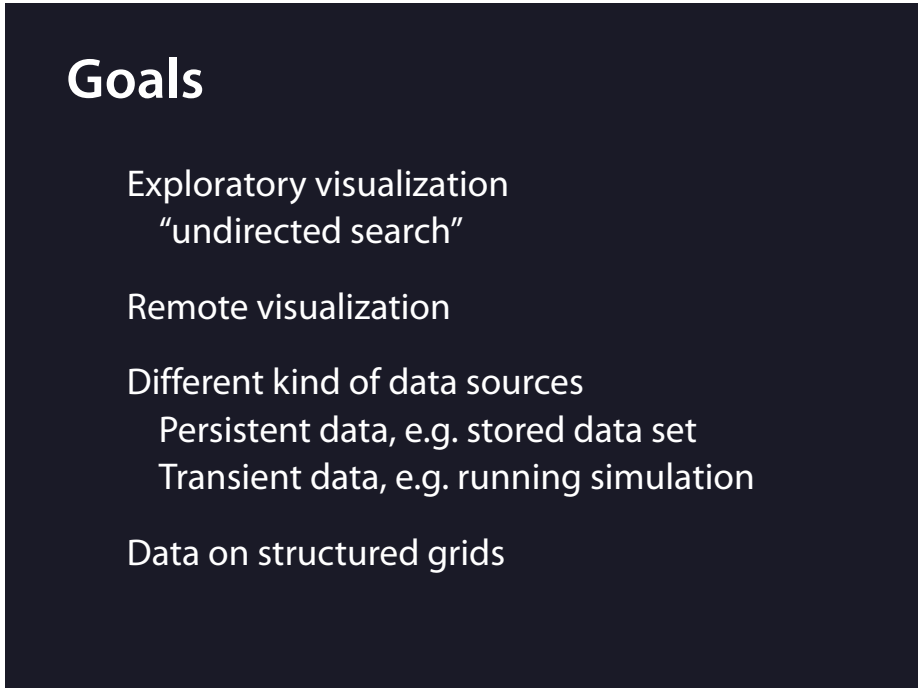
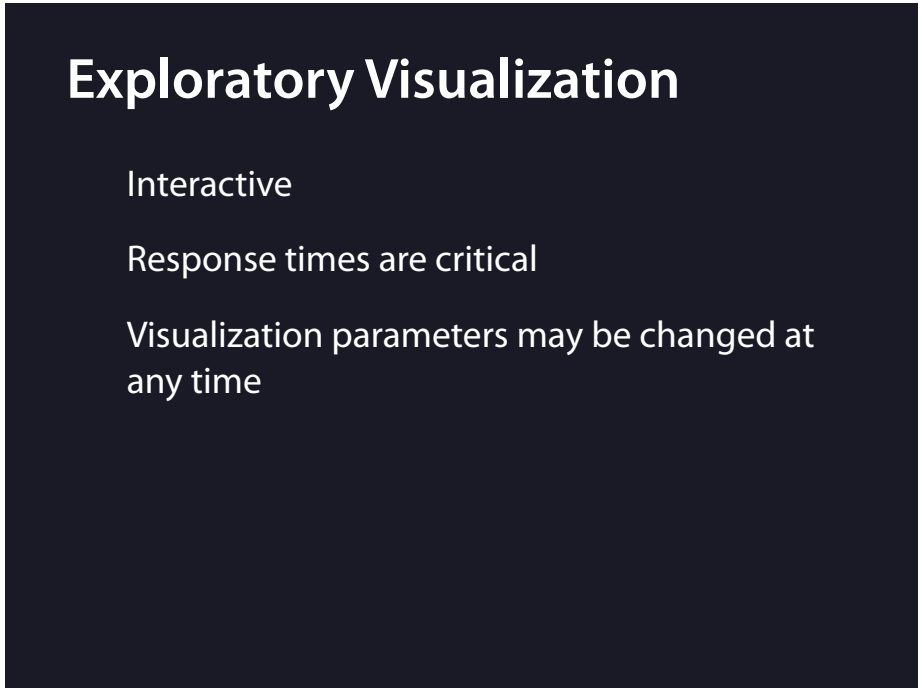
In the contribution to the conference, we describe some requirements for a remote data access interface suitable for interactive visualization—and we developed a prototype.

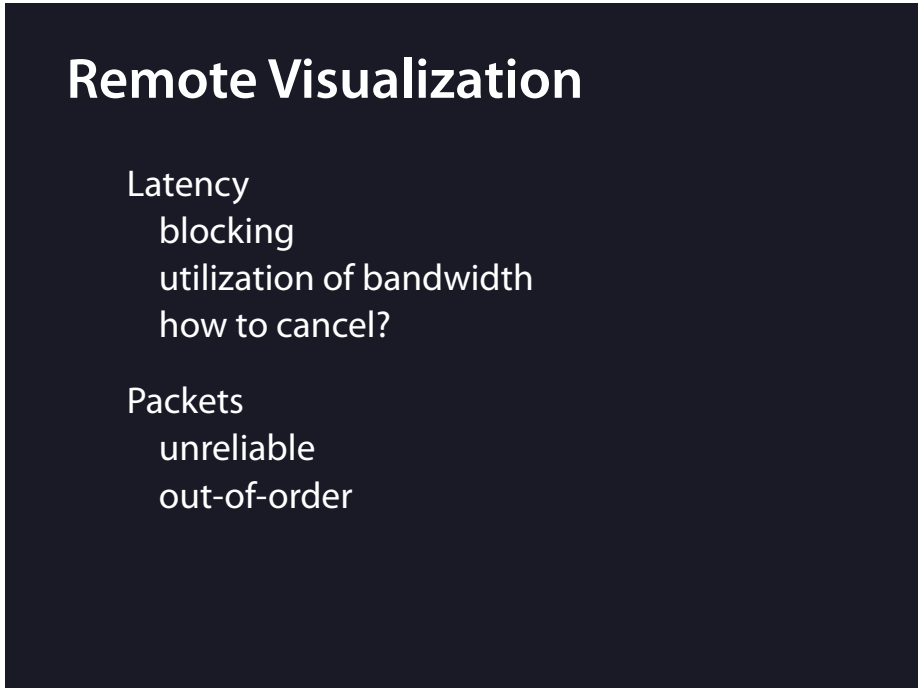
I’ll start describing the goals of this work.



The targeted application area is 'Exploratory Visualization'. No prior knowledge of the data is assumed. The task is to explore data to understand what's in it. Interactive tools, which let the user conduct this undirected search are required. The data shall be located remotely and accessed through a network.
This is useful for different kinds of data. Data sets of persistent nature might be stored centrally in data repositories. Running simulations might also be another source. The transient nature of these should be taken into account.
In our work, we only deal with structured grids. Simply spoken, data is organized as multidimensional arrays. But we believe that the general framework can also be used for other data types.



‘Exploratory Visualization’ requires interactivity.
A user should be able to control the application at any time. This requires some guarantees on response times. Interaction at any time also requires the application to deal with changing parameters at any time. Previously requested data might become unnecessary.



A network introduces specific problems that make it hard to build such an application.
The network introduces latency, which can differ in orders of magnitude. If no care is taken, this might lead to blocking behavior and insufficient utilization of network bandwidth. How to cancel previous requests might also be an issue.
A packet network might deliver data unreliably or out-of-order. Hiding this nature by abstraction layers is useful but might influence efficient utilization of the network.

Data Sources

- Persistent data
  - stored in data repository
  - preprocessing step possible
- Transient data
  - limited buffer size
  - unreliable
  - e.g. running simulation, sensors

Our data sources may be of persistent or transient nature.
Persistent data can be preprocessed to generate for example a multi resolution representation. They may be repeatedly and reliably accessed.
Transient data sources on the other hand side, might be available only at a certain point in time. They might deliver partial results caused by limited buffer sizes or other restriction in resources. It might for example be unwanted to stop a running simulation completely until a remote data request is served.

