ParaView uses Proxies to control vtkObjects. A Proxy's Properties control individual Methods on the Object.

The proxies give configuration independence to the application code. The same call to control a proxy works whether the Object lives inside the same process or on remote and possibly parallel processes, or both.

A vtkProcessModule enum determines where the object controlled by the proxy lives. The default is on the data server, but it can be on every process or on particular ones.
Proxy C++ class inheritance

Proxies are implemented in C++ classes (specifically `vtkSMProxy`). Many subclasses exist to refine behavior.

`vtkObject` C++ inheritance

The objects that proxies control also have C++ inheritance.

SubProxies

Proxies can contain SubProxies. The parent Proxy can share properties with its Subproxies, and thus one Proxy can control many `vtkObject`

Proxy configuration inheritance

Proxies are configured from the contents of XML files (`Servers/ServerManager/rendering.xml`).

The proxy name from the XML defines the specific C++ `vtkSMProxy` subclass that is instantiated when a given proxy is asked for.

The XML configurations have an inheritance relationship that is independent of the C++ class inheritance.

Configurations also can define containment relationships (SubProxies).

Run time configuration

The C++ and XML configuration determine what overall static structure of a proxy is, but the proxy has to be finalized at run time. Here, internal pipelines are constructed by calling mainly `vtkSMProxy::BeginCreateVTKObjects` and then `vtkSMProxy::CreatePipeline()`.
ParaView sets up data processing pipeline by instantiating SourceProxies. vtkSMSourceProxies are vtkSMProxies that are specialized to control vtkAlgorithms.

Connections between SourceProxies are managed with SMOutputPort proxies. These correlate to vtkAlgorithm::OutputPorts, each of which produces vtkDataObjects.

The SourceProxy pipeline graph mirrors the Algorithm pipeline graph, but it is not 1:1 because Proxies (via SubProxies) can control more than one Algorithm. This happens for example in a Clip filter which controls the a widget and the filter that clips onto the widget.
Representations manage formatting and delivery of data from the producing algorithm to the consuming rendering engine (ex, surface extract, mapper, actor etc). The algorithm and engine may live on different machines.

Representations often have sub representations, these can be swapped in and out (surface mode versus wireframe mode), or any number of them can be enabled simultaneously (surface mode and selection labelling.)

Every vtkDataObject produced along the pipeline can be displayed simultaneously. ParaView manages the rendering pipelines with Representations within Views. Both Views and Representations are Proxies.
Every output along pipeline may be displayed simultaneously in many different views.

Exact representation chosen depends on data type and view type. Ex, Spreadsheet view doesn't have mappers and actors in the representation, nor cameras and lights in the View.
Representations have internal Strategy proxies. Strategies give the display pipeline configuration independence. The Strategy chosen depends on data type and configuration.
Simple Strategy

Simple Strategy is instantiated for 3D rendering in builtin (serial) configuration.

The LOD pipeline in a strategy is active during camera motion. It does geometric downsampling to help maintain interactivity with large data. When the mouse button releases, the full-res subpipeline activates instead.
Parallel Strategies are more complex. They have UpdateSupressors, which give the client application a way to tell the remote pipelines to update. They are named suppressors because they prevent the pipeline from running off the upstream end on the client (and renderserver) for which the US will have no input. Parallel Strategies also have MPIMoveData filters which send data forward across processes. On Rendering servers only, distributors exist to swap data chunks between server nodes to enforce back to front ordering for volume rendering.
Simple Parallel Strategy
On Server

Geometry

Update Suppressor

MPI Move Data

Post Collect US

Distributor

Post Distributor US

Render Window

Renderer

Composite

Simple Parallel Strategy
On Client

Update Suppressor

MPI Move Data

Post Collect US

Post Distributor US

Render Window

Renderer

Mapper

NOTE: LOD pipelines omitted from diagram for clarity

Depending on settings and data size, rendering may happen on the server or on the client (or in tiled display, both).

ParallelRenderManager objects are attached to the RenderWindows and synchronize window, mouse, and render event calls on all of them.

Two ParallelRenderManagers are involved for ParaView, the first does depth compositing on the server, the second delivers the final result to the client.
SERVER

Process 0

RenderWindow

3D Renderer
"Data"

2D Renderer
"Annotation"

Composite Render Manager
binary swap after RW::Render()  

Client Server Deliver Render Manager After CRP::Composite()  

Client Server Deliver Render Manager After 3D Renderer Render()  

Process 1

RenderWindow

3D Renderer
"Data"

2D Renderer
"Annotation"

Composite Render Manager
binary swap after RW::Render()  

Process 2

RenderWindow

3D Renderer
"Data"

2D Renderer
"Annotation"

Composite Render Manager
binary swap after RW::Render()  

Process 3

RenderWindow

3D Renderer
"Data"

2D Renderer
"Annotation"

Composite Render Manager
binary swap after RW::Render()  

CLIENT

RenderWindow

3D Renderer
"Data"

2D Renderer
"Annotation"
DataServer/RenderServer mode inserts another between the data producer and the client.

NOTE: LOD pipelines omitted from diagram for clarity.