resolution before high resolution.

The next section discusses existing solutions for remote data access in the light of these requirements.

4. Related Work

There are many remote data access architectures and protocols in use by the scientific community today. We will try to enumerate some of the most representative grouped by the kind of features that are of interest for us.

4.1. Data Selection

Many of the file access libraries and APIs provide useful mechanisms for data selections. A widely used mechanism, present in similar forms in HDF5 (9), netCDF (10) and OpenDAP (11) is the hyperslab selection mechanism. A hyperslab is a selection performed on a multidimensional (n dimensions) data set that represents a repetition of a k selected, l unselected elements, repeated by n times on each dimension (k, l, n potentially different for each dimension). A hyperslab is thus represented by 4 multi-dimensional vectors storing the starting point (also denoted as origin) of the selection, the number of selected and unselected elements on each dimension, and the number of repetitions on each dimensions. The main issue is that these libraries usually provide these useful selection methods only for a specific file format (HDF5 for the HDF5 file format, netCDF for the netCDF file format). We propose the usage of this selection mechanism in a file-format independent way, similar to OpenDAP.

4.2. Remote Access

The main differences between the data access libraries appear when dealing with remote data. While high-level selection operations are equally important for remote data access, some of the existing remote access libraries lack such capabilities. Systems like the storage resource broker (SRB) (13), and older but still widely used systems like FTP and NFS provide only a common view on the data objects as files. These are only of little interest for us and we will concentrate on some of the architectures that provide higher-level selection capabilities.

MPIIO, defined in the MPI-2 standard (14), provides a way of defining "views" of a file, where a view is a simple selection pattern defined by a displacement, the definition of an element, and the repeating pattern of selected/unselected elements in a file. The access API is MPI; the remote protocol is implementation-specific.

The visitdata access library (15) uses a push model. The simulation sends the data to be visualized to the visualization client. While this is not the scenario we are targeting, visit makes a useful distinction between the data selection and transport mechanisms and allows the integration of various data transport mechanisms. The data access protocol is defined in C, Perl and Fortran and the data transport protocol can be any protocol supporting in-order, reliable data transport (like TCP).

The Active Data Repository (16) provides a flexible way of organizing and retrieving large data. In its model, the input data is organized to match the needs of a distributed analysis algorithm in the best possible way. The data is indexed and stored according to a schema that provides the best performance for a given algorithm. Data sets can be partitioned in chunks but cannot be subsampled directly using ADR. The data access interface is defined in C++ and accepts regular expressions when searching for data sets. The remote interface is internal to ADR.

DataCutter (17) provides an efficient system for subsetting and filtering large data sets. It provides an indexing schema, and uses a filter-stream programming model for the user-definable filters. DataCutter uses a reliable stream model and could be used by our system to access archived data on the server side.

GridFTP (18) is an extension of the FTP protocol. As in FTP, it separates the control and data channels and as an enhancement, it provides support for parallel TCP streams and striped transfer. The data channel can thus be formed by multiple TCP streams. GridFTP also includes a flexible server-side processing feature that allows a client to specify various processing operations using a string.

HTTP (19) can encode data selection operations either using the POST mechanism, or by encoding the selection in the URL as done in OpenDAP (11). Data transport is performed using HTTP media attachments (MIME). The same mechanism may also be used in SOAP (20; 21), a XML-based protocol for remote procedure calls. An enhancement of this mechanism is represented by the usage of SOAP in DIME (22) messages as defined in the WS-Attachments specification (23), resulting in a series of improvements when doing large data transfer but its possibilities are limited by those of a one-way TCP pipe.

The HDF5 library was not designed having remote data access as a primary goal. Although, its flexible I/O driver plugin mechanism (Virtual File Drivers) allows usage in a distributed environment. The main limitation of this mech-

anism is the lack of support for high-level operations. With small modifications to the HDF5 internals we were able to include partial support for high-level selection operations in the Virtual File Driver (6). In this way, the HDF5 high-level selection API can be used for remote data access in an efficient way. Still, HDF5 remains limited when dealing with remote data, since it lacks an asynchronous API and can only process one request at a time. The HDF5 API is defined in C, C++ and Fortran90.

NetCDF (10) is another file format and API that is widely used in the scientific community. It provides interfaces in C, C++, Fortran77 and Fortran90. NetCDF also provides a way of subsampling data arrays (simpler than HDF5), and only specifies a synchronous API. While the original NetCDF API was designed for serial data access, it was enhanced to parallel NetCDF (24), based on MPIIO and having similar features with MPIIO described above.