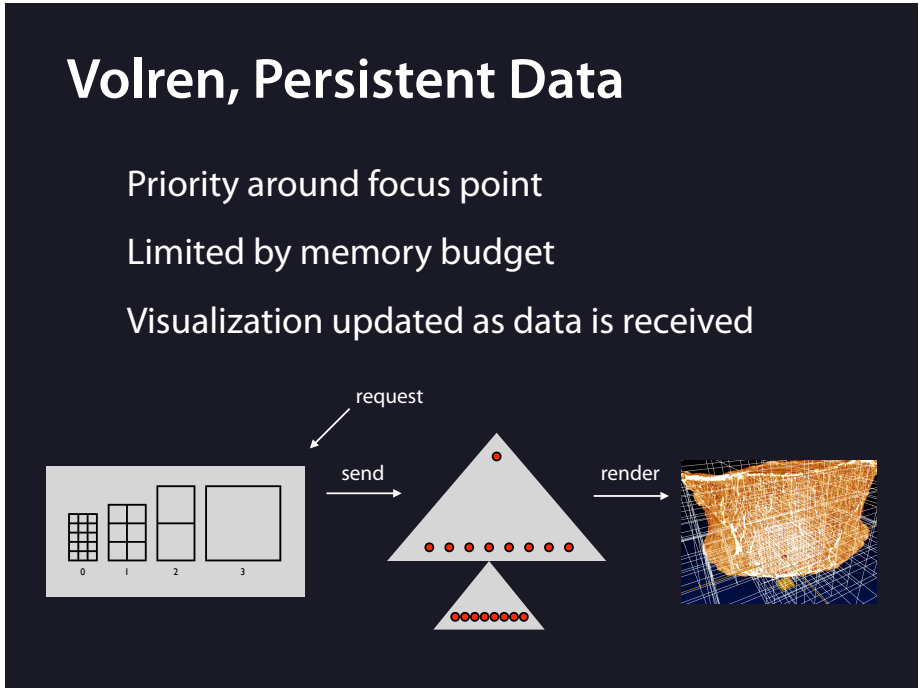
Use Cases

The next few slides will present some visualization use cases.

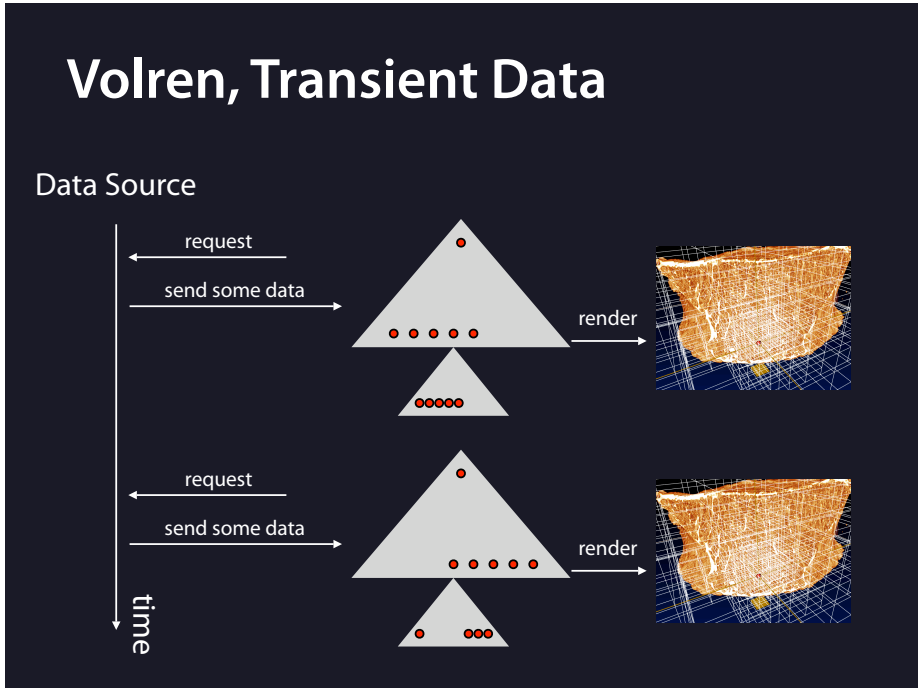
Use Cases

- Hardware based volume rendering
- Orthogonal slicing
- Retrieval of sub-blocks

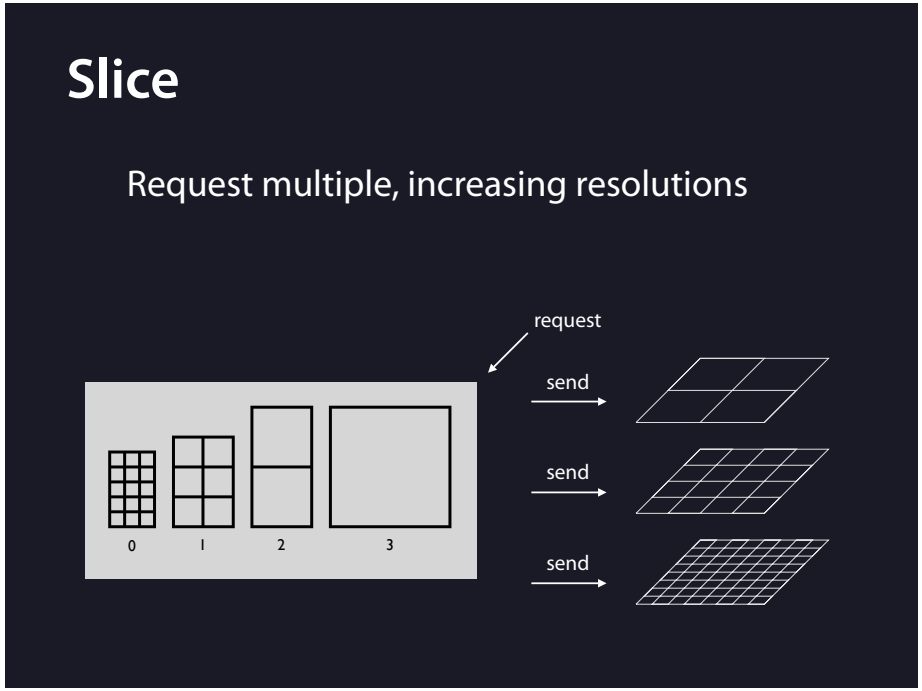
I’m going to cover volume rendering, slicing and simple retrieval of sub volumes.



