Problem

When I try to stop and start a container again it fails. Why?

Solution

A: The shutdown() command is oneway

The shutdown() command accepted by the Manager (and by the Containers) to request a shutdown is a oneway method (since ORB::shutdown() will be called before invocation is completed - see also the explanation given in the Adv. C++ CORBA prog. book.).

Therefore the method returns immediately.

The Manager needs some time to actually shutdown ensuring that there are no pending activities. Therefore the Manager and JVM remain active several seconds after the shutdown() call has returned.

We will look for a better solution with ACS 4.0.

For the time being applications should check (if possible) for the Manager JVM in the process table or wait for some seconds (10 should be a reasonable value) before assuming that the Manager really shutdown.

Q: When I try to stop and start a container again it fails. Why?

A: Either the *acsStop** scripts have failed or more likely is that a developer component implementation fails to kill a thread it spawns.

The reason why *objexp* can be used to manipulate components that have been shutdown and restarted (by restarting a container) without restarting *obj exp* itself is because the components are persistent objects. This is accomplished by the *acsStartContainer* script assigning what is more or less a static TCP port to the container it runs. What does this have to do with the inability to restart a container? A lot believe it or not!

Much like the manager shutdown delay described above, the *acsStopContainer* command uses a CORBA *oneway* command to stop the container. That is, just because *acsStopContainer* returns control does not necessarily mean the container has really shutdown! Furthermore, the container has no real control over what threads are started and more importantly stopped by your component code. In a worst case scenario:

1. a component, **abc** living in container **xyz**, creates a thread at some point in time
2. *acsStopContainer xyz* is called
3. **xyz** calls the appropriate ZLegacy/ACS.LifeCycle methods of **abc** to destroy it but **abc** forgets to destroy the thread it created
4. **xyz** finishes executing and exits out of the *main* function
5. the process where **xyz** is running remains because of the thread started by **abc**
6. the TCP port remains tied up because the container actually uses a singleton ORB which does not really release the TCP port until the process dies (naturally or by the *kill* UNIX command)
7. *acsStartContainer -lang xyz* is called
8. **xyz** cannot be restarted using the quasi-static TCP port chosen by *acsStartContainer* because the first **xyz** process is still alive!

Nine out of ten times the scenario depicted above is what's really going on but there are indeed other possible culprits:

- the *acsStopContainer* script is broken. If you **do not** find a file named similarly to $ACSDATA/tmp /acs_local_log_maciContainerShutdown_somePID (where **somePID** is a process ID) after *acsStopContainer* returns control this could very well be the case. It's best to verify the name found in this file matches the name of the container you're trying to shutdown.
- manager receives the CORBA command from the *acsStopContainer* script but does not propagate the request to the container. There have never been any reports of this to date.
- the container receives the shutdown command from manager but fails to kill one of its **own** threads. There have been reports of this in the past and the problems have since been fixed.

Q: After a container segfaults and is restarted, *objexp* and other clients cannot seem to connect to components within the container. Why?

With ACS 4.1.1, we implemented extra logic into the *acsStartContainer* script itself to workaround the segfaulting components.

A: Even though the process segfaulted and control has been returned to the console, you must issue the *acsStopContainer* command to reclaim the TCP port

When C++ containers segfault as a result of poorly implemented components, you must run the *acsStopContainer* command if the container was started by the *acsStartContainer* script to reclaim the TCP port number. If you do not do this - the next time *acsStartContainer* is run it picks a new TCP port for the container. *objexp* as well as other clients of components use the old TCP port for the components causing CORBA no resources exceptions and it to appear like the container and components are broken when in fact they are not.

The detailed summary is the following:

1. C++ container segfaults at some time other than shutdown as a result of misbehaving component code
2. *acsStopContainer* is not issued after the segfault
3. *acsStartContainer -cpp ...* is run again. Since $ACSDATA/tmp/ACS_INSTANCE.$ACS_INSTANCE was not cleaned-up by running the *acsStop Container* command after the segfault, the script picks a new port for the container to run under
4. *objexp*, which has not been restarted since the container segfaulted and does not need to be, is used to try to manipulate the component(s). Obviously this is not going to work because the container is running under a new TCP port number and objexp assumes the component is running under the old one (as it should). Actually if you use "File=>Connect=>BACI", you will not see this problem from *objexp*
5. When you run *acsStop*, you see extra messages being emitted for the defunct container because the *acsStopContainer* command was never run which modifies an internal list of containers in ACS_INSTANCE.$ACS_INSTANCE. FYI, these extra messages are harmless.

-- DavidFugate - 17 Sep 2004

Related articles

- How can more people do development with ACS on the same machine without disturbing each other?
- Which ports are used by ACS?
- Problems connecting to ACS servers on a remote machine: bad /etc/hosts
- Why does the getComponent method of ZLegacy/ACS.ContainerServices return an object of type None?
- Why are some of my print statements not showing up in the container output section of acscommandcenter?