OpenDAP provides a file-format independent view on data files (11). The data selection mechanism is defined using a special syntax extending the HTTP URL to a DAP URL. Data transport is handled by HTTP using HTTP attachments. DAP also specifies a subsampling mechanism, similar to NetCDF's subsampling mechanism.

5. Remote Data Interface

In order to satisfy the requirements described in Section 3, we propose a client/server architecture that separates the data pipe in a data selection/control channel and a binary data transport channel[1]. We will present the operations of the control channel in detail while leaving the exact specification of the binary channel protocol open. Nevertheless, we will provide a set of requirements that need to be fulfilled by an implementation of the binary channel in order to be compliant with our architecture. For the control channel communication we choose SOAP/XML. We use gSOAP (25) to automatically generate stubs for C/C++ from a simple, C-like declaration language.

5.1. Establishing a Virtual Pipe

A visualization session starts with the client contacting the data server (or active simulation) and negotiating the establishment of a (virtual) pipe.

5.1.1. Selecting a Data Set. The negotiation includes selection of a data set. The selection mechanism is dependent on the way data is stored in a file, or on the way the data sets are referenced to in a running simulation. The details of identifying the desired data set is outside the scope of this work. We propose a string with an application specific structure as the identifier of the data set.

5.1.2. Session Id. Once a data set is selected, the server generates an identifier that is used by the client in subsequent calls to identify this session.

5.1.3. Selecting the operating mode for the pipe. This SOAP call includes information about the way the client wants to operate the complete data pipe. We support three types of ordering: ordered, semi-ordered, and unordered; and two types of delivery: reliable and unreliable. For example, the combination of unreliable and unordered means that the data pipe can be operated in an unordered mode. This can be independent of the semantic of the binary channel, for example one can use a TCP connection to operate an unreliable pipe but it might also mean that specific types of binary channels are incompatible with the operating mode of the pipe. Is the client job to make sure those types of channels are not used in the next step.

5.1.4. Negotiating a Binary Channel. In this step, the client sends a list of supported types of binary channels to the server (encoded as a vector of strings). The order, in which the types of binary channels are listed represents the priority list defined by the client. The server will compare the received list with its list of supported binary channel types and will select the first common type. The server will then create an endpoint that will be encoded and sent to the client together with the binary channel type in the SOAP response. The endpoint could be encoded as a URI. The client establishes the binary pipe by connecting to the received endpoint.

For example, the endpoint information of an ordered, reliable, TCP-based pipe will be represented by a hostname followed by a port number, like the following: "litchi.zib.de:10412". With the appearance of grid APIs for interprocess communication, we believe that a standardized way of exchanging and encoding endpoint information will be shortly available.

5.2. Selecting and Reading Data

The most important operation in our API is the data read and selection operation. Our data selections are described similar to a hyperslab but is adapted to visualization needs. A d-dimensional data set's size s_i is quantified for each dimension i at the highest resolution. Our selection is specified by a starting index p_i, a stride Δp_i and the number of selected values n_i. This selects $\prod_i n_i$ values at the positions $\{(p_0 + k_0 \cdot \Delta p_0, \ldots, p_{d-1} + k_{d-1} \cdot \Delta p_{d-1}) | k_i = 0 \ldots n_i, i = 0 \ldots d-1\}$. The stride Δp_i not only describes the position of the data values but also the targeted resolution.

[1] We refer to the control channel and binary channel combination as the data pipe

The data source should be able to deliver a filtered version of the data set suitable for display at this resolution. This could be achieved by averaging cubes with a size of 2^l voxels in each direction and storing the averaged values. This data source would deliver the same value for each voxel $p_i \ldots p_i + \Delta p_i - 1$ in each direction. Advanced filters might be used for higher quality low resolution versions. Subsampling could be used as an approximation—but would cause aliasing artifacts. Details of the available filtering mechanisms might be negotiated during session startup. The described selection mechanism can be directly used to implement orthogonal slicing, hierarchical volume rendering, and retrieval of subvolumes at user specified resolution.

Data read operations are executed in two stages. In the first stage, the requests are sent to the server and acknowledged by the server. The requests are stored queued and served through the binary channel in a way consistent with the data pipe operating mode. The requests for selected data (block requests) are associated with a block request id. As the data is pushed trough the binary channel, the block id is transferred together with the block data. Thus, the binary channel must deliver complete data blocks together with their block ids.