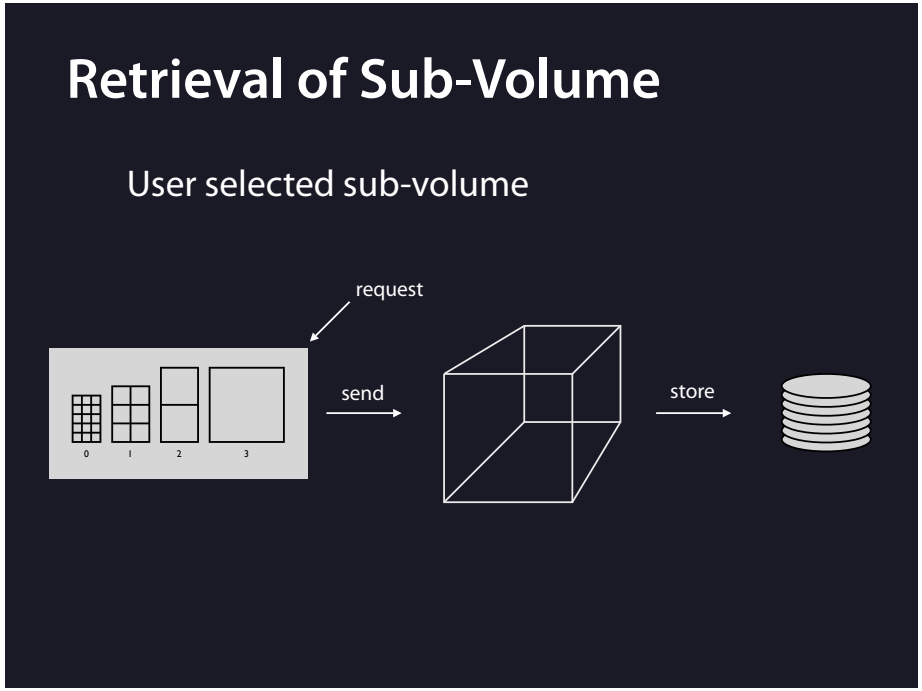
We assume a hierarchical volume renderer based on graphics hardware.
Data are requested in blocks at multiple resolutions with a higher priority around a focus point. An octree is built up until a memory budged is used. As new data becomes available, the visualization is updated.
A data set should be stored at several resolutions to allow efficient delivery of the requested data. This can be easily achieved for persistent data in a preprocessing step.



A transient data source might deliver only partial results.
And each request might deliver different results, if the source is changing over time.
Nonetheless, displaying such data can be useful. For example the evolution of a running simulation might be monitored. That would allow to decide if it should be continued or be aborted.



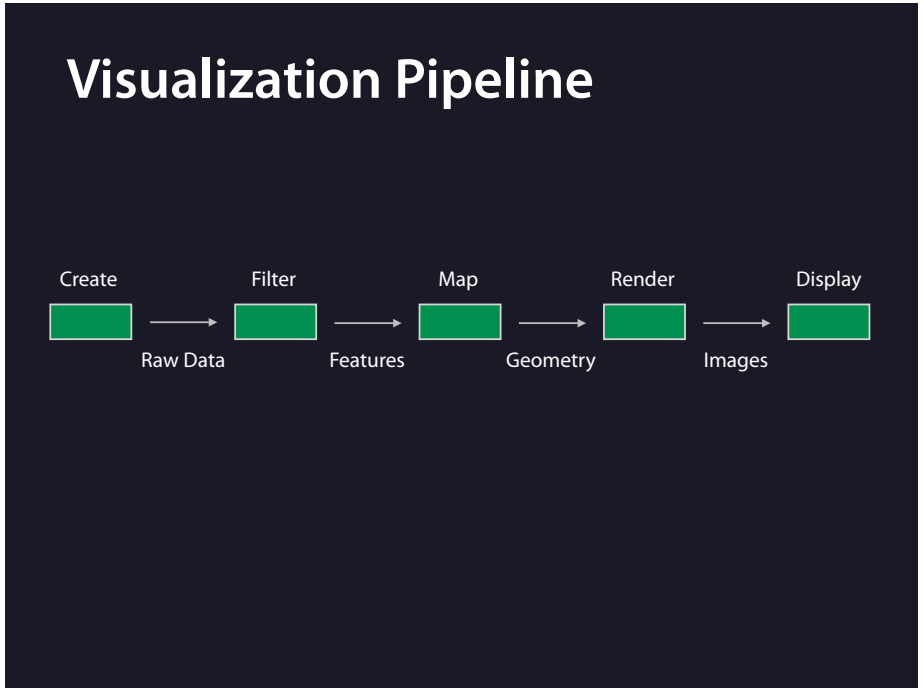
Slicing is similar in nature, it only requires a different kind of selection in the data. The request consists of a certain slice that should be delivered at increasingly higher resolutions.



Retrieval of user specified sub volumes is the last scenario.
The user selects the volume and resolution she's interested in. This part of the data will be sent and stored to a local disk for further processing.
This concludes the section about use cases.

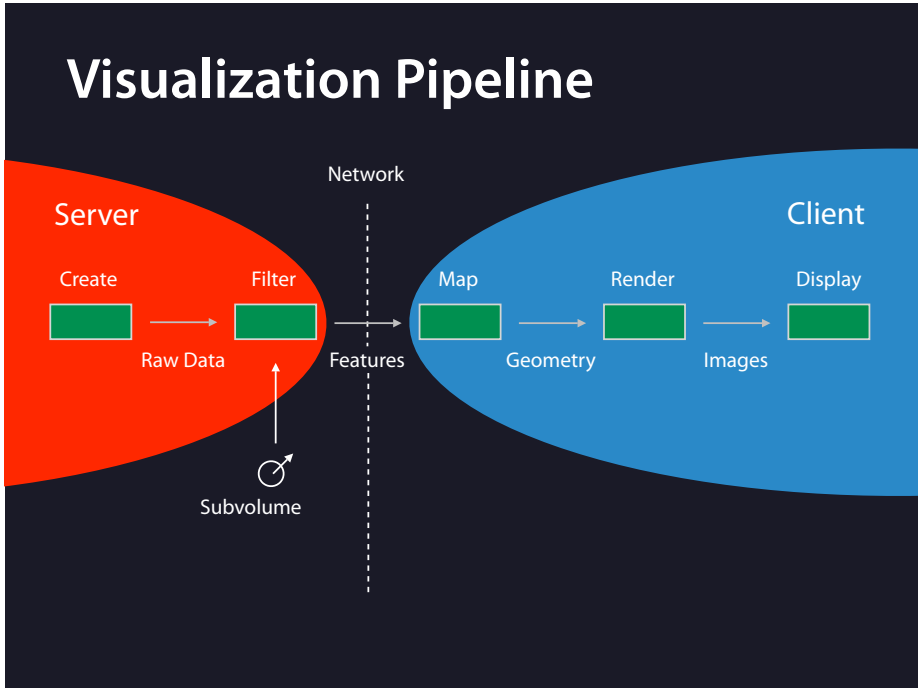
I'll move on to our solution.

We split the problem into two major tasks.
The first task is to pass high level descriptions from the client side to the server. In our case, these are hyperslabs, describing the selection in the data. I'll present more details shortly. For other visualization techniques, the description needs to be replaced. We use SOAP to pass requests, because it is a standardized remote procedure call interface and is rather simple to implement.
The second task is to deliver data back to the client. This needs to be done efficiently. We propose to use a binary pipe that must be able to deliver data blocks tagged with an id. It should also provide an efficient mechanism to cancel operations, flush previously requested data and restart with new requests. Different implementations can be useful. Today, I'll present a prototype based on a TCP socket.