5.2.1. Combining Read Operations. When using a partially ordered data pipe, as useful for example when doing multi-resolution orthoslicing where the data for the slice in one resolution must be received completely before the data for the higher-resolution slice, the client needs to combine the block requests that belong together. We combine the block requests in vectors of blocks named transactions. When submitting a vector of transactions trough the control channel, the transactions are explicitly ordered in the order they appear in the vector. Our data pipe will send served block requests from the second transaction to the application only after all the block requests from the first transaction were sent and so on. Through this operation, the server will gain knowledge of the ordering information and will use the binary channel accordingly.

The minimal requirement placed on a semi-ordered binary channel to support the implementation of this operation mode is for the binary channel to accept a parameter that specifies the maximum number of out-of-order data blocks that the binary channel should guarantee. If the maximum number is k that means that two blocks b_i and b_j that are separated by n blocks when inserted in the binary channel will be separated by at most $n+k$ blocks when exiting the binary channel.

5.3. Canceling

The client can send a cancel operation trough the control channel using the session id. This operation will cancel all the requests that are still pending on the server queue. Before responding to the request, the server must stop sending blocks trough the binary channel. On the client side, after receiving the answer from the server, the binary channel will be canceled. This means that the binary channel must not deliver any more blocks until a new block is inserted on the server end of the channel. Implementation of this requirement is specific to the particular binary channels.

5.4. Summary of the Binary Channel

Summarizing the requirements to the binary channel from above we get the following list:

- Operating mode: the pipe receives and delivers data blocks and block ids (write (block, id), read(block, id)). Simplex binary channel : only the server writes and only the client reads.
- Asynchronous read: at least on the client side, the channel should support asynchronous operations and notifications. The application registers for notification and as soon as a data block is ready to be delivered the binary channel notifies the application per callback or using other mechanisms
- Canceling: on both sides, all the pending operations can be canceled. After cancel is called on both the server and client side no more blocks are delivered until new blocks are inserted on the server side and all the blocks pending in the channel will be thrown away
- Semi-ordered channel: if a binary channel supports the semi-ordered operating mode then it must have a parameter that controls the maximum number of blocks that can be delivered out of order

6. Simple Prototype

Our first implementation of the data pipe uses a raw TCP socket for the binary channel. This socket is established in an initial negotiation between client and server. The client requests selections from the server using the SOAP control channel. HTTP keep-alive is used to establish persistent connection for the control channel. The implementation is based on gSOAP (25), which provides basic SOAP handling and function stubs that need to be filled to complete the server. Requests are queued and ids returned to the client.

The queued requests are pushed to the client through a TCP socket in a background thread. The socket is implemented using the BSD socket interface. We simply send the id followed by all binary data. The client receives the

data in a background thread, queues them and notifies the main application thread. We use non-blocking I/O and stop sending further data if the SOAP server thread received a cancel message.

For testing purposes, we implemented a server that computes data from a simple analytical expression. In real-world applications a server could be a running simulation. To evaluate network performance, we switched computation completely off and sent uninitialized data blocks to avoid to be limited by disk I/O or CPU. Measurements showed that our API is able to efficiently utilize the binary channel. We were able to fully consume the bandwidth of a 100Mb network and about 70% of the bandwidth of a Gigabit network, which is realistic for an untuned TCP connection.

7. Summary and Future Work

We proposed a remote data access and filtering schema based on the separation of control and binary channels in order to support a number of use cases of interactive visualization of three dimensional data organized on a structured grid. Our control channel API which includes the data selection API is implemented using the language-independent SOAP protocol. The binary channel protocol is an abstraction of an interprocess communication protocol and gives the application the freedom to choose the wire protocol that is most suitable for implementing the needed functionality. Our initial tests show that this API does not limit the network performance, which was one of our most important design goals. Different implementations of the binary channel, partially backed by the appearance of standard grid APIs like the GAT or the SAGA API, will help us refine our own API. We target at finally providing a remote data access API suitable for a wide range of interactive visualization applications.

References

[1] K. W. Brodlie, D. A. Duce, J. R. Gallop, J. P. R. B. Waltonand, and J. D. Wood, "Distributed and collaborative visualization," *Comput. Graph. Forum*, vol. 23, no. 2, pp. 223–251, 2004.