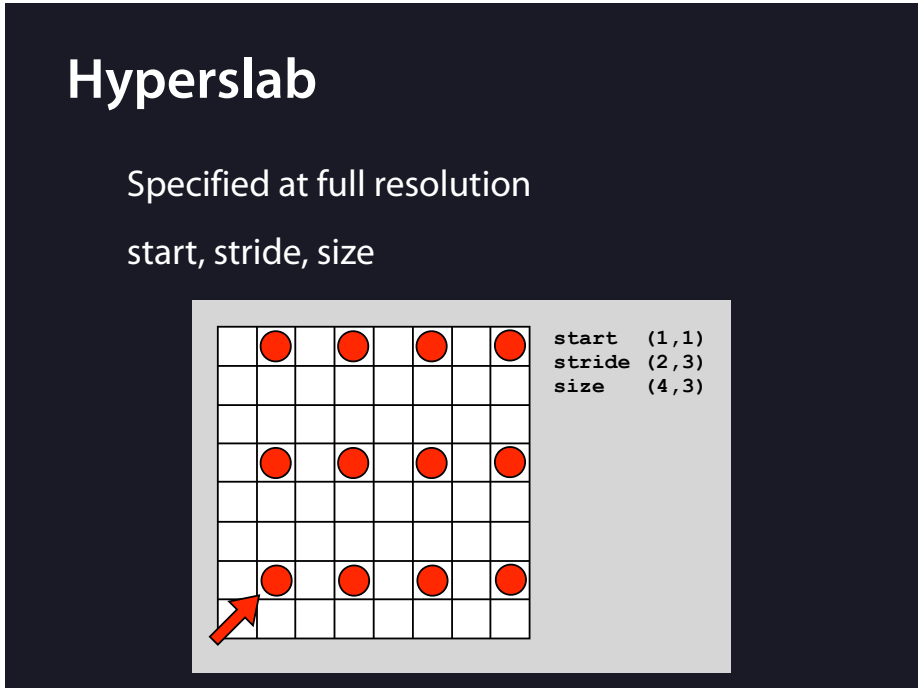


Visualization can be understood as a pipeline of tasks to generate the final image.

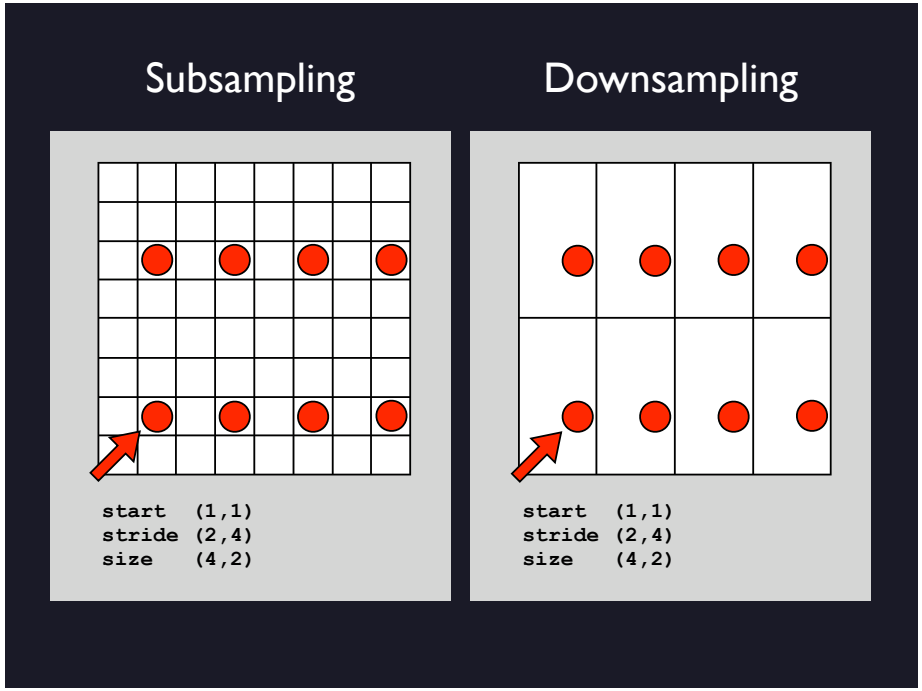
The ‘Visualization Pipeline’ describes the process of generating visual representations of data. From left to right:
data is created and passed as raw data to a stage, which filters it to extract features. These are then mapped to some geometry or texture, which is rendered to generate an image displayed on a screen.



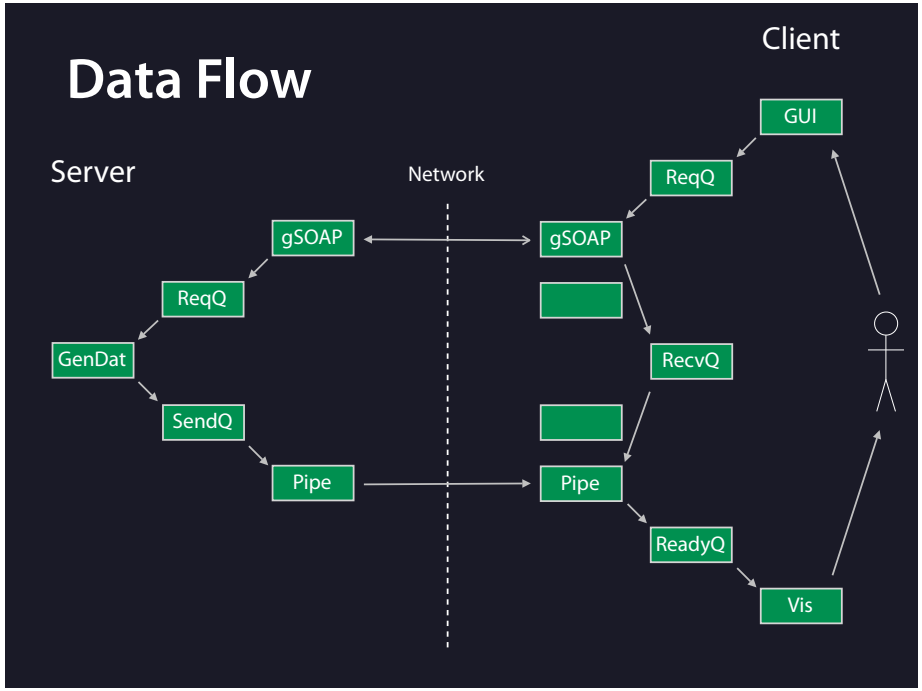
We split the pipeline as displayed in this slide. The server creates a filtered version of the data. These ‘Features’ are passed to the client which utilizes local hardware to generate a geometry and render an image.
The subvolume and resolution need to be specified to the filter stage.



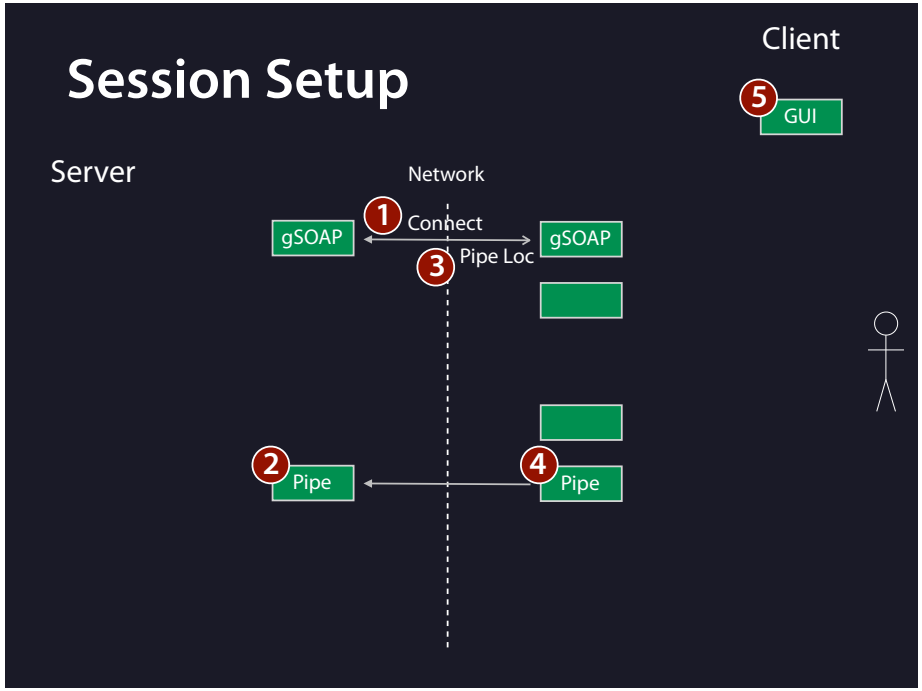
We looked through a variety of data access interfaces. For a detailed list I would like to refere you to our paper.
A common concept when dealing with multidimensional arrays is the hyperslab. A hyperslab describes a regular selection in a multidimensional array by specifying a starting point, the number of elements to advance to reach the next selected element —denoted as ‘stride’—and the number of selected elements in each direction. We use this description to pass our requests. We always specify the hyperslab at the full resolution of the data set.



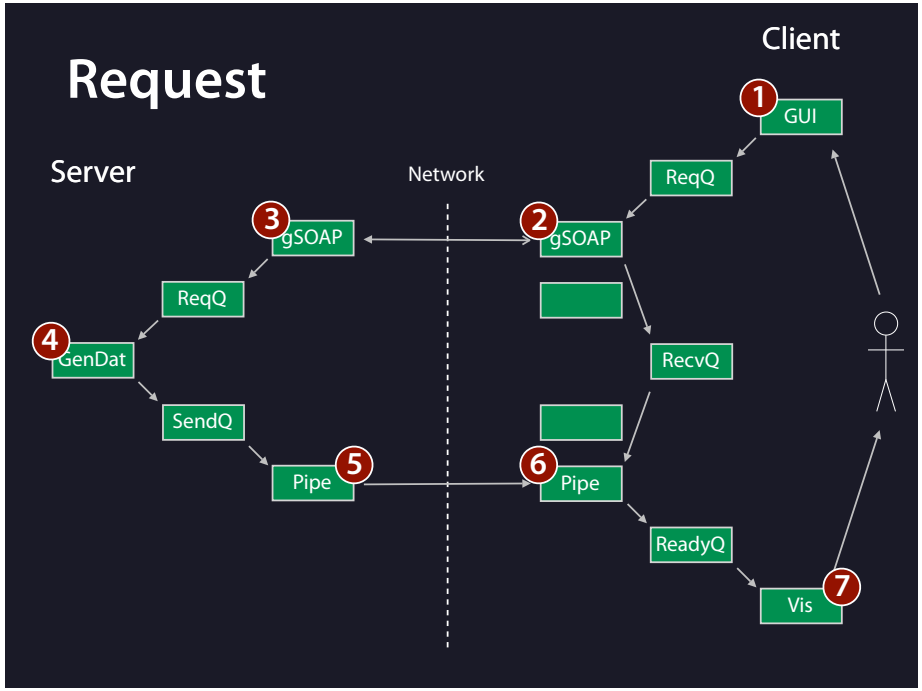
A data source might provide low resolution data by either using subsampling or downsampling. The description of the hyperslab stays the same. Downsampling should be implemented to avoid aliasing, but subsampling might be easier to achieve. Details of the available mechanisms can be passed during session setup. I'll skip them here.



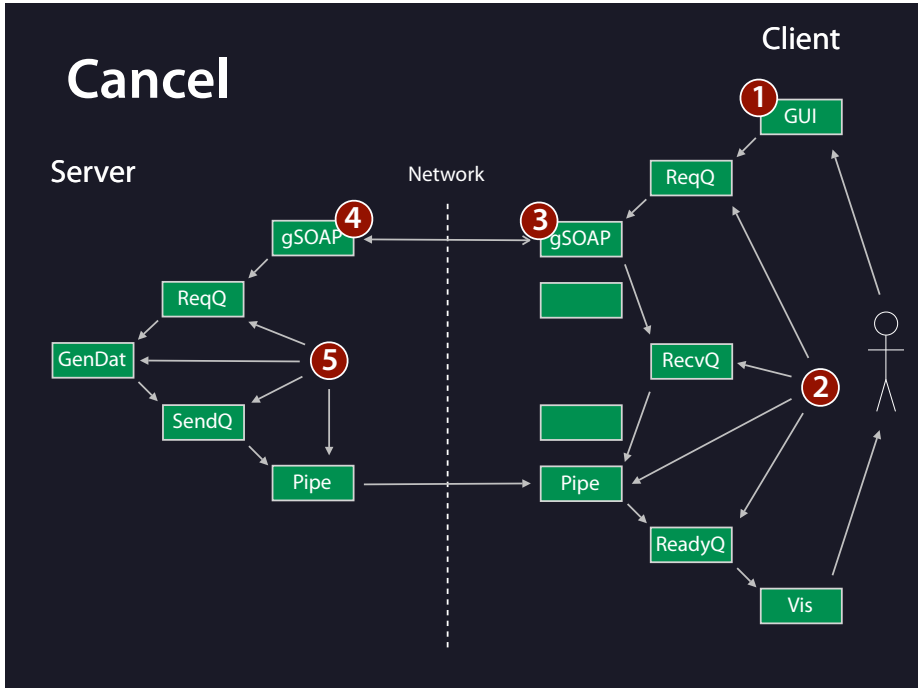
Such requests flow through our application until they are final passed to the visualization layer. The high level data flow is displayed in this slide.
I'll first describe session setup, followed by the handling of a standard request and a cancel operation. All the details of the diagram will be visited shortly.



The first step is session startup.
The client submits a connect request, at (1). The server creates the End Point of pipe a pipe, see (2), and passes the required information to connect to the endpoint back to the client, at (3). Information is passed using SOAP. Meta data about the data set are also passed to the client.
The client uses this information to connect the pipe, at (4), and present the data set to the user, see (5).



The user can now interact with the data set
Starting at (1), a user interaction creates a request that is placed in the Request Queue (ReqQ). An id is attached to each request. At (2) the request is sent to the server as a SOAP request. At the same time, it is placed in the Receive Queue—at the middle right—containing pending requests. At (3), the server receives the request and places it in Request Queue. Data is generated at (4), attached to the request, which is then placed in the Send Queue.
At (5), the binary pipe sends the data to the client. The client receives the data block at (6), compares it with pending request in the Receive Queue and places it into the Ready Queue (ReadyQ). A notification is send to the application about newly available data and the Visualization is updated at (7).
Multiple requests might be generated by a user interaction. They may be send as an array of requests to the server.



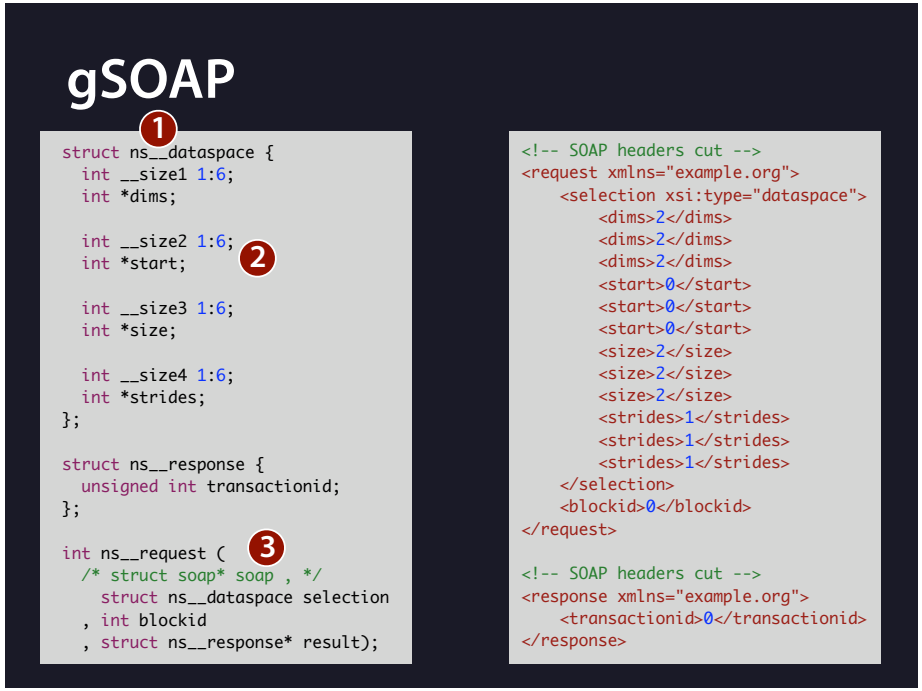
After a user interaction, already requested data might become unnecessary and need to be canceled.
This process is displayed here. At (1), a user interaction triggers cancel; all local queues and the pipe is canceled at (2). At (3), the cancel request is sent to server and received at (4). The server executes the request at (5).
Note that new request may be already queued, before the cancel request is completely processed.