[2] J. Shalf and E. W. Bethel, "The grid and future visualization system architectures." *IEEE Computer Graphics and Applications*, vol. 23, no. 2, pp. 6–9, 2003.

[3] K. Brodlie, D. Duce, J. Gallop, M. Sagar, J. Walton, and J. Wood, "Visualization in grid computing environments," in *Proc. IEEE Visualization '04*. IEEE Computer Society, 2004, pp. 155–162.

[4] E. J. Luke and C. D. Hansen, "Semotus visum: a flexible remote visualization framework," in *Proc. IEEE Visualization '02*. IEEE Computer Society, 2002, pp. 61–68.

[5] W. Bethel and J. Shalf, "Consuming network bandwidth with visapult," in *The Visualization Handbook*, C. D. Hansen and C. R. Johnson, Eds. Elsevier, 2005, ch. 29, pp. 569–589.

[6] S. Prohaska, A. Hutanu, R. Kähler, and H.-C. Hege, "Interactive exploration of large remote micro-CT scans," in *Proc. IEEE Visualization '04*. IEEE Computer Society, 2004, pp. 345–352.

[7] C. T. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom, "Out-of-core algorithms for scientific visualization and computer graphics," Visualization '02, Course notes, 2002.

[8] O. Kreylos, A. M. Tesdall, B. Hamann, J. K. Hunter, and K. I. Joy, "Interactive visualization and steering of CFD simulations," in *VISSYM '02: Proc. Symposium on Data Visualisation 2002*. Eurographics Association, 2002, pp. 25–34.

[9] NCSA, "HDF5 - A new generation of HDF," 2003. [Online]. Available: http://hdf.ncsa.uiuc.edu/HDF5/

[10] UNIDATA, "NetCDF—network common data format," 2004. [Online]. Available: http://my.unidata.ucar.edu/content/software/netcdf

[11] J. Gallagher, N. Potter, T. Sgouros, S. Hankin, and G. Flierl, "The data access protocol—DAP 2.0," 2004. [Online]. Available: http://opendap.org/

[12] D. Bergeron, "Visualization reference model (panel session position statement)," in *Proc. IEEE Visualization '93*, 1993, pp. 337–342.

[13] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," *Proc. CASCON'98, Toronto, Canada*, 1998.

[14] M. Forum, "MPI-2: Extensions to the message-passing interface," 1997. [Online]. Available: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[15] T. Eickermann, W. Frings, and A. Häming, "Visit – a visualization interface toolkit. version 2.0 manual," 2002. [Online]. Available: http://www.fz-juelich.de/zam/visit/visit20beta/manual

[16] T. Kurc, Ü. Çatalyürek, C. Chang, A. Sussman, and J. Saltz, "Visualization of large data sets with the active data repository," *IEEE Comput. Graph. Appl.*, vol. 21, no. 4, pp. 24–33, 2001.

[17] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, "DataCutter: Middleware for filtering very large scientific datasets on archival storage systems," in *Proc. Mass Storage Systems*. IEEE Computer Society Press, Mar. 2000, pp. 119–133.

[18] W. Allcock, J. Bester, J. Bresnahan, S. Meder, P. Plaszczak, and S. Tuecke, "GridFTP: Protocol extensions to FTP for the Grid," *GWD-R (Recommendation)*, Apr. 2003.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masiner, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," 1999. [Online]. Available: ftp://ftp.isi.edu/in-notes/rfc2616.txt

[20] "SOAP specifications." [Online]. Available: http://www.w3.org/TR/soap/

[21] J. Barton, S. Thatte, and H. F. Nielsen, "SOAP messages with attachments," 2000. [Online]. Available: http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211

[22] H. F. Nielsen, H. Sanders, R. Butek, and S. Nash, "Direct internet message encapsulation (DIME)," 2002. [Online]. Available: http://msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-02.txt

[23] H. F. Nielsen, E. Christensen, and J. Farrell, "Specification: WS-attachments," 2002. [Online]. Available: http://www-106.ibm.com/developerworks/webservices/library/ws-attach.html

[24] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A scientific high-performance I/O interface," in *Proc. Supercomputing Conference*, Phoenix, Arizona, Nov. 2003.

[25] R. van Engelen and K. Gallivan, "The gSOAP toolkit for web services and peer-to-peer computing networks," in *CCGRID*, 2002, pp. 128–135.