I’ll now present in more depth, how we implemented the SOAP layer and our prototype of a binary pipe. We use gSOAP.

gSOAP

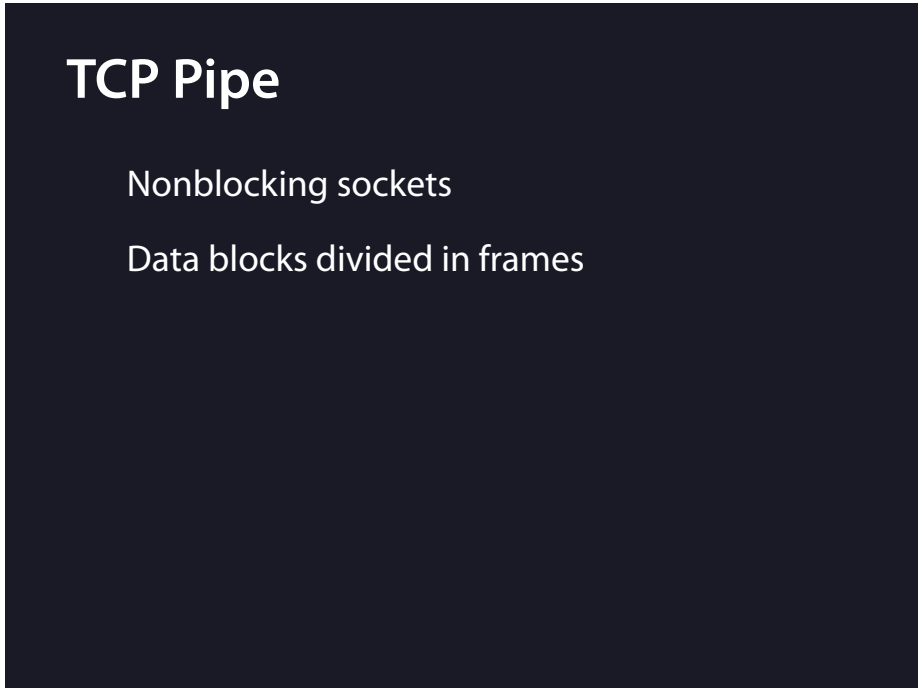
- Light weight SOAP server
- C-like declarations
- Preprocessor generates C-function decls and efficient SOAP parser (and XML Schema)
- Streaming, DIME, keep-alive
- www.cs.fsu.edu/~engelen/soap.html

gSOAP provides a light weight SOAP server, which is automatically generated from C-like declarations. A preprocessor generates an efficient SOAP parser and C-function declarations that need to be implemented.

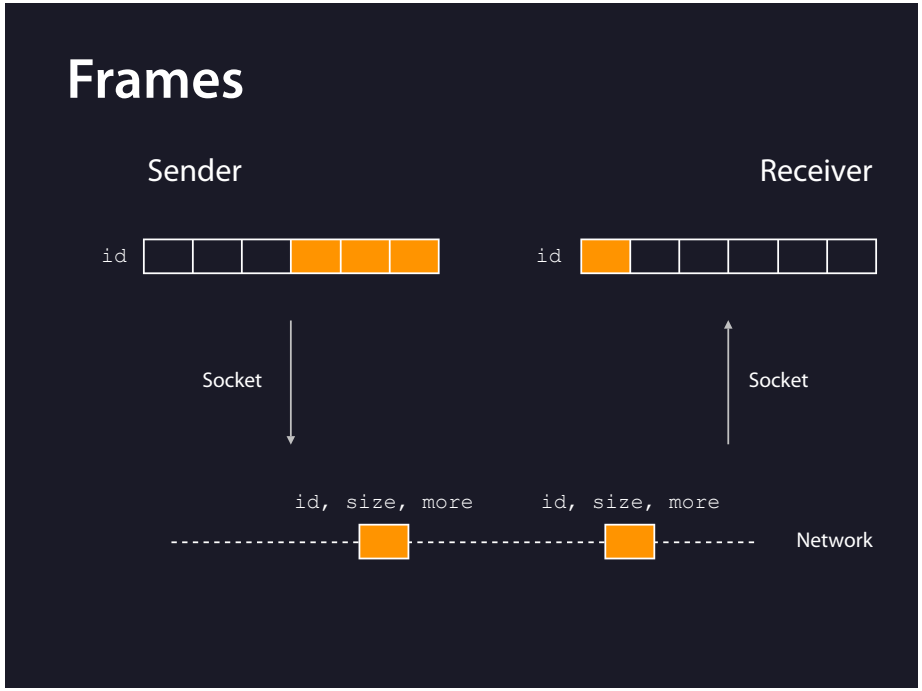


One small example to give a sense of this process. On the left side, you can see the declaration of a SOAP call.
gSOAP handles namespaces by a simple naming convention. Two underscores in a C symbol are used to indicated a namespace, see (1). At (2), a dynamic array with a restricted size between 1 and 6 is declared. gSOAP handles all the memory allocation and bounds checking. A function for each call needs to be implemented. It is declared at (3). gSOAP adds a pointer to an internal data structure.
On the right hand side, part of the mapping to a SOAP request is shown. All these details are handled by gSOAP.

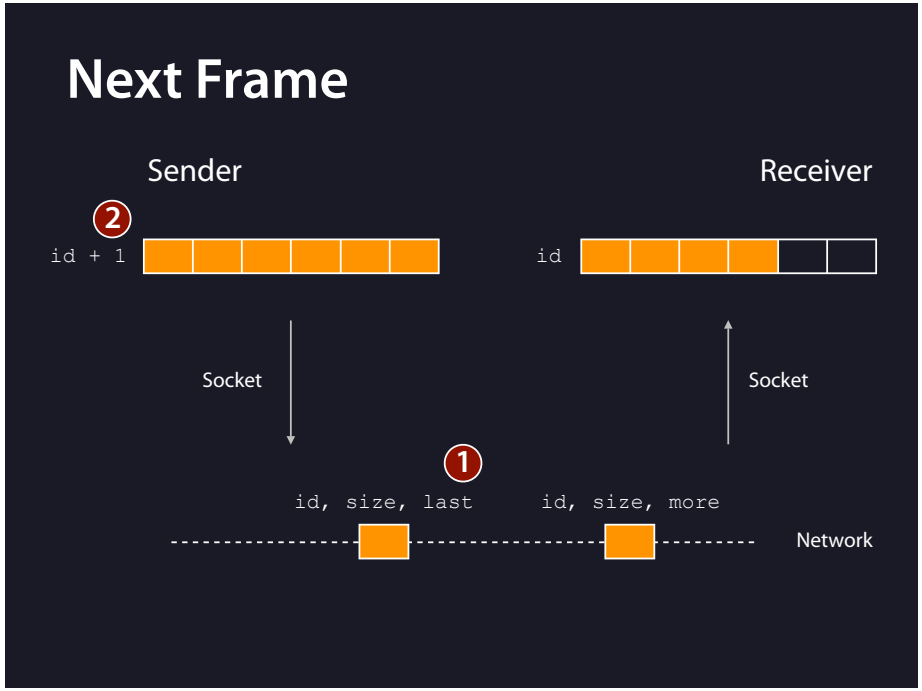
I’ll now move on to our binary pipe prototype.



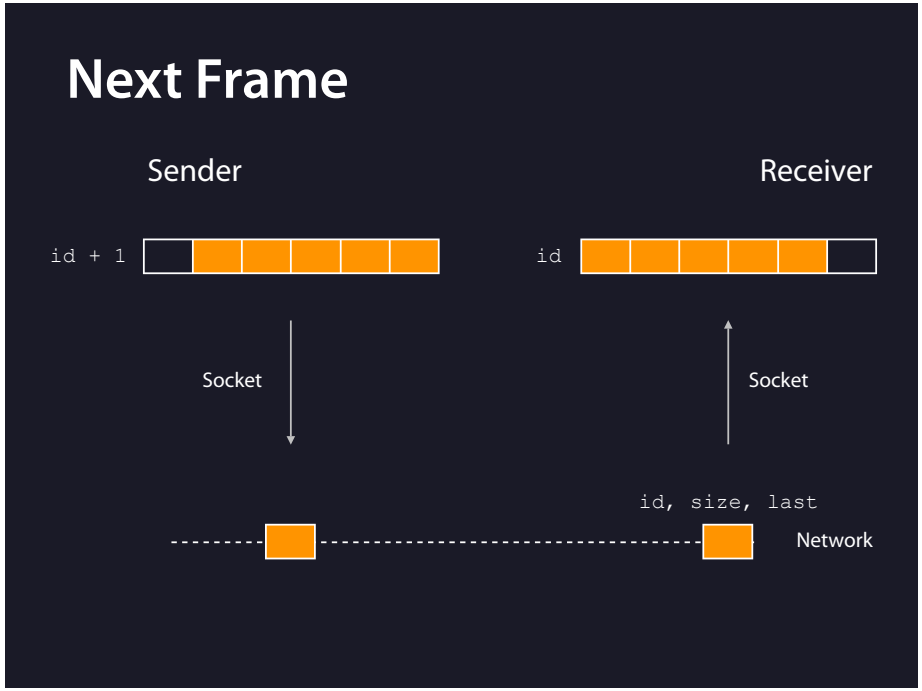
It is based on TCP and we use nonblocking sockets to implement it.
Each data block is sent in multiple parts, as I will show now.



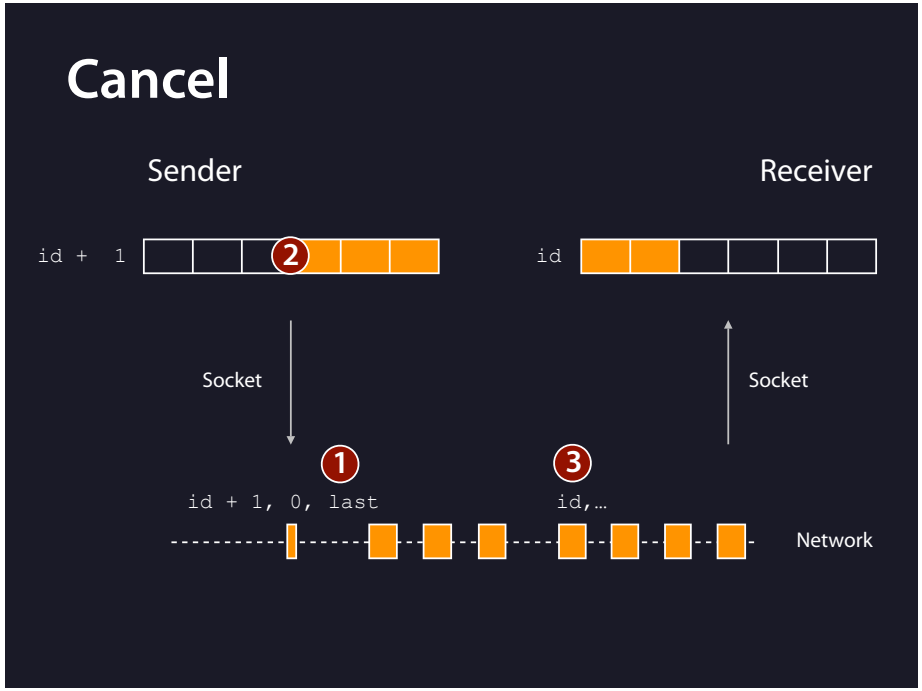
At the sender, a block is split in frames. Each frame is tagged with the id of the block, the frame size and a flag indicating if more frames will follow. The sender sends frame by frame, the receiver fills a buffer as frames are received.



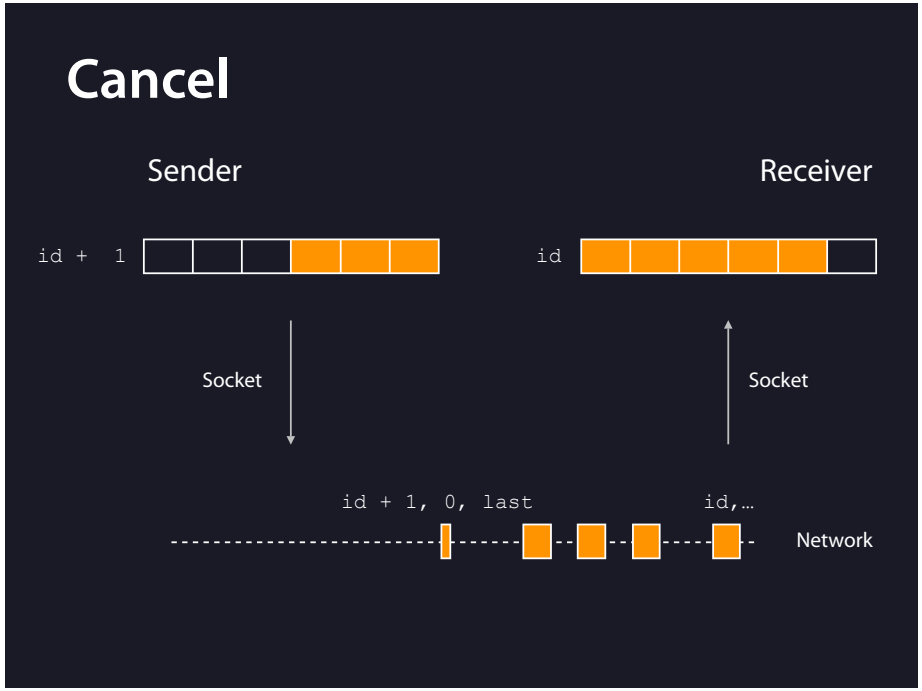
At some later point in time, the last frame is sent. It is tagged with a last flag, see (1). The next block with id + 1 is brought into the sent buffer, at (2)



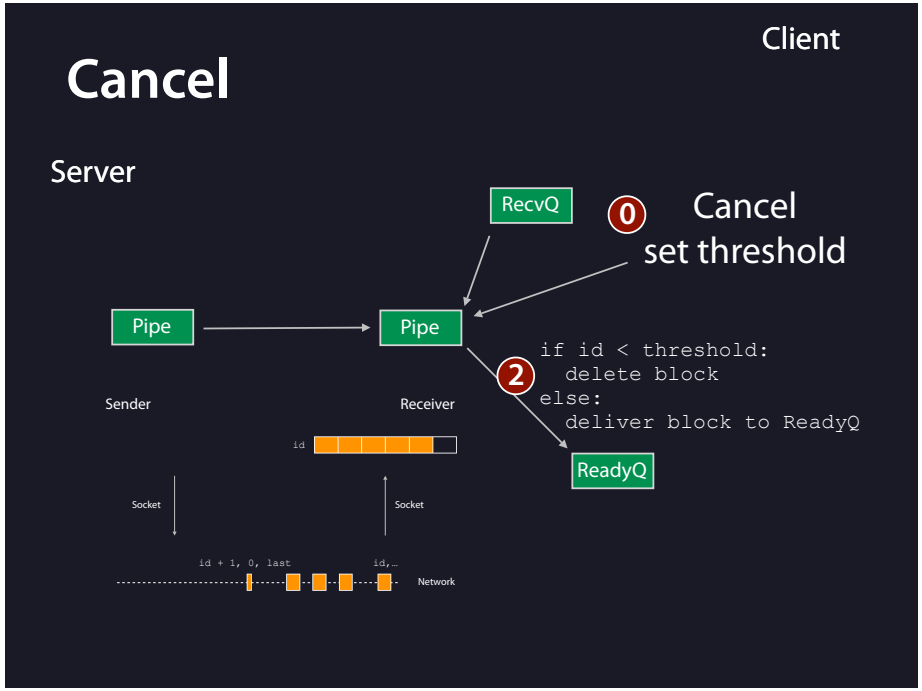
This last frame is received on the client and delivered to the visualization as discussed before.



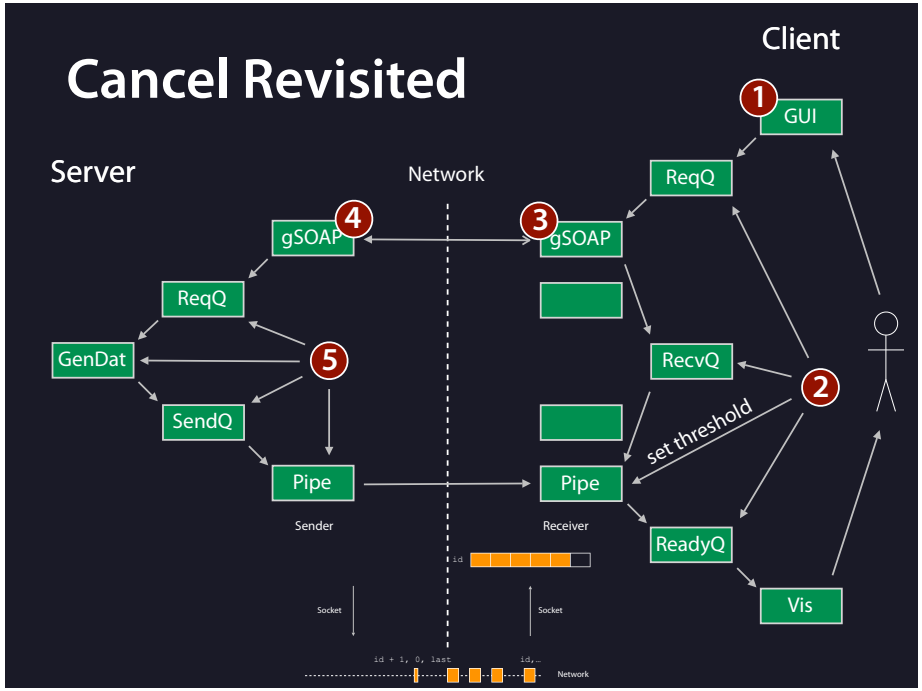
I continue the discussion explaining how to cancel operations on the pipe. I'll use a more complicated situation now. Here, one block is not yet completely received while the next block is already partially sent.
A cancel request is handled the following way. Instead of sending more frames, an empty frame, with the last flag set, is sent, see (1), and the sender buffer is discarded, at (2). The empty frame, with the last flag set, clearly indicates a canceled block. But there's another complete block on the network, at (3), which will be received earlier.



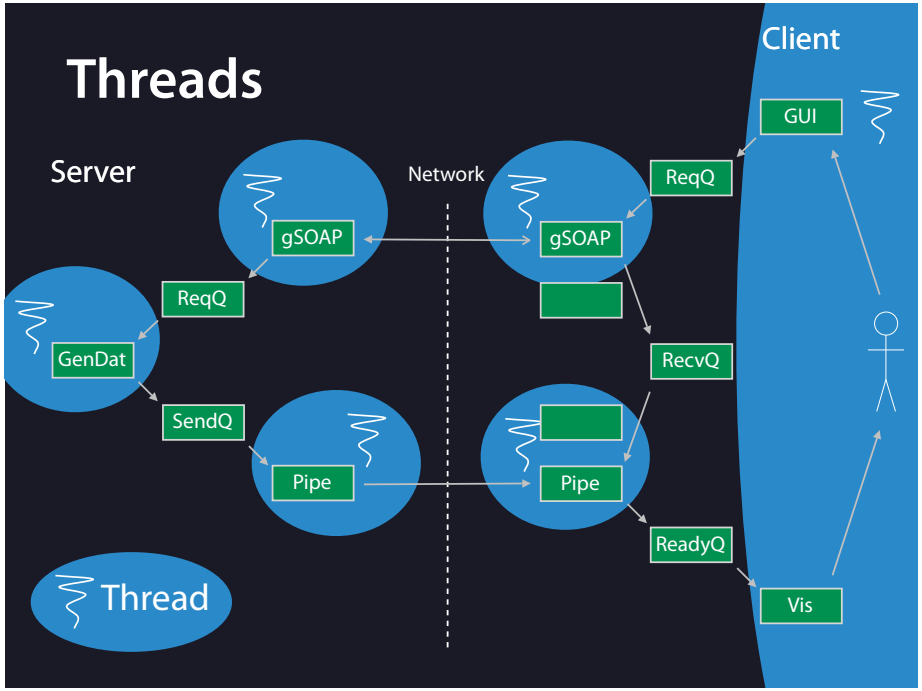
Now, the last frame of this block is received, but should be discarded on the client side.
This is achieved in the following way.



To understand this, I'll show the context in the overall data flow. The two pipe end points and two queues on the client side are displayed here.
Earlier during the cancel operation, at (0), the client set a threshold on the ids. Blocks below this threshold are silently deleted by the receive end point of the pipe. As all the block ids are issued by the client, it knows this threshold at any time.



Back to the diagram from above. It has become clear, how all the steps of such a cancel operation will be executed.
This concludes the discussion of the TCP pipe.



We implemented the different parts of the system in separated threads. The GUI and the visualization run in one thread. gSOAP and the pipe end point on server and client are separated threads, as well as the data generation on the server.

To wrap up.

Summary

XML SOAP to send high level requests

Binary pipe to transport requested data

Fully asynchronous architecture

Efficient way to cancel pending requests

Prototype achieved 70% utilization of GigE

We developed a system that uses SOAP to pass high level requests between client and server. The binary data associated with these requests are sent by a binary pipe. I discussed a prototype implementation based on a simple TCP socket, but other implementation might be plugged in. The architecture is fully asynchronous and is able to efficiently cancel operations.
With our prototype we were able to consume 70% of the bandwidth of GigE running the application in a local area network.

Future

- Various types of pipes
  - unordered
  - many-to-one
- Real world
  - Security
  - Integration into existing applications
  - e.g. Cactus?
- Other visualization scenarios

We plan to implement or use various kinds of pipes.
As we discussed in the paper, it might be more efficient to deliver blocks without fixed order. If might also be useful to connect a visualization client to more than one server machine.
We would like to integrate our work into real world applications. This could require to add security mechanism to session startup and data transfer.
Visualization scenarios for other data types and use cases are another point on our agenda.